pandas: powerful Python data analysis toolkit

Release 0.21.0

Wes McKinney & PyData Development Team

Oct 27, 2017
## 1 What’s New

### 1.1 v0.21.0 (October 27, 2017)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 New features</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1.1 Integration with Apache Parquet file format</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1.2 <code>infer_objects</code> type conversion</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1.3 Improved warnings when attempting to create columns</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1.4 <code>drop</code> now also accepts index/columns keywords</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1.5 <code>rename</code>, <code>reindex</code> now also accept axis keyword</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1.6 <code>CategoricalDtype</code> for specifying categoricals</td>
<td>7</td>
</tr>
<tr>
<td>1.1.1.7 <code>GroupBy</code> objects now have a <code>pipe</code> method</td>
<td>8</td>
</tr>
<tr>
<td>1.1.1.8 <code>Categorical.rename_categories</code> accepts a dict-like</td>
<td>9</td>
</tr>
<tr>
<td>1.1.1.9 Other Enhancements</td>
<td>10</td>
</tr>
<tr>
<td>1.1.2 Backwards incompatible API changes</td>
<td>11</td>
</tr>
<tr>
<td>1.1.2.1 Dependencies have increased minimum versions</td>
<td>11</td>
</tr>
<tr>
<td>1.1.2.2 Sum/Prod of all-NaN Series/DataFrames is now consistently NaN</td>
<td>11</td>
</tr>
<tr>
<td>1.1.2.3 Indexing with a list with missing labels is Deprecated</td>
<td>12</td>
</tr>
<tr>
<td>1.1.2.4 NA naming Changes</td>
<td>13</td>
</tr>
<tr>
<td>1.1.2.5 Iteration of Series/Index will now return Python scalars</td>
<td>13</td>
</tr>
<tr>
<td>1.1.2.6 Indexing with a Boolean Index</td>
<td>14</td>
</tr>
<tr>
<td>1.1.2.7 <code>PeriodIndex</code> resampling</td>
<td>15</td>
</tr>
<tr>
<td>1.1.2.8 Improved error handling during item assignment in <code>pd.eval</code></td>
<td>17</td>
</tr>
<tr>
<td>1.1.2.9 Dtype Conversions</td>
<td>17</td>
</tr>
<tr>
<td>1.1.2.10 MultiIndex Constructor with a Single Level</td>
<td>18</td>
</tr>
<tr>
<td>1.1.2.11 UTC Localization with Series</td>
<td>19</td>
</tr>
<tr>
<td>1.1.2.12 Consistency of Range Functions</td>
<td>19</td>
</tr>
<tr>
<td>1.1.2.13 Other API Changes</td>
<td>20</td>
</tr>
<tr>
<td>1.1.3 Deprecations</td>
<td>21</td>
</tr>
<tr>
<td>1.1.3.1 <code>Series.select</code> and <code>DataFrame.select</code></td>
<td>22</td>
</tr>
<tr>
<td>1.1.3.2 <code>Series.argmax</code> and <code>Series.argmin</code></td>
<td>22</td>
</tr>
<tr>
<td>1.1.4 Removal of prior version deprecations/changes</td>
<td>23</td>
</tr>
<tr>
<td>1.1.5 Performance Improvements</td>
<td>23</td>
</tr>
<tr>
<td>1.1.6 Documentation Changes</td>
<td>23</td>
</tr>
<tr>
<td>1.1.7 Bug Fixes</td>
<td>23</td>
</tr>
<tr>
<td>1.1.7.1 Conversion</td>
<td>23</td>
</tr>
<tr>
<td>1.1.7.2 Indexing</td>
<td>24</td>
</tr>
<tr>
<td>1.1.7.3 I/O</td>
<td>25</td>
</tr>
<tr>
<td>1.1.7.4 Plotting</td>
<td>25</td>
</tr>
<tr>
<td>1.1.7.5 Groupby/Resample/Rolling</td>
<td>26</td>
</tr>
<tr>
<td>1.1.7.6 Sparse</td>
<td>26</td>
</tr>
<tr>
<td>1.1.7.7 Reshaping</td>
<td>26</td>
</tr>
</tbody>
</table>
1.1.7.8 Numeric ................................................................. 27
1.1.7.9 Categorical ......................................................... 27
1.1.7.10 PyPy ............................................................... 27
1.1.7.11 Other .............................................................. 28

1.2 v0.20.3 (July 7, 2017) ............................................... 28
1.2.1 Bug Fixes ............................................................. 28
1.2.1.1 Conversion ....................................................... 28
1.2.1.2 Indexing .......................................................... 28
1.2.1.3 I/O ................................................................. 29
1.2.1.4 Plotting .......................................................... 29
1.2.1.5 Reshaping ....................................................... 29
1.2.1.6 Categorical ..................................................... 29

1.3 v0.20.2 (June 4, 2017) ............................................... 29
1.3.1 Enhancements ....................................................... 30
1.3.2 Performance Improvements .................................... 30
1.3.3 Bug Fixes ............................................................. 30
1.3.3.1 Conversion ....................................................... 31
1.3.3.2 Indexing .......................................................... 31
1.3.3.3 I/O ................................................................. 31
1.3.3.4 Plotting .......................................................... 31
1.3.3.5 Groupby/Resample/Rolling .................................. 31
1.3.3.6 Sparse ........................................................... 32
1.3.3.7 Reshaping ....................................................... 32
1.3.3.8 Numeric .......................................................... 32
1.3.3.9 Categorical ..................................................... 32
1.3.3.10 Other ............................................................ 32

1.4 v0.20.1 (May 5, 2017) ............................................... 32
1.4.1 New features ......................................................... 35
1.4.1.1 agg API for DataFrame/Series ................................. 35
1.4.1.2 dtype keyword for data IO .................................. 36
1.4.1.3 .to_datetime() has gained an origin parameter ....... 37
1.4.1.4 Groupby Enhancements ...................................... 37
1.4.1.5 Better support for compressed URLs in read_csv .... 38
1.4.1.6 Pickle file I/O now supports compression ............... 38
1.4.1.7 UInt64 Support Improved .................................... 39
1.4.1.8 GroupBy on Categoricals .................................... 40
1.4.1.9 Table Schema Output ....................................... 41
1.4.1.10 SciPy sparse matrix from/to SparseDataFrame ....... 42
1.4.1.11 Excel output for styled DataFrames ...................... 42
1.4.1.12 IntervalIndex ............................................... 44
1.4.1.13 Other Enhancements ....................................... 45

1.4.2 Backwards incompatible API changes ....................... 47
1.4.2.1 Possible incompatibility for HDF5 formats created with pandas < 0.13.0 .......................... 47
1.4.2.2 Map on Index types now return other Index types .... 48
1.4.2.3 Accessing datetime fields of Index now return Index . 49
1.4.2.4 pd.unique will now be consistent with extension types ............................................. 50
1.4.2.5 S3 File Handling ............................................. 51
1.4.2.6 Partial String Indexing Changes ............................ 51
1.4.2.7 Concat of different float dtypes will not automatically upcast ..................................... 52
1.4.2.8 Pandas Google BigQuery support has moved .......... 52
1.4.2.9 Memory Usage for Index is more Accurate ............. 52
1.4.2.10 DataFrame.sort_index changes .......................... 53
1.4.2.11 Groupby Describe Formatting ............................. 55
1.4.2.12 Window Binary Corr/Cov operations return a MultiIndex DataFrame ......................... 56
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.19.2</td>
<td>December 24, 2016</td>
<td>Enhancements</td>
</tr>
<tr>
<td>v0.19.1</td>
<td>November 3, 2016</td>
<td>Performance Improvements</td>
</tr>
<tr>
<td>v0.19.0</td>
<td>October 2, 2016</td>
<td>Bug Fixes</td>
</tr>
</tbody>
</table>

### 1.5 Enhancements

- **merge_asof** for asof-style time-series joining
- **.rolling()** is now time-series aware
- **read_csv** has improved support for duplicate column names
- **read_csv** supports parsing **Categorical** directly
- Categorical Concatenation
- Semi-Month Offsets
- New Index methods
- Google BigQuery Enhancements
- Fine-grained numpy errstate
- **get_dummies** now returns integer dtypes
- Downcast values to smallest possible dtype in **to_numeric**
- pandas development API
- Other enhancements

### 1.6 API Changes

- **Series.tolist()** will now return Python types
- **Series** operators for different indexes
- **Series** type promotion on assignment

### 1.4.7 Bug Fixes

- Numeric
- Reshaping
- Plotting
- I/O
- Conversion

### 1.4.4 Deprecations

- Deprecate **.ix**
- Deprecate **Panel**
- Deprecate **groupby.agg()** with a dictionary when renaming
- Deprecate **.plotting**
- Other Deprecations

### 1.4.5 Removal of prior version deprecations/changes

### 1.4.6 Performance Improvements

### 1.4.3 Reorganization of the library: Privacy Changes

- **pandas.errors**
- **pandas.plotting**
- **pandas.testing**
- **pandas.errors**

### 1.4.2 API Changes

- **Semi-Month Offsets**
- **Categorical Concatenation**
- **Index.intersection** and **inner join** now preserve the order of the left Index
- **Pivot Table always returns a DataFrame**
- Other API Changes
1.7.2.4 .to_datetime() changes ........................................... 96
1.7.2.5 Merging changes ................................................. 96
1.7.2.6 .describe() changes ............................................ 98
1.7.2.7 Period changes .................................................. 99
1.7.2.8 Index +/- no longer used for set operations ................. 101
1.7.2.9 Index.difference and .symmetric_difference changes .. 101
1.7.2.10 Index.unique consistently returns Index ................. 102
1.7.2.11 MultiIndex constructors, groupby and set_index preserve categorical dtypes ............................. 102
1.7.2.12 read_csv will progressively enumerate chunks ........... 104
1.7.2.13 Sparse Changes ................................................. 105
1.7.2.14 Indexer dtype changes ........................................ 107
1.7.2.15 Other API Changes .............................................. 108
1.7.3 Deprecations .......................................................... 109
1.7.4 Removal of prior version deprecations/changes ............... 110
1.7.5 Performance Improvements ........................................ 110
1.7.6 Bug Fixes .............................................................. 111
1.8 v0.18.1 (May 3, 2016) .................................................. 117
1.8.1 New features .......................................................... 118
1.8.1.1 Custom Business Hour ........................................ 118
1.8.1.2 .groupby(..) syntax with window and resample operations ..................................................... 118
1.8.1.3 Method chaining improvements ................................ 120
1.8.1.4 Partial string indexing on DateTimeIndex when part of a MultiIndex ......................................... 122
1.8.1.5 Assembling Datetimes ............................................ 123
1.8.1.6 Other Enhancements .............................................. 124
1.8.2 Sparse changes ........................................................ 125
1.8.3 API changes .......................................................... 126
1.8.3.1 .groupby(..).nth() changes .................................. 126
1.8.3.2 numpy function compatibility .................................. 128
1.8.3.3 Using .apply on groupby resampling ......................... 128
1.8.3.4 Changes in read_csv exceptions .............................. 129
1.8.3.5 to_datetime error changes ...................................... 130
1.8.3.6 Other API changes .............................................. 130
1.8.3.7 Deprecations ...................................................... 131
1.8.4 Performance Improvements ......................................... 131
1.8.5 Bug Fixes .............................................................. 131
1.9 v0.18.0 (March 13, 2016) ................................................ 134
1.9.1 New features .......................................................... 136
1.9.1.1 Window functions are now methods ......................... 136
1.9.1.2 Changes to rename .............................................. 138
1.9.1.3 Range Index ....................................................... 138
1.9.1.4 Changes to str.extract ........................................ 139
1.9.1.5 Addition of str.extractall ..................................... 140
1.9.1.6 Changes to str.cat .............................................. 141
1.9.1.7 Datetimelike rounding .......................................... 141
1.9.1.8 Formatting of Integers in FloatIndex ....................... 143
1.9.1.9 Changes to dtypes assignment behaviors .................. 143
1.9.1.10 to_xarray .......................................................... 145
1.9.1.11 Latex Representation .......................................... 146
1.9.1.12 pd.read_sas() changes ........................................ 146
1.9.1.13 Other enhancements ........................................... 146
1.9.2 Backwards incompatible API changes .......................... 146
1.9.2.1 NaT and Timedelta operations ................................ 147
1.9.2.2 Changes to msgpack ............................................ 148
1.9.2.3 Signature change for .rank .................................... 149
1.12 v0.16.2 (June 12, 2015) ....................................................... 195
1.12.1 New features .............................................................. 195
1.12.1.1 Pipe ................................................................. 195
1.12.1.2 Other Enhancements ............................................. 197
1.12.2 API Changes .............................................................. 197
1.12.3 Performance Improvements ......................................... 197
1.12.4 Bug Fixes ................................................................. 197
1.13 v0.16.1 (May 11, 2015) ................................................... 199
1.13.1 Enhancements .............................................................. 199
1.13.1.1 CategoricalIndex .................................................. 199

1.10 v0.17.1 (November 21, 2015) ........................................ 164
1.10.1 New features .............................................................. 164
1.10.2 Enhancements .............................................................. 165
1.10.3 API changes ............................................................... 166
1.10.3.1 Deprecations .......................................................... 167
1.10.4 Performance Improvements ......................................... 167
1.10.5 Bug Fixes ................................................................. 167

1.9.2 v0.17.0 (October 9, 2015) ............................................. 169
1.11 New features ............................................................... 171
1.11.1 Datetime with TZ .......................................................... 171
1.11.1.1 Conditional HTML Formatting .................................. 172
1.11.1.2 Additional methods for dt accessor ............................. 173
1.11.1.3 Plot submethods .................................................... 173
1.11.1.4 Period Frequency Enhancement .................................. 175
1.11.1.5 Support for SAS XPORT files ................................... 175
1.11.1.6 Support for Math Functions in .eval() ......................... 176
1.11.1.7 Other API Changes ................................................. 178
1.11.1.8 Other enhancements ................................................ 178
1.11.2 Backwards incompatible API changes ............................ 182
1.11.2.1 Changes to sorting API ............................................ 182
1.11.2.2 Changes to to_datetime and to_timedelta ..................... 183
1.11.2.3 Changes to Index Comparisons .................................. 184
1.11.2.4 Changes to Boolean Comparisons vs. None ................. 185
1.11.2.5 HDFStore dropna behavior ...................................... 186
1.11.2.6 Changes to display.precision option .......................... 187
1.11.2.7 Changes to Categorical.unique .................................. 187
1.11.2.8 Changes to bool passed as header in Parsers ............... 188
1.11.2.9 Other API Changes ................................................ 188
1.11.2.10 Removal of prior version deprecations/changes .......... 190
1.11.3 Performance Improvements ......................................... 191
1.11.4 Bug Fixes ................................................................. 191

1.9.1 v0.16.1 (May 11, 2015) ................................................. 149
1.9.2 v0.16.2 (June 12, 2015) ................................................. 150
1.9.2.1 New features ............................................................. 150
1.9.2.2 Plot submethods ...................................................... 151
1.9.2.3 Additional methods for dt accessor ............................. 152
1.9.2.4 Bug in QuarterBegin with n=0 ................................. 153
1.9.2.5 Resample API .......................................................... 154
1.9.2.6 Changes to eval ...................................................... 154
1.9.2.7 Other API Changes .................................................. 156
1.9.2.8 Deprecations .......................................................... 157
1.9.2.9 Removal of deprecated float indexers .......................... 158
1.9.2.10 Removal of prior version deprecations/changes .......... 160
1.9.3 Performance Improvements ......................................... 161
1.9.4 Bug Fixes ................................................................. 161

1.10 v0.17.0 (October 9, 2015) ............................................. 164
1.10.1 New features .............................................................. 164
1.10.2 Enhancements .............................................................. 165
1.10.3 API changes ............................................................... 166
1.10.3.1 Deprecations .......................................................... 167
1.10.4 Performance Improvements ......................................... 167
1.10.5 Bug Fixes ................................................................. 167

1.11 New features ............................................................... 171
1.11.1 Datetime with TZ .......................................................... 171
1.11.1.1 Conditional HTML Formatting .................................. 172
1.11.1.2 Additional methods for dt accessor ............................. 173
1.11.1.3 Plot submethods .................................................... 173
1.11.1.4 Period Frequency Enhancement .................................. 175
1.11.1.5 Support for SAS XPORT files ................................... 175
1.11.1.6 Support for Math Functions in .eval() ......................... 176
1.11.1.7 Other API Changes ................................................. 178
1.11.1.8 Other enhancements ................................................ 178
1.11.2 Backwards incompatible API changes ............................ 182
1.11.2.1 Changes to sorting API ............................................ 182
1.11.2.2 Changes to to_datetime and to_timedelta ..................... 183
1.11.2.3 Changes to Index Comparisons .................................. 184
1.11.2.4 Changes to Boolean Comparisons vs. None ................. 185
1.11.2.5 HDFStore dropna behavior ...................................... 186
1.11.2.6 Changes to display.precision option .......................... 187
1.11.2.7 Changes to Categorical.unique .................................. 187
1.11.2.8 Changes to bool passed as header in Parsers ............... 188
1.11.2.9 Other API Changes ................................................ 188
1.11.2.10 Removal of prior version deprecations/changes .......... 190
1.11.3 Performance Improvements ......................................... 191
1.11.4 Bug Fixes ................................................................. 191

1.9.2 v0.17.0 (October 9, 2015) ............................................. 164
1.9.2.1 New features ............................................................. 164
1.9.2.2 Plot submethods ...................................................... 151
1.9.2.3 Additional methods for dt accessor ............................. 152
1.9.2.4 Bug in QuarterBegin with n=0 ................................. 153
1.9.2.5 Resample API .......................................................... 154
1.9.2.6 Changes to eval ...................................................... 154
1.9.2.7 Other API Changes .................................................. 156
1.9.2.8 Deprecations .......................................................... 157
1.9.2.9 Removal of deprecated float indexers .......................... 158
1.9.2.10 Removal of prior version deprecations/changes .......... 160
1.9.3 Performance Improvements ......................................... 161
1.9.4 Bug Fixes ................................................................. 161
## 1.19 v0.14.0 (May 31, 2014)
- **API changes**
- **Output Formatting Enhancements**
- **Prior Version Deprecations/Changes**
- **Deprecations**
- **Enhancements**
- **Performance**
- **Experimental**
- **Bug Fixes**

## 1.20 v0.13.1 (February 3, 2014)
- **API changes**
- **Output Formatting Enhancements**
- **Prior Version Deprecations/Changes**
- **Deprecations**
- **Enhancements**
- **Performance**
- **Experimental**
- **Bug Fixes**

## 1.21 v0.13.0 (January 3, 2014)
- **API changes**
- **Prior Version Deprecations/Changes**
- **Deprecations**
- **Indexing API Changes**
- **Float64Index API Change**
- **HDFStore API Changes**
- **DataFrame repr Changes**
- **Enhancements**
- **Experimental**
- **Internal Refactoring**
- **Bug Fixes**

## 1.22 v0.12.0 (July 24, 2013)
- **API changes**
- **I/O Enhancements**
- **Other Enhancements**
- **Experimental Features**
- **Bug Fixes**

## 1.23 v0.11.0 (April 22, 2013)
- **Selection Choices**
- **Selection Deprecations**
- **Dtypes**
- **Dtype Conversion**
- **Dtype Gotchas**
- **Datetimes Conversion**
- **API changes**
- **Enhancements**

## 1.24 v0.10.1 (January 22, 2013)
- **API changes**
1.24.2 New features .......................................................... 351
1.24.3 HDFStore ............................................................ 351
1.25 v0.10.0 (December 17, 2012) ........................................ 356
  1.25.1 File parsing new features ........................................ 356
  1.25.2 API changes ....................................................... 356
  1.25.3 New features ....................................................... 362
  1.25.4 DataFrame Printing ............................................... 362
  1.25.5 Updated PyTables Support ..................................... 364
  1.25.6 N Dimensional Panels (Experimental) ....................... 367
1.26 v0.9.1 (November 14, 2012) ....................................... 367
  1.26.1 New features ....................................................... 367
  1.26.2 API changes ....................................................... 371
1.27 v0.9.0 (October 7, 2012) .......................................... 371
  1.27.1 New features ....................................................... 372
  1.27.2 API changes ....................................................... 372
1.28 v0.8.1 (July 22, 2012) ............................................. 373
  1.28.1 New features ....................................................... 373
  1.28.2 Performance improvements ..................................... 374
1.29 v0.8.0 (June 29, 2012) ............................................. 374
  1.29.1 Support for non-unique indexes ................................ 374
  1.29.2 NumPy datetime64 dtype and 1.6 dependency ............... 374
  1.29.3 Time series changes and improvements ....................... 374
  1.29.4 Other new features ............................................. 375
  1.29.5 New plotting methods .......................................... 376
  1.29.6 Other API changes ............................................. 377
  1.29.7 Potential porting issues for pandas <= 0.7.3 users ........ 377
1.30 v0.7.3 (April 12, 2012) ........................................... 379
  1.30.1 New features ....................................................... 379
  1.30.2 NA Boolean Comparison API Change ......................... 381
  1.30.3 Other API Changes ............................................. 382
1.31 v0.7.2 (March 16, 2012) .......................................... 383
  1.31.1 New features ....................................................... 383
  1.31.2 Performance improvements ..................................... 383
1.32 v0.7.1 (February 29, 2012) ...................................... 383
  1.32.1 New features ....................................................... 383
  1.32.2 Performance improvements ..................................... 383
1.33 v0.7.0 (February 9, 2012) ....................................... 384
  1.33.1 New features ....................................................... 384
  1.33.2 API Changes to integer indexing .......................... 385
  1.33.3 API tweaks regarding label-based slicing ................ 386
  1.33.4 Changes to Series [ ] operator .............................. 387
  1.33.5 Other API Changes ............................................. 388
  1.33.6 Performance improvements ..................................... 388
1.34 v0.6.1 (December 13, 2011) .................................... 389
  1.34.1 New features ....................................................... 389
  1.34.2 Performance improvements ..................................... 390
1.35 v0.6.0 (November 25, 2011) .................................... 390
  1.35.1 New Features ....................................................... 390
  1.35.2 Performance Enhancements .................................... 391
1.36 v0.5.0 (October 24, 2011) ....................................... 391
  1.36.1 New Features ....................................................... 391
  1.36.2 Performance Enhancements .................................... 392
1.37 v0.4.3 through v0.4.1 (September 25 - October 9, 2011) .... 392
  1.37.1 New Features ....................................................... 392
2 Installation

2.1 Python version support

2.2 Installing pandas

2.2.1 Installing pandas with Anaconda

2.2.2 Installing pandas with Miniconda

2.2.3 Installing from PyPI

2.2.4 Installing using your Linux distribution’s package manager.

2.2.5 Installing from source

2.2.6 Running the test suite

2.3 Dependencies

2.3.1 Recommended Dependencies

2.3.2 Optional Dependencies

3 Contributing to pandas

3.1 Where to start?

3.2 Bug reports and enhancement requests

3.3 Working with the code

3.3.1 Version control, Git, and GitHub

3.3.2 Getting started with Git

3.3.3 Forking

3.3.4 Creating a branch

3.3.5 Creating a development environment

3.3.6 Creating a Windows development environment

3.3.7 Making changes

3.4 Contributing to the documentation

3.4.1 About the pandas documentation

3.4.2 How to build the pandas documentation

3.4.2.1 Requirements

3.4.2.2 Building the documentation

3.4.2.3 Building master branch documentation

3.5 Contributing to the code base

3.5.1 Code standards

3.5.1.1 C (cpplint)

3.5.1.2 Python (PEP8)

3.5.1.3 Backwards Compatibility

3.5.2 Testing With Continuous Integration

3.5.3 Test-driven development/code writing

3.5.3.1 Writing tests

3.5.3.2 Transitioning to pytest

3.5.3.3 Using pytest

3.5.4 Running the test suite

3.5.5 Running the performance test suite

3.5.6 Documenting your code

3.6 Contributing your changes to pandas

3.6.1 Committing your code

3.6.2 Combining commits

3.6.3 Pushing your changes

3.6.4 Review your code

3.6.5 Finally, make the pull request

3.6.6 Delete your merged branch (optional)

4 Package overview
4.1 Data Structures .......................................................... 419
  4.1.1 Why more than one data structure? ............................. 419
4.2 Mutability and copying of data ..................................... 420
4.3 Getting Support ....................................................... 420
4.4 Community ............................................................. 420
4.5 Project Governance ................................................. 420
4.6 Development Team ................................................. 420
4.7 Institutional Partners .............................................. 421
4.8 License ............................................................... 421

5 10 Minutes to pandas ................................................... 423
  5.1 Object Creation ..................................................... 423
  5.2 Viewing Data ....................................................... 425
  5.3 Selection ............................................................ 426
    5.3.1 Getting .......................................................... 426
    5.3.2 Selection by Label ............................................ 427
    5.3.3 Selection by Position ........................................ 428
    5.3.4 Boolean Indexing ............................................ 429
    5.3.5 Setting ........................................................ 430
  5.4 Missing Data ....................................................... 431
  5.5 Operations ......................................................... 432
    5.5.1 Stats ............................................................ 432
    5.5.2 Apply ............................................................ 433
    5.5.3 Histogramming ............................................... 433
    5.5.4 String Methods ............................................... 434
  5.6 Merge ............................................................... 434
    5.6.1 Concat .......................................................... 434
    5.6.2 Join ............................................................. 435
    5.6.3 Append ........................................................ 436
  5.7 Grouping ............................................................ 436
  5.8 Reshaping .......................................................... 437
    5.8.1 Stack ............................................................ 437
    5.8.2 Pivot Tables .................................................. 439
  5.9 Time Series ......................................................... 439
  5.10 Categoricals ....................................................... 441
  5.11 Plotting ............................................................ 443
  5.12 Getting Data In/Out .............................................. 444
    5.12.1 CSV ............................................................. 444
    5.12.2 HDF5 ........................................................... 445
    5.12.3 Excel .......................................................... 445
  5.13 Gotchas ............................................................. 446

6 Tutorials ................................................................. 447
  6.1 Internal Guides ..................................................... 447
  6.2 pandas Cookbook .................................................. 447
  6.3 Lessons for New pandas Users ................................... 448
  6.4 Practical data analysis with Python ............................. 448
  6.5 Exercises for New Users ......................................... 448
  6.6 Modern Pandas .................................................... 449
  6.7 Excel charts with pandas, vincent and xlsxwriter ............. 449
  6.8 Various Tutorials ................................................ 449

7 Cookbook ............................................................... 451
  7.1 Idioms .............................................................. 451
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.1</td>
<td>if-then...</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Splitting</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Building Criteria</td>
</tr>
<tr>
<td>7.2</td>
<td>Selection</td>
</tr>
<tr>
<td>7.2.1</td>
<td>DataFrames</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Panels</td>
</tr>
<tr>
<td>7.2.3</td>
<td>New Columns</td>
</tr>
<tr>
<td>7.3</td>
<td>MultiIndexing</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Slicing</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Sorting</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Levels</td>
</tr>
<tr>
<td>7.3.5</td>
<td>panelInd</td>
</tr>
<tr>
<td>7.4</td>
<td>Missing Data</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Replace</td>
</tr>
<tr>
<td>7.5</td>
<td>Grouping</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Expanding Data</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Splitting</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Pivot</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Apply</td>
</tr>
<tr>
<td>7.6</td>
<td>Timeseries</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Resampling</td>
</tr>
<tr>
<td>7.7</td>
<td>Merge</td>
</tr>
<tr>
<td>7.8</td>
<td>Plotting</td>
</tr>
<tr>
<td>7.9</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>7.9.1</td>
<td>CSV</td>
</tr>
<tr>
<td>7.9.1.1</td>
<td>Reading multiple files to create a single DataFrame</td>
</tr>
<tr>
<td>7.9.1.2</td>
<td>Parsing date components in multi-columns</td>
</tr>
<tr>
<td>7.9.1.3</td>
<td>Skip row between header and data</td>
</tr>
<tr>
<td>7.9.2</td>
<td>SQL</td>
</tr>
<tr>
<td>7.9.3</td>
<td>Excel</td>
</tr>
<tr>
<td>7.9.4</td>
<td>HTML</td>
</tr>
<tr>
<td>7.9.5</td>
<td>HDFStore</td>
</tr>
<tr>
<td>7.9.6</td>
<td>Binary Files</td>
</tr>
<tr>
<td>7.10</td>
<td>Computation</td>
</tr>
<tr>
<td>7.11</td>
<td>Timedeltas</td>
</tr>
<tr>
<td>7.12</td>
<td>Aliasing Axis Names</td>
</tr>
<tr>
<td>7.13</td>
<td>Creating Example Data</td>
</tr>
<tr>
<td>8</td>
<td>Intro to Data Structures</td>
</tr>
<tr>
<td>8.1</td>
<td>Series</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Series is ndarray-like</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Series is dict-like</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Vectorized operations and label alignment with Series</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Name attribute</td>
</tr>
<tr>
<td>8.2</td>
<td>DataFrame</td>
</tr>
<tr>
<td>8.2.1</td>
<td>From dict of Series or dicts</td>
</tr>
<tr>
<td>8.2.2</td>
<td>From dict of ndarrays / lists</td>
</tr>
<tr>
<td>8.2.3</td>
<td>From structured or record array</td>
</tr>
<tr>
<td>8.2.4</td>
<td>From a list of dicts</td>
</tr>
<tr>
<td>8.2.5</td>
<td>From a dict of tuples</td>
</tr>
<tr>
<td>8.2.6</td>
<td>From a Series</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Alternate Constructors</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Column selection, addition, deletion</td>
</tr>
</tbody>
</table>
9 Essential Basic Functionality

9.1 Head and Tail .......................................................... 513
9.2 Attributes and the raw ndarray(s) ................................ 514
9.3 Accelerated operations .............................................. 515
9.4 Flexible binary operations ......................................... 515
  9.4.1 Matching / broadcasting behavior .......................... 516
  9.4.2 Missing data / operations with fill values ............... 519
  9.4.3 Flexible Comparisons ........................................ 520
  9.4.4 Boolean Reductions ........................................... 520
  9.4.5 Comparing if objects are equivalent ...................... 521
  9.4.6 Comparing if objects are equivalent ...................... 522
  9.4.7 Combining overlapping data sets .......................... 523
  9.4.8 General DataFrame Combine ................................. 524
9.5 Descriptive statistics .............................................. 524
  9.5.1 Summarizing data: describe ................................ 527
  9.5.2 Index of Min/Max Values ................................... 529
  9.5.3 Value counts (histogramming) / Mode ..................... 530
  9.5.4 Discretization and quantiling .............................. 531
9.6 Function application .............................................. 532
  9.6.1 Tablewise Function Application ............................ 533
  9.6.2 Row or Column-wise Function Application ............... 534
  9.6.3 Aggregation API ............................................... 536
    9.6.3.1 Aggregating with multiple functions ................. 537
    9.6.3.2 Aggregating with a dict .............................. 538
    9.6.3.3 Mixed Dtypes ........................................ 538
    9.6.3.4 Custom describe ...................................... 539
  9.6.4 Transform API ................................................ 539
    9.6.4.1 Transform with multiple functions .................... 541
    9.6.4.2 Transforming with a dict ............................. 542
  9.6.5 Applying Elementwise Functions ........................... 542
  9.6.6 Applying with a Panel ....................................... 543
9.7 Reindexing and altering labels ................................ 545
  9.7.1 Reindexing to align with another object ................ 546
  9.7.2 Aligning objects with each other with align ............ 549
  9.7.3 Filling while reindexing .................................... 550

8.3 Panel ................................................................. 505
  8.3.1 From 3D ndarray with optional axis labels ............... 506
  8.3.2 From dict of DataFrame objects ............................ 506
  8.3.3 From DataFrame using to_panel method ..................... 507
  8.3.4 Item selection / addition / deletion ...................... 508
  8.3.5 Transposing .................................................. 508
  8.3.6 Indexing / Selection ......................................... 508
  8.3.7 Squeezing ..................................................... 509
  8.3.8 Conversion to DataFrame .................................... 510
8.4 Deprecate Panel .................................................. 510
8.5 Panel4D and PanelND (Deprecated) ............................. 512
16.4 Aggregation ................................................................. 756
  16.4.1 Applying multiple functions at once ......................... 758
  16.4.2 Applying different functions to DataFrame columns .......... 759
  16.4.3 Cython-optimized aggregation functions .................... 760
16.5 Transformation .......................................................... 760
  16.5.1 New syntax to window and resample operations ............ 765
16.6 Filtration ................................................................. 767
16.7 Dispatching to instance methods ..................................... 769
16.8 Flexible apply ............................................................. 770
16.9 Other useful features .................................................... 772
  16.9.1 Automatic exclusion of “nuisance” columns ............... 772
  16.9.2 NA and NaT group handling ..................................... 773
  16.9.3 Grouping with ordered factors ................................. 773
  16.9.4 Grouping with a Grouper specification ..................... 773
  16.9.5 Taking the first rows of each group ......................... 775
  16.9.6 Taking the nth row of each group ......................... 775
  16.9.7 Enumerate group items ....................................... 777
  16.9.8 Enumerate groups ............................................ 778
  16.9.9 Plotting ............................................................ 779
  16.9.10 Piping function calls ...................................... 780
16.10 Examples ................................................................. 780
  16.10.1 Regrouping by factor ........................................ 780
  16.10.2 Multi-column factorization .................................. 781
  16.10.3 Groupby by Indexer to ‘resample’ data .................... 782
  16.10.4 Returning a Series to propagate names .................... 782
17 Merge, join, and concatenate .......................................... 785
  17.1 Concatenating objects .............................................. 785
  17.1.1 Set logic on the other axes .................................. 788
  17.1.2 Concatenating using append ................................ 789
  17.1.3 Ignoring indexes on the concatenation axis ............ 791
  17.1.4 Concatenating with mixed ndims ........................... 792
  17.1.5 More concatenating with group keys ....................... 793
  17.1.6 Appending rows to a DataFrame ............................ 796
  17.2 Database-style DataFrame joining/merging ....................... 797
  17.2.1 Brief primer on merge methods (relational algebra) ..... 799
  17.2.2 Checking for duplicate keys ................................ 802
  17.2.3 The merge indicator .......................................... 802
  17.2.4 Merge Dtypes .................................................. 803
  17.2.5 Joining on index ............................................. 806
  17.2.6 Joining key columns on an index ........................... 807
  17.2.7 Joining a single Index to a Multi-index .................... 809
  17.2.8 Joining with two multi-indexes ............................. 810
  17.2.9 Overlapping value columns .................................. 811
  17.2.10 Joining multiple DataFrame or Panel objects ............ 812
  17.2.11 Merging together values within Series or DataFrame columns .... 812
  17.3 Timeseries friendly merging ...................................... 813
  17.3.1 Merging Ordered Data ........................................... 813
  17.3.2 Merging AsOf ................................................... 814
18 Reshaping and Pivot Tables .............................................. 817
  18.1 Reshaping by pivoting DataFrame objects ...................... 817
  18.2 Reshaping by stacking and unstacking ......................... 818
  18.2.1 Multiple Levels .................................................. 820
19 Time Series / Date functionality

19.1 Overview ................................................................. 838
19.2 Timestamps vs. Time Spans ........................................ 838
19.3 Converting to Timestamps .......................................... 840
   19.3.1 Providing a Format Argument ............................... 840
   19.3.2 Assembling Datetime from Multiple DataFrame Columns .......................... 841
   19.3.3 Invalid Data .................................................... 841
   19.3.4 Epoch Timestamps ............................................. 842
   19.3.5 From Timestamps to Epoch .................................. 842
   19.3.6 Using the origin Parameter ................................. 843
19.4 Generating Ranges of Timestamps .................................. 843
   19.4.1 Custom Frequency Ranges .................................... 845
19.5 Timestamp Limitations ............................................... 846
19.6 Indexing ................................................................. 846
   19.6.1 Partial String Indexing ....................................... 847
   19.6.2 Slice vs. Exact Match ......................................... 852
   19.6.3 Exact Indexing ................................................ 853
   19.6.4 Truncating & Fancy Indexing ................................. 854
19.7 Time/Date Components ............................................. 855
19.8 DateOffset Objects .................................................. 856
   19.8.1 Parametric Offsets ........................................... 858
   19.8.2 Using Offsets with Series / DatetimeIndex ................. 858
   19.8.3 Custom Business Days ....................................... 860
   19.8.4 Business Hour ............................................... 861
   19.8.5 Custom Business Hour ....................................... 863
   19.8.6 Offset Aliases ............................................... 864
   19.8.7 Combining Aliases ........................................... 864
   19.8.8 Anchored Offsets ............................................ 865
   19.8.9 Anchored Offset Semantics ................................ 866
   19.8.10 Holidays / Holiday Calendars .............................. 867
19.9 Time Series-Related Instance Methods ............................. 870
   19.9.1 Shifting / Lagging ........................................... 870
   19.9.2 Frequency Conversion ........................................ 871
   19.9.3 Filling Forward / Backward ................................ 871
   19.9.4 Converting to Python Datetimes ............................ 871
19.10 Resampling ............................................................ 872
   19.10.1 Basics ......................................................... 872
   19.10.2 Upsampling .................................................. 874
   19.10.3 Sparse Resampling .......................................... 875
   19.10.4 Aggregation .................................................. 876
19.11 Time Span Representation ......................................... 879
### 20 Time Deltas

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Parsing</td>
<td>897</td>
</tr>
<tr>
<td>20.1.1 to timedelta</td>
<td>898</td>
</tr>
<tr>
<td>20.1.2 Timedelta limitations</td>
<td>899</td>
</tr>
<tr>
<td>20.2 Operations</td>
<td>899</td>
</tr>
<tr>
<td>20.3 Reductions</td>
<td>903</td>
</tr>
<tr>
<td>20.4 Frequency Conversion</td>
<td>903</td>
</tr>
<tr>
<td>20.5 Attributes</td>
<td>905</td>
</tr>
<tr>
<td>20.6 TimedeltaIndex</td>
<td>906</td>
</tr>
<tr>
<td>20.6.1 Using the TimedeltaIndex</td>
<td>907</td>
</tr>
<tr>
<td>20.6.2 Operations</td>
<td>908</td>
</tr>
<tr>
<td>20.6.3 Conversions</td>
<td>909</td>
</tr>
<tr>
<td>20.7 Resampling</td>
<td>910</td>
</tr>
</tbody>
</table>

### 21 Categorical Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Object Creation</td>
<td>911</td>
</tr>
<tr>
<td>21.2 CategoricalDtype</td>
<td>914</td>
</tr>
<tr>
<td>21.2.1 Equality Semantics</td>
<td>915</td>
</tr>
<tr>
<td>21.3 Description</td>
<td>915</td>
</tr>
<tr>
<td>21.4 Working with categories</td>
<td>916</td>
</tr>
<tr>
<td>21.4.1 Renaming categories</td>
<td>917</td>
</tr>
<tr>
<td>21.4.2 Appending new categories</td>
<td>919</td>
</tr>
<tr>
<td>21.4.3 Removing categories</td>
<td>919</td>
</tr>
<tr>
<td>21.4.4 Removing unused categories</td>
<td>919</td>
</tr>
<tr>
<td>21.4.5 Setting categories</td>
<td>920</td>
</tr>
<tr>
<td>21.5 Sorting and Order</td>
<td>920</td>
</tr>
<tr>
<td>21.5.1 Reordering</td>
<td>922</td>
</tr>
<tr>
<td>21.5.2 Multi Column Sorting</td>
<td>923</td>
</tr>
<tr>
<td>21.6 Comparisons</td>
<td>923</td>
</tr>
<tr>
<td>21.7 Operations</td>
<td>926</td>
</tr>
<tr>
<td>21.8 Data munging</td>
<td>927</td>
</tr>
<tr>
<td>21.8.1 Getting</td>
<td>927</td>
</tr>
<tr>
<td>21.8.2 String and datetime accessors</td>
<td>928</td>
</tr>
<tr>
<td>21.8.3 Setting</td>
<td>930</td>
</tr>
<tr>
<td>21.8.4 Merging</td>
<td>931</td>
</tr>
<tr>
<td>21.8.5 Unioning</td>
<td>932</td>
</tr>
<tr>
<td>21.8.6 Concatenation</td>
<td>934</td>
</tr>
<tr>
<td>21.9 Getting Data In/Out</td>
<td>935</td>
</tr>
<tr>
<td>21.10 Missing Data</td>
<td>936</td>
</tr>
<tr>
<td>21.11 Differences to R’s factor</td>
<td>937</td>
</tr>
<tr>
<td>21.12 Gotchas</td>
<td>938</td>
</tr>
<tr>
<td>21.12.1 Memory Usage</td>
<td>938</td>
</tr>
</tbody>
</table>
### 22 Visualization

#### 22.1 Basic Plotting: `plot` .......................... 943
#### 22.2 Other Plots .................................. 946
  - 22.2.1 Bar plots ................................ 948
  - 22.2.2 Histograms ................................ 950
  - 22.2.3 Box Plots ................................ 956
  - 22.2.4 Area Plot ................................ 964
  - 22.2.5 Scatter Plot ................................ 966
  - 22.2.6 Hexagonal Bin Plot ......................... 970
  - 22.2.7 Pie plot ................................ 972
#### 22.3 Plotting with Missing Data ....................... 976
#### 22.4 Plotting Tools ................................ 977
  - 22.4.1 Scatter Matrix Plot ......................... 977
  - 22.4.2 Density Plot ................................ 978
  - 22.4.3 Andrews Curves .............................. 979
  - 22.4.4 Parallel Coordinates ......................... 980
  - 22.4.5 Lag Plot ................................ 981
  - 22.4.6 Autocorrelation Plot ......................... 982
  - 22.4.7 Bootstrap Plot ................................ 983
  - 22.4.8 RadViz ................................ 984
#### 22.5 Plot Formatting ................................ 985
  - 22.5.1 Setting the plot style ....................... 985
  - 22.5.2 General plot style arguments ............... 985
  - 22.5.3 Controlling the Legend ...................... 986
  - 22.5.4 Scales ................................ 987
  - 22.5.5 Plotting on a Secondary Y-axis ............. 988
  - 22.5.6 Suppressing Tick Resolution Adjustment .... 991
  - 22.5.7 Automatic Date Tick Adjustment ......... 994
  - 22.5.8 Subplots ................................ 994
  - 22.5.9 Using Layout and Targeting Multiple Axes .. 995
  - 22.5.10 Plotting With Error Bars .................. 998
  - 22.5.11 Plotting Tables ............................ 1000
  - 22.5.12 Colormaps ................................ 1003
#### 22.6 Plotting directly with matplotlib ............. 1008
#### 22.7 Trellis plotting interface ....................... 1009

### 23 Styling ............................................ 1011

#### 23.1 Building Styles ................................ 1011
  - 23.1.1 Building Styles Summary ..................... 1013
#### 23.2 Finer Control: Slicing ........................ 1014
#### 23.3 Finer Control: Display Values ................ 1014
#### 23.4 Built-in Styles ............................... 1015
  - 23.4.1 Bar charts ................................ 1015
#### 23.5 Sharing Styles ................................ 1017
#### 23.6 Other Options ................................ 1017
  - 23.6.1 Precision ................................ 1017
  - 23.6.2 Captions ................................ 1018
  - 23.6.3 Table Styles ................................ 1018
  - 23.6.4 CSS Classes ................................ 1018
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.6.5 Limitations</td>
<td>1019</td>
</tr>
<tr>
<td>23.6.6 Terms</td>
<td>1019</td>
</tr>
<tr>
<td>23.7 Fun stuff</td>
<td>1019</td>
</tr>
<tr>
<td>23.8 Export to Excel</td>
<td>1020</td>
</tr>
<tr>
<td>23.9 Extensibility</td>
<td>1021</td>
</tr>
<tr>
<td>23.9.1 Subclassing</td>
<td>1021</td>
</tr>
<tr>
<td>24 IO Tools (Text, CSV, HDF5, ...)</td>
<td>1023</td>
</tr>
<tr>
<td>24.1 CSV &amp; Text files</td>
<td>1023</td>
</tr>
<tr>
<td>24.1.1 Parsing options</td>
<td>1023</td>
</tr>
<tr>
<td>24.1.1.1 Basic</td>
<td>1024</td>
</tr>
<tr>
<td>24.1.1.2 Column and Index Locations and Names</td>
<td>1024</td>
</tr>
<tr>
<td>24.1.1.3 General Parsing Configuration</td>
<td>1025</td>
</tr>
<tr>
<td>24.1.1.4 NA and Missing Data Handling</td>
<td>1026</td>
</tr>
<tr>
<td>24.1.1.5 Datetime Handling</td>
<td>1026</td>
</tr>
<tr>
<td>24.1.1.6 Iteration</td>
<td>1027</td>
</tr>
<tr>
<td>24.1.1.7 Quoting, Compression, and File Format</td>
<td>1027</td>
</tr>
<tr>
<td>24.1.1.8 Error Handling</td>
<td>1028</td>
</tr>
<tr>
<td>24.1.2 Specifying column data types</td>
<td>1028</td>
</tr>
<tr>
<td>24.1.3 Specifying Categorical dtype</td>
<td>1030</td>
</tr>
<tr>
<td>24.1.4 Naming and Using Columns</td>
<td>1032</td>
</tr>
<tr>
<td>24.1.4.1 Handling column names</td>
<td>1032</td>
</tr>
<tr>
<td>24.1.5 Duplicate names parsing</td>
<td>1033</td>
</tr>
<tr>
<td>24.1.5.1 Filtering columns (usecols)</td>
<td>1033</td>
</tr>
<tr>
<td>24.1.6 Comments and Empty Lines</td>
<td>1034</td>
</tr>
<tr>
<td>24.1.6.1 Ignoring line comments and empty lines</td>
<td>1034</td>
</tr>
<tr>
<td>24.1.6.2 Comments</td>
<td>1036</td>
</tr>
<tr>
<td>24.1.7 Dealing with Unicode Data</td>
<td>1036</td>
</tr>
<tr>
<td>24.1.8 Index columns and trailing delimiters</td>
<td>1037</td>
</tr>
<tr>
<td>24.1.9 Date Handling</td>
<td>1038</td>
</tr>
<tr>
<td>24.1.9.1 Specifying Date Columns</td>
<td>1038</td>
</tr>
<tr>
<td>24.1.9.2 Date Parsing Functions</td>
<td>1040</td>
</tr>
<tr>
<td>24.1.9.3 Inferring Datetime Format</td>
<td>1041</td>
</tr>
<tr>
<td>24.1.9.4 International Date Formats</td>
<td>1042</td>
</tr>
<tr>
<td>24.1.10 Specifying method for floating-point conversion</td>
<td>1042</td>
</tr>
<tr>
<td>24.1.11 Thousand Separators</td>
<td>1043</td>
</tr>
<tr>
<td>24.1.12 NA Values</td>
<td>1043</td>
</tr>
<tr>
<td>24.1.13 Infinity</td>
<td>1044</td>
</tr>
<tr>
<td>24.1.14 Returning Series</td>
<td>1044</td>
</tr>
<tr>
<td>24.1.15 Boolean values</td>
<td>1044</td>
</tr>
<tr>
<td>24.1.16 Handling “bad” lines</td>
<td>1045</td>
</tr>
<tr>
<td>24.1.17 Dialect</td>
<td>1045</td>
</tr>
<tr>
<td>24.1.18 Quoting and Escape Characters</td>
<td>1046</td>
</tr>
<tr>
<td>24.1.19 Files with Fixed Width Columns</td>
<td>1047</td>
</tr>
<tr>
<td>24.1.20 Indexes</td>
<td>1049</td>
</tr>
<tr>
<td>24.1.20.1 Files with an “implicit” index column</td>
<td>1049</td>
</tr>
<tr>
<td>24.1.20.2 Reading an index with a MultiIndex</td>
<td>1049</td>
</tr>
<tr>
<td>24.1.20.3 Reading columns with a MultiIndex</td>
<td>1050</td>
</tr>
<tr>
<td>24.1.21 Automatically “sniffing” the delimiter</td>
<td>1051</td>
</tr>
<tr>
<td>24.1.22 Reading multiple files to create a single DataFrame</td>
<td>1052</td>
</tr>
<tr>
<td>24.1.23 Iterating through files chunk by chunk</td>
<td>1052</td>
</tr>
<tr>
<td>24.1.24 Specifying the parser engine</td>
<td>1053</td>
</tr>
<tr>
<td>24.1.25 Reading remote files</td>
<td>1053</td>
</tr>
<tr>
<td>24.1.26 Writing out Data</td>
<td>1054</td>
</tr>
</tbody>
</table>
24.1.26.1 Writing to CSV format ........................................... 1054
24.1.26.2 Writing a formatted string ..................................... 1054
24.2 JSON ......................................................................... 1055
24.2.1 Writing JSON .......................................................... 1055
  24.2.1.1 Orient Options ....................................................... 1056
  24.2.1.2 Date Handling ....................................................... 1057
  24.2.1.3 Fallback Behavior ............................................... 1058
24.2.2 Reading JSON .......................................................... 1059
  24.2.2.1 Data Conversion .................................................. 1060
  24.2.2.2 The Numpy Parameter ......................................... 1063
24.2.3 Normalization .......................................................... 1063
24.2.4 Line delimited json .................................................. 1064
24.2.5 Table Schema .......................................................... 1065
24.2.6 Reading JSON .......................................................... 1064
  24.2.6.1 The Numpy Parameter ......................................... 1065
  24.2.6.2 Data Conversion .................................................. 1066
24.2.7 Writing JSON .......................................................... 1067
  24.2.7.1 Orient Options ..................................................... 1068
  24.2.7.2 Date Handling ..................................................... 1069
  24.2.7.3 Fallback Behavior ............................................... 1070
24.3 HTML ......................................................................... 1068
24.3.1 Reading HTML Content .............................................. 1068
24.3.2 Writing to HTML files ................................................ 1072
24.3.3 HTML Table Parsing Gotchas ....................................... 1076
24.4 Excel files ................................................................. 1076
24.4.1 Reading Excel Files ................................................... 1077
  24.4.1.1 ExcelFile class ..................................................... 1077
  24.4.1.2 Specifying Sheets ................................................. 1077
  24.4.1.3 Reading a MultiIndex .......................................... 1078
  24.4.1.4 Parsing Specific Columns .................................... 1079
  24.4.1.5 Parsing Dates ...................................................... 1080
  24.4.1.6 Cell Converters ..................................................... 1080
  24.4.1.7 dtype Specifications ............................................. 1080
24.4.2 Writing Excel Files .................................................... 1080
  24.4.2.1 Writing Excel Files to Disk .................................. 1080
  24.4.2.2 Writing Excel Files to Memory ............................... 1081
24.4.3 Excel writer engines .................................................. 1081
24.4.4 Style and Formatting ................................................ 1082
24.5 Clipboard ................................................................. 1082
24.6 Pickling ..................................................................... 1083
24.6.1 Compressed pickle files .............................................. 1084
24.7 msgpack .................................................................... 1087
  24.7.1 Read/Write API ....................................................... 1089
24.8 HDF5 (PyTables) .......................................................... 1089
24.8.1 Read/Write API ........................................................ 1091
24.8.2 Fixed Format .......................................................... 1093
24.8.3 Table Format ........................................................... 1093
24.8.4 Hierarchical Keys ...................................................... 1094
24.8.5 Storing Types .......................................................... 1095
  24.8.5.1 Storing Mixed Types in a Table .............................. 1095
  24.8.5.2 Storing Multi-Index DataFrames ............................ 1096
24.8.6 Querying ................................................................. 1097
  24.8.6.1 Querying a Table ................................................ 1097
  24.8.6.2 Using timedelta64[ns] ........................................ 1101
  24.8.6.3 Indexing ............................................................. 1102
  24.8.6.4 Query via Data Columns .................................... 1103
  24.8.6.5 Iterator ............................................................. 1104
  24.8.6.6 Advanced Queries ............................................. 1106
  24.8.6.7 Multiple Table Queries ....................................... 1108
24.8.7 Delete from a Table ................................................... 1109
24.8.8 Notes & Caveats ....................................................... 1110
31 Comparison with R / R libraries
31.1 Quick Reference .......................................................... 1175
31.1.1 Querying, Filtering, Sampling ...................................... 1175
31.1.2 Sorting .................................................................. 1176
31.1.3 Transforming ......................................................... 1176
31.1.4 Grouping and Summarizing ....................................... 1176
31.2 Base R .................................................................. 1176
31.2.1 Slicing with R’s c ..................................................... 1176
31.2.2 aggregate ............................................................. 1178
31.2.3 match/%in% ......................................................... 1179
31.2.4 tapply ............................................................... 1179
31.2.5 subset ............................................................... 1180
31.2.6 with ................................................................. 1181
31.3 plyr ........................................................................ 1181
31.3.1 ddply ............................................................... 1182
31.4 reshape / reshape2 .......................................................... 1183
31.4.1 melt.array .......................................................... 1183
31.4.2 melt.list ........................................................... 1183
31.4.3 melt.data.frame .................................................. 1184
31.4.4 cast ............................................................... 1184
31.4.5 factor ............................................................. 1186

32 Comparison with SQL
32.1 SELECT ................................................................ 1187
32.2 WHERE ................................................................ 1188
32.3 GROUP BY ............................................................. 1190
32.4 JOIN ................................................................ 1192
32.4.1 INNER JOIN ....................................................... 1192
32.4.2 LEFT OUTER JOIN ............................................. 1193
32.4.3 RIGHT JOIN .......................................................... 1193
32.4.4 FULL JOIN ............................................................. 1193
32.5 UNION ................................................................ 1194
32.6 Pandas equivalents for some SQL analytic and aggregate functions ................................................................. 1195
32.6.1 Top N rows with offset ......................................... 1195
32.6.2 Top N rows per group ........................................... 1195
32.7 UPDATE ............................................................... 1197
32.8 DELETE ............................................................... 1197

33 Comparison with SAS
33.1 Data Structures ............................................................. 1199
33.1.1 General Terminology Translation .............................. 1199
33.1.2 DataFrame/Series ................................................. 1199
33.1.3 Index ............................................................... 1200
33.2 Data Input / Output ..................................................... 1200
33.2.1 Constructing a DataFrame from Values ..................... 1200
33.2.2 Reading External Data ........................................... 1200
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.2.3</td>
<td>Exporting Data</td>
<td>1201</td>
</tr>
<tr>
<td>33.3</td>
<td>Data Operations</td>
<td>1201</td>
</tr>
<tr>
<td>33.3.1</td>
<td>Operations on Columns</td>
<td>1201</td>
</tr>
<tr>
<td>33.3.2</td>
<td>Filtering</td>
<td>1202</td>
</tr>
<tr>
<td>33.3.3</td>
<td>If/Then Logic</td>
<td>1202</td>
</tr>
<tr>
<td>33.3.4</td>
<td>Date Functionality</td>
<td>1203</td>
</tr>
<tr>
<td>33.3.5</td>
<td>Selection of Columns</td>
<td>1204</td>
</tr>
<tr>
<td>33.3.6</td>
<td>Sorting by Values</td>
<td>1205</td>
</tr>
<tr>
<td>33.4</td>
<td>String Processing</td>
<td>1205</td>
</tr>
<tr>
<td>33.4.1</td>
<td>Length</td>
<td>1205</td>
</tr>
<tr>
<td>33.4.2</td>
<td>Find</td>
<td>1206</td>
</tr>
<tr>
<td>33.4.3</td>
<td>Substring</td>
<td>1206</td>
</tr>
<tr>
<td>33.4.4</td>
<td>Scan</td>
<td>1206</td>
</tr>
<tr>
<td>33.4.5</td>
<td>Upcase, Lowcase, and Propcase</td>
<td>1207</td>
</tr>
<tr>
<td>33.5</td>
<td>Merging</td>
<td>1207</td>
</tr>
<tr>
<td>33.6</td>
<td>Missing Data</td>
<td>1209</td>
</tr>
<tr>
<td>33.7</td>
<td>GroupBy</td>
<td>1211</td>
</tr>
<tr>
<td>33.7.1</td>
<td>Aggregation</td>
<td>1211</td>
</tr>
<tr>
<td>33.7.2</td>
<td>Transformation</td>
<td>1211</td>
</tr>
<tr>
<td>33.7.3</td>
<td>By Group Processing</td>
<td>1212</td>
</tr>
<tr>
<td>33.8</td>
<td>Other Considerations</td>
<td>1213</td>
</tr>
<tr>
<td>33.8.1</td>
<td>Disk vs Memory</td>
<td>1213</td>
</tr>
<tr>
<td>33.8.2</td>
<td>Data Interop</td>
<td>1213</td>
</tr>
</tbody>
</table>

34 API Reference |
34.1 Input/Output |
34.1.1 Pickling |
34.1.1.1 pandas.read_pickle |
34.1.2 Flat File |
34.1.2.1 pandas.read_table |
34.1.2.2 pandas.read_csv |
34.1.2.3 pandas.read_fwf |
34.1.2.4 pandas.read_msgpack |
34.1.3 Clipboard |
34.1.3.1 pandas.read_clipboard |
34.1.4 Excel |
34.1.4.1 pandas.read_excel |
34.1.4.2 pandas.ExcelFile.parse |
34.1.5 JSON |
34.1.5.1 pandas.read_json |
34.1.5.2 pandas.io.json.json_normalize |
34.1.5.3 pandas.io.json.build_table_schema |
34.1.6 HTML |
34.1.6.1 pandas.read_html |
34.1.7 HDFStore: PyTables (HDF5) |
34.1.7.1 pandas.read_hdf |
34.1.7.2 pandas.HDFStore.put |
34.1.7.3 pandas.HDFStore.append |
34.1.7.4 pandas.HDFStore.get |
34.1.7.5 pandas.HDFStore.select |
34.1.7.6 pandas.HDFStore.info |
34.1.8 Feather |
34.1.8.1 pandas.read_feather |
34.1.9 Parquet |
34.3.14.3 pandas.Series.dt.to_pytdedelta
34.3.14.4 pandas.Series.dt.total_seconds
34.3.14 String handling
34.3.14.1 pandas.Series.str.capitalize
34.3.14.2 pandas.Series.str.cat
34.3.14.3 pandas.Series.str.center
34.3.14.4 pandas.Series.str.contains
34.3.14.5 pandas.Series.str.decode
34.3.14.6 pandas.Series.str.decode
34.3.14.7 pandas.Series.str.encode
34.3.14.8 pandas.Series.str.endswith
34.3.14.9 pandas.Series.str.extract
34.3.14.10 pandas.Series.str.extractall
34.3.14.11 pandas.Series.str.find
34.3.14.12 pandas.Series.str.get
34.3.14.13 pandas.Series.str.findall
34.3.14.14 pandas.Series.str.index
34.3.14.15 pandas.Series.str.join
34.3.14.16 pandas.Series.str.len
34.3.14.17 pandas.Series.str.ljust
34.3.14.18 pandas.Series.str.lower
34.3.14.19 pandas.Series.str.lstrip
34.3.14.20 pandas.Series.str.match
34.3.14.21 pandas.Series.str.pad
34.3.14.22 pandas.Series.str.partition
34.3.14.23 pandas.Series.str.pad
34.3.14.24 pandas.Series.str.repeat
34.3.14.25 pandas.Series.str.replace
34.3.14.26 pandas.Series.str.rfind
34.3.14.27 pandas.Series.str.rindex
34.3.14.28 pandas.Series.str.rjust
34.3.14.29 pandas.Series.str.rpartition
34.3.14.30 pandas.Series.str.rstrip
34.3.14.31 pandas.Series.str.rsplit
34.3.14.32 pandas.Series.str.split
34.3.14.33 pandas.Series.str.slice_replace
34.3.14.34 pandas.Series.str.slice
34.3.14.35 pandas.Series.str.split
34.3.14.36 pandas.Series.str.split
34.3.14.37 pandas.Series.str.swapcase
34.3.14.38 pandas.Series.str.title
34.3.14.39 pandas.Series.str.translate
34.3.14.40 pandas.Series.str.upper
34.3.14.41 pandas.Series.str.wrap
34.3.14.42 pandas.Series.str.zfill
34.3.14.43 pandas.Series.str.isalnum
34.3.14.44 pandas.Series.str.isalpha
34.3.14.45 pandas.Series.str.isdecimal
34.3.14.46 pandas.Series.str.isdigit
34.3.14.47 pandas.Series.str.islower
34.3.14.48 pandas.Series.str.isupper
34.3.14.49 pandas.Series.str.istitle
34.3.14.50 pandas.Series.str.isnumeric
34.3.14.51 pandas.Series.str.isdecimal
34.4.13.8 pandas.DataFrame.plot.kde ........................................... 1624
34.4.13.9 pandas.DataFrame.plot.line ........................................... 1625
34.4.13.10 pandas.DataFrame.plot.pie ........................................... 1625
34.4.13.11 pandas.DataFrame.plot.scatter ..................................... 1625
34.4.14 Serialization / IO / Conversion ........................................ 1626
34.4.15 Sparse ........................................................................ 1626
34.4.15.1 pandas.SparseDataFrame.to_coo ....................................... 1626

34.5 Panel ........................................................................... 1627
34.5.1 Constructor .................................................................. 1627
34.5.1.1 pandas.Panel .............................................................. 1627
34.5.2 Attributes and underlying data ......................................... 1713
34.5.3 Conversion .................................................................... 1713
34.5.4 Getting and setting ......................................................... 1714
34.5.5 Indexing, iteration, slicing ................................................ 1714
34.5.5.1 pandas.Panel.__iter__ .................................................. 1714
34.5.6 Binary operator functions ................................................ 1714
34.5.7 Function application, GroupBy .......................................... 1715
34.5.8 Computations / Descriptive Stats .................................... 1715
34.5.9 Reindexing / Selection / Label manipulation ...................... 1716
34.5.10 Missing data handling .................................................... 1716
34.5.11 Reshaping, sorting, transposing ....................................... 1716
34.5.12 Combining / joining / merging ......................................... 1717
34.5.13 Time series-related ....................................................... 1717
34.5.14 Serialization / IO / Conversion ....................................... 1717

34.6 Index ........................................................................... 1717
34.6.1 pandas.Index ................................................................. 1718
34.6.1.1 pandas.Index.T .............................................................. 1719
34.6.1.2 pandas.Index.asi8 .......................................................... 1719
34.6.1.3 pandas.Index.base .......................................................... 1719
34.6.1.4 pandas.Index.data .......................................................... 1719
34.6.1.5 pandas.Index.dtype .......................................................... 1719
34.6.1.6 pandas.Index.dtype_str ................................................... 1720
34.6.1.7 pandas.Index.empty ........................................................ 1720
34.6.1.8 pandas.Index.flags ........................................................ 1720
34.6.1.9 pandas.Index.has_duplicates ........................................ 1720
34.6.1.10 pandas.Index.hasnans .................................................... 1720
34.6.1.11 pandas.Index.inferred_type ........................................... 1720
34.6.1.12 pandas.Index.is_all_dates ............................................. 1720
34.6.1.13 pandas.Index.is_monotonic .......................................... 1720
34.6.1.14 pandas.Index.is_monotonic_decreasing ............................ 1720
34.6.1.15 pandas.Index.is_monotonic_increasing ............................ 1721
34.6.1.16 pandas.Index.is_unique ............................................... 1721
34.6.1.17 pandas.Index.itemsize ................................................... 1721
34.6.1.18 pandas.Index.name ...................................................... 1721
34.6.1.19 pandas.Index.names ..................................................... 1721
34.6.1.20 pandas.Index.nbytes .................................................... 1721
34.6.1.21 pandas.Index.ndim ....................................................... 1721
34.6.1.22 pandas.Index.nlevels ................................................... 1721
34.6.1.23 pandas.Index.shape ..................................................... 1722
34.6.1.24 pandas.Index.size ....................................................... 1722
34.6.1.25 pandas.Index.strides .................................................... 1722
34.6.1.26 pandas.Index.values ..................................................... 1722
34.6.1.27 pandas.Index.all ........................................................ 1724
34.6.1.28 pandas.Index.any ........................................................ 1724

xxx
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.TimedeltaIndex.factorize</td>
<td>1844</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.fillna</td>
<td>1845</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.floor</td>
<td>1845</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.format</td>
<td>1845</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_duplicates</td>
<td>1845</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_indexer</td>
<td>1845</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_indexer_for</td>
<td>1846</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_indexer_non_unique</td>
<td>1846</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_level_values</td>
<td>1846</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_loc</td>
<td>1847</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_slice_bound</td>
<td>1847</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_value</td>
<td>1847</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.get_values</td>
<td>1847</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.groupby</td>
<td>1847</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.holds_integer</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.identical</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.intersection</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_boolean</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_categorical</td>
<td>1848</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_float</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_integer</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_interval</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_lexsorted_for_tuple</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_mixed</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_numeric</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_object</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.is_type_compatible</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.isin</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.isna</td>
<td>1849</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.isnull</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.item</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.join</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.map</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.max</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.memory_usage</td>
<td>1850</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.min</td>
<td>1851</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.notna</td>
<td>1851</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.notnull</td>
<td>1851</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.uniq</td>
<td>1851</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.putmask</td>
<td>1852</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.ravel</td>
<td>1852</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.reindex</td>
<td>1852</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.rename</td>
<td>1852</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.repeat</td>
<td>1852</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.reshape</td>
<td>1853</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.round</td>
<td>1853</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.searchsorted</td>
<td>1853</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.set_names</td>
<td>1854</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.set_value</td>
<td>1855</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.set_value</td>
<td>1855</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.shift</td>
<td>1855</td>
</tr>
<tr>
<td>pandas.TimedeltaIndex.slice_indexer</td>
<td>1855</td>
</tr>
</tbody>
</table>
34.18.1.1 pandas.io.formats.style.Styler ........................................... 1947
34.18.2 Style Application ....................................................................... 1959
34.18.3 Built-in Styles ......................................................................... 1960
34.18.4 Style Export and Import ............................................................. 1960
34.19 General utility functions ............................................................... 1960
34.19.1 Working with options ................................................................. 1960
  34.19.1.1 pandas.describe_option ......................................................... 1961
  34.19.1.2 pandas.reset_option ............................................................. 1964
  34.19.1.3 pandas.get_option ............................................................... 1968
  34.19.1.4 pandas.set_option ............................................................... 1971
  34.19.1.5 pandas.option_context ......................................................... 1975
34.19.2 Testing functions ...................................................................... 1975
  34.19.2.1 pandas.testing.assert_frame_equal ........................................ 1975
  34.19.2.2 pandas.testing.assert_series_equal ...................................... 1976
  34.19.2.3 pandas.testing.assert_index_equal ....................................... 1977
34.19.3 Exceptions and warnings ........................................................... 1978
  34.19.3.1 pandas.errors.DtypeWarning .............................................. 1978
  34.19.3.2 pandas.errors.EmptyDataError ........................................... 1978
  34.19.3.3 pandas.errors.OutOfBoundsDatetime ..................................... 1978
  34.19.3.4 pandas.errors.ParserError .................................................. 1978
  34.19.3.5 pandas.errors.ParserWarning .............................................. 1978
  34.19.3.6 pandas.errors.PerformanceWarning ...................................... 1979
  34.19.3.7 pandas.errors.UnsupportedFunctionCall ................................ 1979
  34.19.3.8 pandas.errors.UnsortedIndexError ...................................... 1979
34.19.4 Data types related functionality .................................................. 1979
  34.19.4.1 pandas.api.types.union_categoricals .................................... 1979
  34.19.4.2 pandas.api.types.infer_dtype .............................................. 1981
  34.19.4.3 pandas.api.types.pandas_dtype ........................................... 1983
  34.19.4.4 pandas.api.types.is_bool_dtype ......................................... 1984
  34.19.4.5 pandas.api.types.is_categorical_dtype .................................. 1984
  34.19.4.6 pandas.api.types.is_complex_dtype ..................................... 1985
  34.19.4.7 pandas.api.types.is_datetime64_any_dtype ............................. 1985
  34.19.4.8 pandas.api.types.is_datetime64_dtype .................................. 1986
  34.19.4.9 pandas.api.types.is_datetime64_ns_dtype ............................. 1986
  34.19.4.10 pandas.api.types.is_datetime64tz_dtype ............................. 1987
  34.19.4.11 pandas.api.types.is_extension_type .................................... 1988
  34.19.4.12 pandas.api.types.is_float_dtype ........................................ 1988
  34.19.4.13 pandas.api.types.is_integer_dtype ..................................... 1989
  34.19.4.14 pandas.api.types.is_int64_dtype ........................................ 1990
  34.19.4.15 pandas.api.types.is_interval_dtype .................................... 1990
  34.19.4.16 pandas.api.types.is_numeric_dtype .................................... 1991
  34.19.4.17 pandas.api.types.is_object_dtype ...................................... 1991
  34.19.4.18 pandas.api.types.is_period_dtype ...................................... 1992
  34.19.4.19 pandas.api.types.is_period_dtype ...................................... 1992
  34.19.4.20 pandas.api.types.is_string_dtype ...................................... 1993
  34.19.4.21 pandas.api.types.is_timedelta64_dtype ................................. 1993
  34.19.4.22 pandas.api.types.is_timedelta64_ns_dtype ................................ 1994
  34.19.4.23 pandas.api.types.is_unsigned_integer_dtype ........................ 1994
  34.19.4.24 pandas.api.types.is_timedelta64_dtype ................................ 1995
  34.19.4.25 pandas.api.types.is_timedelta64_ns_dtype ................................ 1996
  34.19.4.26 pandas.api.types.is_complex_dtype .................................... 1996
  34.19.4.27 pandas.api.types.is_list_like .......................................... 1997
  34.19.4.28 pandas.api.types.pandas_dtype ........................................ 1997
  34.19.4.29 pandas.api.types.is_iterator ........................................... 1997
<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.15</td>
<td>pandas 0.15.0</td>
<td>2051</td>
</tr>
<tr>
<td>37.15.1</td>
<td>Thanks</td>
<td>2052</td>
</tr>
<tr>
<td>37.16</td>
<td>pandas 0.14.1</td>
<td>2054</td>
</tr>
<tr>
<td>37.16.1</td>
<td>Thanks</td>
<td>2054</td>
</tr>
<tr>
<td>37.17</td>
<td>pandas 0.14.0</td>
<td>2056</td>
</tr>
<tr>
<td>37.17.1</td>
<td>Thanks</td>
<td>2056</td>
</tr>
<tr>
<td>37.18</td>
<td>pandas 0.13.1</td>
<td>2059</td>
</tr>
<tr>
<td>37.18.1</td>
<td>New Features</td>
<td>2059</td>
</tr>
<tr>
<td>37.18.2</td>
<td>API Changes</td>
<td>2059</td>
</tr>
<tr>
<td>37.18.3</td>
<td>Experimental Features</td>
<td>2059</td>
</tr>
<tr>
<td>37.18.4</td>
<td>Improvements to existing features</td>
<td>2059</td>
</tr>
<tr>
<td>37.18.5</td>
<td>Bug Fixes</td>
<td>2060</td>
</tr>
<tr>
<td>37.19</td>
<td>pandas 0.13.0</td>
<td>2062</td>
</tr>
<tr>
<td>37.19.1</td>
<td>New Features</td>
<td>2062</td>
</tr>
<tr>
<td>37.19.2</td>
<td>Experimental Features</td>
<td>2062</td>
</tr>
<tr>
<td>37.19.3</td>
<td>Improvements to existing features</td>
<td>2063</td>
</tr>
<tr>
<td>37.19.4</td>
<td>API Changes</td>
<td>2065</td>
</tr>
<tr>
<td>37.19.5</td>
<td>Internal Refactoring</td>
<td>2068</td>
</tr>
<tr>
<td>37.19.6</td>
<td>Bug Fixes</td>
<td>2070</td>
</tr>
<tr>
<td>37.20</td>
<td>pandas 0.12.0</td>
<td>2076</td>
</tr>
<tr>
<td>37.20.1</td>
<td>New Features</td>
<td>2076</td>
</tr>
<tr>
<td>37.20.2</td>
<td>Improvements to existing features</td>
<td>2076</td>
</tr>
<tr>
<td>37.20.3</td>
<td>API Changes</td>
<td>2077</td>
</tr>
<tr>
<td>37.20.4</td>
<td>Experimental Features</td>
<td>2079</td>
</tr>
<tr>
<td>37.20.5</td>
<td>Bug Fixes</td>
<td>2079</td>
</tr>
<tr>
<td>37.21</td>
<td>pandas 0.11.0</td>
<td>2082</td>
</tr>
<tr>
<td>37.21.1</td>
<td>New Features</td>
<td>2082</td>
</tr>
<tr>
<td>37.21.2</td>
<td>Improvements to existing features</td>
<td>2083</td>
</tr>
<tr>
<td>37.21.3</td>
<td>API Changes</td>
<td>2085</td>
</tr>
<tr>
<td>37.21.4</td>
<td>Bug Fixes</td>
<td>2086</td>
</tr>
<tr>
<td>37.22</td>
<td>pandas 0.10.1</td>
<td>2088</td>
</tr>
<tr>
<td>37.22.1</td>
<td>New Features</td>
<td>2089</td>
</tr>
<tr>
<td>37.22.2</td>
<td>API Changes</td>
<td>2089</td>
</tr>
<tr>
<td>37.22.3</td>
<td>Improvements to existing features</td>
<td>2089</td>
</tr>
<tr>
<td>37.22.4</td>
<td>Bug Fixes</td>
<td>2090</td>
</tr>
<tr>
<td>37.23</td>
<td>pandas 0.10.0</td>
<td>2091</td>
</tr>
<tr>
<td>37.23.1</td>
<td>New Features</td>
<td>2091</td>
</tr>
<tr>
<td>37.23.2</td>
<td>Experimental Features</td>
<td>2092</td>
</tr>
<tr>
<td>37.23.3</td>
<td>API Changes</td>
<td>2092</td>
</tr>
<tr>
<td>37.23.4</td>
<td>Improvements to existing features</td>
<td>2093</td>
</tr>
<tr>
<td>37.23.5</td>
<td>Bug Fixes</td>
<td>2094</td>
</tr>
<tr>
<td>37.24</td>
<td>pandas 0.9.1</td>
<td>2095</td>
</tr>
<tr>
<td>37.24.1</td>
<td>New Features</td>
<td>2096</td>
</tr>
<tr>
<td>37.24.2</td>
<td>API Changes</td>
<td>2096</td>
</tr>
<tr>
<td>37.24.3</td>
<td>Improvements to existing features</td>
<td>2096</td>
</tr>
<tr>
<td>37.24.4</td>
<td>Bug Fixes</td>
<td>2096</td>
</tr>
<tr>
<td>37.25</td>
<td>pandas 0.9.0</td>
<td>2098</td>
</tr>
<tr>
<td>37.25.1</td>
<td>New Features</td>
<td>2098</td>
</tr>
<tr>
<td>37.25.2</td>
<td>Improvements to existing features</td>
<td>2098</td>
</tr>
<tr>
<td>37.25.3</td>
<td>API Changes</td>
<td>2099</td>
</tr>
<tr>
<td>37.25.4</td>
<td>Bug Fixes</td>
<td>2099</td>
</tr>
<tr>
<td>37.26</td>
<td>pandas 0.8.1</td>
<td>2103</td>
</tr>
<tr>
<td>37.26.1</td>
<td>New Features</td>
<td>2103</td>
</tr>
<tr>
<td>37.26.2</td>
<td>Improvements to existing features</td>
<td>2103</td>
</tr>
</tbody>
</table>
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 actually 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-floating point 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, moving window linear regressions, date shifting and lagging, etc.
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.**

**Note:** This documentation assumes general familiarity with NumPy. If you haven’t used NumPy much or at all, do invest some time in learning about NumPy first.

See the package overview for more detail about what’s in the library.
CHAPTER ONE

WHAT’S NEW

These are new features and improvements of note in each release.

1.1 v0.21.0 (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, 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
  – `infer_objects` type conversion
  – Improved warnings when attempting to create columns
  – `drop` now also accepts index/columns keywords
  – `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 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**
  - **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**
    - **I/O**
    - **Plotting**
    - **Groupby/Resample/Rolling**
    - **Sparse**
    - **Reshaping**
    - **Numeric**
    - **Categorical**
    - `PYPY`
    - **Other**
1.1.1 New features

1.1.1.1 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 see the IO docs on Parquet.

1.1.1.2 `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 for more details. (GH11221)

This method only performs soft conversions on object columns, converting Python objects to native types, but not any coercive conversions. For example:

```python
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
dtype: object

In [3]: df.infer_objects().dtypes
Out[3]:
A    int64
B    int64
C     object
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()`).

```python
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
dtype: object
```
1.1.1.3 Improved warnings when attempting to create columns

New users are often puzzled by the relationship between column operations and attribute access on DataFrame instances (GH7175). One specific instance of this confusion is attempting to create a new column by setting an attribute on the DataFrame:

```python
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:

```python
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.

1.1.1.4 `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.drop(['B', 'C'], axis=1)
Out[8]:
   A  D
0  0  3
1  4  7
```

# the following is now equivalent
```python
In [10]: df.drop(columns=['B', 'C'])
Out[10]:
   A  D
0  0  3
1  4  7
```

1.1.1.5 `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:

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

In [13]: df.rename(id, axis='index')
A  B
4453153456  1  4
4453153488  2  5
4453153520  3  6

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

In [15]: df.reindex([0, 1, 3], columns=['A', 'B', 'C'])
A  B  C
0  1.0  4.0  NaN
1  2.0  5.0  NaN
3  NaN  NaN  NaN

The “index, columns” style continues to work as before.

In [16]: df.rename(index=id, columns=str.lower)
   a  b
4453153456  1  4
4453153488  2  5
4453153520  3  6

In [17]: df.reindex(index=[0, 1, 3], columns=['A', 'B', 'C'])
   A  B  C
   1.0  4.0  NaN
   2.0  5.0  NaN
   NaN  NaN  NaN

We highly encourage using named arguments to avoid confusion when using either style.

1.1.1.6 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
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
   ...: a,1
   ...: b,2
   ...: c,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.

### 1.1.1.7 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.random(n)*50+10).round(2),
                      'Quantity': np.random.randint(1, 10, size=n)})

In [29]: df.head(2)
Out[29]:
     Product  Quantity  Revenue    Store
0  Product_3       1    14.23  Store_2
1  Product_2       8    46.07  Store_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))
Out[30]:
     Product_1  Product_2  Product_3
Store
Store_1   7.26       7.00       7.22
Store_2   7.10       6.35       7.45

See the documentation for more.

1.1.1.8 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().

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.

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]
1.1.1.9 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()`, and `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)

1.1.2 Backwards incompatible API changes

1.1.2.1 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).

1.1.2.2 Sum/Prod of all-NaN Series/DataFrames is now consistently NaN

The behavior of `sum` and `prod` on all-NaN Series/DataFrames no longer depends on whether `bottleneck` is installed. (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.

```python
In [33]: s = Series([np.nan])
```
Previously NO bottleneck

```python
In [2]: s.sum()
Out[2]: np.nan
```

Previously WITH bottleneck

```python
In [2]: s.sum()
Out[2]: 0.0
```

New Behavior, without regard to the bottleneck installation.

```python
In [34]: s.sum()
Out[34]: nan
```

Note that this also changes the sum of an empty Series

Previously regardless of bottleneck

```python
In [1]: pd.Series([]).sum()
Out[1]: 0

In [35]: pd.Series([]).sum()
Out[35]: nan
```

### 1.1.2.3 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
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:
The idiomatic way to achieve selecting potentially not-found elements is via .reindex()

In [38]: s.reindex([1, 2, 3])
Out[38]:
1  2.0
2  3.0
3  NaN
dtype: float64

Selection with all keys found is unchanged.

In [39]: s.loc[[1, 2]]
Out[39]:
1  2
2  3
dtype: int64

1.1.2.4 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.

1.1.2.5 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
dtype: int64

Previously:
In [2]: type(list(s)[0])
Out[2]: numpy.int64

New Behaviour:

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 Behaviour:

In [45]: type(df.to_dict()['a'][0])
Out[45]: int

1.1.2.6 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:

In [46]: s = pd.Series([1, 2, 3], index=[False, True, False])
In [47]: s
Out[47]:
False 1
True 2
False 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

In [48]: s.loc[pd.Index([True, False, True])]
Out[48]:
False 1
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:

```
In [49]: s = pd.Series([1,2,3], index=['a', 'b', 'c'])
In [50]: s
Out[50]:
   a    1
   b    2
   c    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

```
In [51]: s.loc[pd.Index([True, False, True])]
Out[51]:
   a    1
   c    3
   dtype: int64
```

### 1.1.2.7 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:

```
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:
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, dtype: float64

In [56]: resampled.index
Out[56]:
PeriodIndex(['2017Q1', '2017Q3'], dtype='period[2Q-DEC]', freq='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:

In [1]: pi = pd.PeriodIndex(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:

In [57]: pi = pd.PeriodIndex(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
2000-01-01 02:00  NaN NaN NaN NaN
2000-01-01 03:00  NaN NaN NaN NaN
2000-01-01 04:00  NaN NaN NaN NaN
2000-01-01 05:00  NaN NaN NaN NaN
2000-01-01 06:00  NaN NaN NaN NaN
...  ...  ...  ...  ...
2000-01-10 17:00  NaN NaN NaN NaN
2000-01-10 18:00  NaN NaN NaN NaN
2000-01-10 19:00  NaN NaN NaN NaN
1.1.2.8 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).

In [61]: arr = np.array([1, 2, 3])

Previously, if you attempted the following expression, you would get a not very helpful error message:

In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)...
   ...:    ValueError: Cannot assign expression output to target

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:

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:

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:

In [4]: pd.eval("1 + 2", target=arr, inplace=True)...
   ...:   ValueError: Cannot operate inplace if there is no assignment

1.1.2.9 Dtype Conversions

Previously assignments, .where() and .fillna() with a bool assignment, would coerce to same the type (e.g. int / float), or raise for datetimelikes. These will now preserve the bools with object dtypes. (GH16821).

In [62]: s = Series([1, 2, 3])
In [5]: s[1] = True

In [6]: s
Out[6]:
   0   1
   1   1
   2   3
dtype: int64

New Behavior

In [63]: s[1] = True

In [64]: s
Out[64]:
   0   1
   1   True
   2   3
dtype: object

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 00:00:00.000000000
   1 1970-01-01 00:00:00.000000001
dtype: datetime64[ns]

These now coerce to object dtype.

In [66]: s[1] = 1

In [67]: s
Out[67]:
   0 2011-01-01 00:00:00
   1 1
dtype: object

• Inconsistent behavior in .where() with datetimelikes which would raise rather than coerce to object (GH16402)
• Bug in assignment against int64 data with np.ndarray with float64 dtype may keep int64 dtype (GH14001)

1.1.2.10 MultiIndex Constructor with a Single Level

The MultiIndex constructors no longer squeezes a MultiIndex with all length-one levels down to a regular Index. This affects all the MultiIndex constructors. (GH17178)

Previous behavior:
Length 1 levels are no longer special-cased. They behave exactly as if you had length 2+ levels, so a `MultiIndex` is always returned from all of the `MultiIndex` constructors:

```python
In [68]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[68]:
MultiIndex(levels=[['a', 'b']],
          labels=[[0, 1]])
```

1.1.2.11 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 = 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
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
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.

1.1.2.12 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:**

---

1.1. v0.21.0 (October 27, 2017)
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]')

In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
Out[3]: PeriodIndex(['2017Q1', '2017Q2', '2017Q3', '2017Q4', '2018Q1', '2018Q2'],
      dtype='period[Q-DEC]', freq='Q-DEC')

New Behavior:

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
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:

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:

In [71]: pd.interval_range(start=0, end=4)
Out[71]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4])
closed='right',
dtype='interval[int64]')

1.1.2.13 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.util.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).

• Pandas no longer registers matplotlib converters on import. The converters will be registered and used when the first plot is draw (GH17710)

1.1.3 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. Multi-index 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).

### 1.1.3.1 Series.select and DataFrame.select

The `Series.select()` and `DataFrame.select()` methods are deprecated in favor of using `df.loc[labels.map(crit)]` (GH12401)

```
In [72]: df = 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
```

### 1.1.3.2 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.

1.1.4 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).

1.1.5 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)

1.1.6 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)

1.1.7 Bug Fixes

1.1.7.1 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)

1.1.7.2 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.timedelta64` 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)

1.1.7.3 I/O

• 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_csv()` 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)
• Bug in `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)

1.1.7.4 Plotting

• Bug in plotting methods using `secondary_y` and `fontsize` not setting secondary axis font size (GH12565)
• Bug when plotting timedelta and datetime dtypes on y-axis (GH16953)
• Line plots no longer assume monotonic x data when calculating xlims, 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 `Series.plot.bar` or `DataFrame.plot.bar` with y not respecting user-passed color (GH16822)
• Bug causing `plotting.parallel_coordinates` to reset the random seed when using random colors (GH17525)

1.1.7.5 Groupby/Resample/Rolling

• Bug in `DataFrame.resample(...)`.size() where an empty DataFrame did not return a Series (GH14962)
• Bug in `infer_freq()` causing indices with 2-day gaps during the working week to be wrongly inferred as business daily (GH16624)
• Bug in `.rolling(...)`.quantile() which incorrectly used different defaults than `Series.quantile()` and `DataFrame.quantile()` (GH9413, GH16211)
• Bug in `groupby.transform()` that would coerce boolean dtypes back to float (GH16875)
• Bug in `Series.resample(...).apply()` where an empty Series modified the source index and did not return the name of a Series (GH14313)
• Bug in `.rolling(...).apply(...)` with a DataFrame with a DatetimeIndex, a window of a timedelta-convertible and `min_periods >= 1` (GH15305)
• Bug in `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 `groupby.nunique()` with TimeGrouper which cannot handle NaT correctly (GH17575)
• Bug in `DataFrame.groupby` where a single level selection from a MultiIndex unexpectedly sorts (GH17537)
• Bug in `DataFrame.groupby` where spurious warning is raised when Grouper object is used to override ambiguous column name (GH17383)
• Bug in `TimeGrouper` differs when passes as a list and as a scalar (GH17530)

1.1.7.6 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)

1.1.7.7 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 checking a tuple of strings raised a *TypeError* (GH17108)

• **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** *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).**

### 1.1.7.8 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).

### 1.1.7.9 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).**

### 1.1.7.10 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)**
pandas: powerful Python data analysis toolkit, Release 0.21.0

- Fix `DataFrame.memory_usage()` to support PyPy. Objects on PyPy do not have a fixed size, so an approximation is used instead (GH17228)

1.1.7.11 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)

1.2 v0.20.3 (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
  - I/O
  - Plotting
  - Reshaping
  - Categorical

1.2.1 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)

1.2.1.1 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).

1.2.1.2 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)
### 1.2.1.3 I/O

- 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)

### 1.2.1.4 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)

### 1.2.1.5 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)

### 1.2.1.6 Categorical

- Bug in `DataFrame.sort_values` not respecting the `kind` parameter with categorical data (GH16793)

### 1.3 v0.20.2 (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`
  - `I/O`
  - `Plotting`
1.3.1 Enhancements

- Series provides a \texttt{to\_latex} method (GH16180)
- A new groupby method \texttt{ngroup()}, parallel to the existing \texttt{cumcount()}, has been added to return the group order (GH11642); see \texttt{here}.

1.3.2 Performance Improvements

- Performance regression fix when indexing with a list-like (GH16285)
- Performance regression fix for MultiIndexes (GH16319, GH16346)
- Improved performance of \texttt{.clip()} with scalar arguments (GH15400)
- Improved performance of groupby with categorical groupers (GH16413)
- Improved performance of MultiIndex.remove_unused_levels() (GH16556)

1.3.3 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 \texttt{pathlib.Path} or \texttt{py.path.local} objects with io functions (GH16291)
- Bug in \texttt{Index.symmetric\_difference()} on two equal MultiIndex’s, results in a \texttt{TypeError} (issue 13490)
- Bug in \texttt{DataFrame.update()} with \texttt{overwrite=False} and \texttt{NaN} values (GH15593)
- Passing an invalid engine to \texttt{read\_csv()} now raises an informative \texttt{ValueError} rather than \texttt{UnboundLocalError}. (GH16511)
- Bug in \texttt{unique()} on an array of tuples (GH16519)
- Bug in \texttt{cut()} when \texttt{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)
1.3.3.1 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)

1.3.3.2 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)

1.3.3.3 I/O

- 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)

1.3.3.4 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 (:issue: 12405)
- Bug in `DataFrame.boxplot` where `figsize` keyword was not respected for non-grouped boxplots (GH11959)

1.3.3.5 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)
1.3.3.6 Sparse

- Bug in construction of SparseDataFrame from scipy.sparse.dok_matrix (GH16179)

1.3.3.7 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)

1.3.3.8 Numeric

- Bug in .interpolate(), where limit_direction was not respected when limit=None (default) was passed (GH16282)

1.3.3.9 Categorical

- Fixed comparison operations considering the order of the categories when both categoricals are unordered (GH16014)

1.3.3.10 Other

- Bug in DataFrame.drop() with an empty-list with non-unique indices (GH16270)

1.4 v0.20.1 (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 codebase. 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 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**
  - _agg API for DataFrame/Series
  - _dtype keyword for data IO
  - _to_datetime() has gained an origin parameter
  - Groupby Enhancements
  - Better support for compressed URLs in read_csv
  - Pickle file I/O now supports compression
  - UInt64 Support Improved
  - GroupBy on Categoricals
  - Table Schema Output
  - SciPy sparse matrix from/to SparseDataFrame
  - Excel output for styled DataFrames
  - IntervallIndex
  - 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
  - I/O
  - Plotting
  - Groupby/Resample/Rolling
  - Sparse
  - Reshaping
  - Numeric
  - Other
1.4.1 New features

1.4.1.1 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

```python
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]:
          A         B        C
2000-01-01  0.665719  0.234544 -0.497107
2000-01-02  0.603650  0.567011 -0.994009
2000-01-03 -2.230893 -1.635263  0.357573
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  1.667624  1.619575 -0.948507
2000-01-09 -0.360596  1.412609 -0.398833
2000-01-10 -2.429301 -0.645124 -0.102111
```

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         B        C
sum -2.083797  1.553352 -2.582995
dtype: float64
```

Multiple aggregations with a list of functions.

```python
In [5]: df.agg(['sum', 'min'])

Out[5]:
         A         B        C
sum  -2.083797  1.553352 -2.582995
min -2.429301 -1.635263 -0.994009
```

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
max  Nan  1.619575
```
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
   --- --- ---
2000-01-01 0.665719 3.095020 0.234544
2000-01-02 0.603650 3.032952 0.567011
2000-01-03 2.230893 0.198409 1.635263
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 1.667624 4.096925 1.619575
2000-01-09 0.360596 2.068705 1.412609
2000-01-10 2.429301 0.000000 0.645124
```

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]
dtype: object
```

```
In [10]: df.agg(["min", "sum"])
Out [10]:
     A  B   C      D
      min 1.0  2013-01-01
      sum 6.0  foobarbaz
```

### 1.4.1.2 dtype keyword 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\n1 2\n3 4"
In [12]: pd.read_fwf(StringIO(data)).dtypes
Out [12]:
a int64
b int64
dtype: object
```
In [13]: pd.read_fwf(StringIO(data), dtype={'a':'float64', 'b':'object'}).dtypes
\nOut[13]:
a  float64
b  object
dtype: object

1.4.1.3 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:

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.

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)

1.4.1.4 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)

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
Out[19]:
    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

1.4. v0.20.1 (May 5, 2017)
1.4.1.5 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 = 'master',  
...:     path = 'pandas/tests/io/parser/data/salaries.csv.bz2',  
...: )  
...:
In [22]: df = pd.read_table(url, compression='infer') # default, infer compression
In [23]: df = pd.read_table(url, compression='bz2')  # explicitly specify compression
In [24]: df.head(2)
Out[24]:
   S  X  E  M
0  13876 1  1  1
1  11608 1  3  0
```

1.4.1.6 Pickle file I/O 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'})
...:
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  0.804438 foo 2013-01-01 00:00:00
1 -0.000818 foo 2013-01-01 00:00:01
2 -1.084383 foo 2013-01-01 00:00:02
3  1.776343 foo 2013-01-01 00:00:03
4  0.884521 foo 2013-01-01 00:00:04

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  0.804438 foo 2013-01-01 00:00:00
1 -0.000818 foo 2013-01-01 00:00:01
2 -1.084383 foo 2013-01-01 00:00:02
3  1.776343 foo 2013-01-01 00:00:03
4  0.884521 foo 2013-01-01 00:00:04

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    1    2
0  0.804438
1 -0.000818
2 -1.084383
3  1.776343
4  0.884521
Name: A, dtype: float64

1.4.1.7 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)

In [35]: idx = pd.UInt64Index([1, 2, 3])
In [36]: df = pd.DataFrame({'A': ['a', 'b', 'c']}, index=idx)
In [37]: df.index

Out[37]: 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)

1.4.1.8 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)

```
In [38]: chromosomes = np.r_[np.arange(1, 23).astype(str), ['X', 'Y']]

In [39]: 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 [40]: df
Out[40]:
A  B  C  chromosomes
0  61 79  43   17
1  61 79  43   11
2  61 79  43   8
3  61 79  43   17
4  61 79  43   14
5  61 79  43   14
6  61 79  43   X
... ... ... ...
93 61 79  43   1
94 61 79  43   1
95 61 79  43   6
96 61 79  43   11
97 61 79  43   13
98 61 79  43   3
99 61 79  43   8

[100 rows x 4 columns]
```

**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 [41]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
Out[41]:
        A    B    C
chromosomes
2  305.0 395.0 215.0
```
The new orient 'table' for `DataFrame.to_json()` will generate a Table Schema compatible string representation of the data.

```python
In [42]: 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 [43]: df
Out[43]:
   A  B         C
idx  
0  1  a 2016-01-01
1  2  b 2016-01-02
2  3  c 2016-01-03
```

```python
In [44]: df.to_json(orient='table')

\{"schema": {"fields": [{"name": "idx", "type": "integer"}, {"name": "A", "type": "integer"},
    {"name": "B", "type": "string"}, {"name": "C", "type": "datetime"}], "primaryKey": ["idx"],
    "pandas_version": "0.20.0"}, "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`.

1.4.1.9 Table Schema Output
1.4.1.10 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 COOOrdinate format will be converted, copying data as needed.

```python
In [45]: from scipy.sparse import csr_matrix
In [46]: arr = np.random.random((1000, 5))
In [47]: arr[arr < .9] = 0
In [48]: sp_arr = csr_matrix(arr)
In [49]: sp_arr
Out[49]: <1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 521 stored elements in Compressed Sparse Row format>
In [50]: sdf = pd.SparseDataFrame(sp_arr)
In [51]: sdf
Out[51]:
          0    1    2    3    4
0       NaN  NaN  NaN  NaN  NaN
1       NaN  NaN  NaN  0.999681  NaN
2  0.989621  NaN  NaN  NaN  NaN
3       NaN  0.986495  NaN  NaN  NaN
4       NaN  NaN  NaN  NaN  0.990083
5       NaN  NaN  0.987794  NaN  NaN
6       NaN  NaN  NaN  NaN  NaN
    ...  ...  ...  ...  ...
993  0.936109  NaN  NaN  NaN  NaN
994       NaN  NaN  NaN  NaN  NaN
995       NaN  NaN  NaN  NaN  0.981932
996       NaN  0.985932  NaN  0.967058  NaN
997       NaN  NaN  NaN  NaN  NaN
998       NaN  NaN  NaN  NaN  NaN
999       NaN  NaN  NaN  NaN  NaN
[1000 rows x 5 columns]
```

To convert a SparseDataFrame back to sparse SciPy matrix in COO format, you can use:

```python
In [52]: sdf.to_coo()
Out[52]: <1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 521 stored elements in COOrdinate format>
```

1.4.1.11 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:
In [53]: np.random.seed(24)

In [54]: df = pd.DataFrame({'A': np.linspace(1, 10, 10)})

In [55]: df = pd.concat([df, pd.DataFrame(np.random.RandomState(24).randn(10, 4),
        columns=list('BCDE'))],
        axis=1)

In [56]: df.iloc[0, 2] = np.nan

In [57]: df
Out[57]:
   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

In [58]: styled = df.style.applymap(lambda val: 'color: %s' % 'red' if val < 0 else 'black').
        highlight_max()

In [59]: styled.to_excel('styled.xlsx', engine='openpyxl')

See the Style documentation for more detail.
1.4.1.12 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 [60]: c = pd.cut(range(4), bins=2)
In [61]: c
Out[61]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]

In [62]: c.categories

IntervalIndex([(-0.003, 1.5], (1.5, 3.0]
       closed='right',
       dtype='interval[float64]')
```

Furthermore, this allows one to bin other data with these same bins, with NaN representing a missing value similar to other dtypes.
```
In [63]: pd.cut([0, 3, 1, 1], bins=c.categories)
Out[63]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]
```

An IntervalIndex can also be used in Series and DataFrame as the index.
```
In [64]: df = pd.DataFrame({'A': range(4),
                  ....:       'B': pd.cut([0, 3, 1, 1], bins=c.categories)})
In [65]: df
Out[65]:
```
A
B
(-0.003, 1.5] 0
(1.5, 3.0] 1
(-0.003, 1.5] 2
(-0.003, 1.5] 3

Selecting via a specific interval:

```
In [66]: df.loc[pd.Interval(1.5, 3.0)]
Out[66]:
A
1
Name: (1.5, 3.0], dtype: int64
```

Selecting via a scalar value that is contained in the intervals.

```
In [67]: df.loc[0]
Out[67]:
A

B
(-0.003, 1.5] 0
(-0.003, 1.5] 2
(-0.003, 1.5] 3
```

1.4.1.13 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.
- 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.
- 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 multicolumn 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.io.formats.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)
• 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'` or `'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'` or `'mid'` or `'zero', the default is "left", which is backward compatible; You can now pass a list of `color=[color_negative, color_positive]`. (GH14757)

### 1.4.2 Backwards incompatible API changes

#### 1.4.2.1 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)
```
1.4.2.2 Map on Index types now return other Index types

map on an Index now returns an Index, not a numpy array (GH12766)

```python
In [68]: idx = Index([1, 2])

In [69]: idx
Out[69]: Int64Index([1, 2], dtype='int64')

In [70]: mi = MultiIndex.from_tuples([(1, 2), (2, 4)])

In [71]: mi
Out[71]: MultiIndex(levels=[[1, 2], [2, 4]], labels=[[0, 1], [0, 1]])
```

Previous Behavior:

```python
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:

```python
In [72]: idx.map(lambda x: x * 2)
Out[72]: Int64Index([2, 4], dtype='int64')

In [73]: idx.map(lambda x: (x, x * 2))
Out[73]: MultiIndex(levels=[[1, 2], [2, 4]], labels=[[0, 1], [0, 1]])

In [74]: mi.map(lambda x: x)
Out[74]: MultiIndex(levels=[[1, 2], [2, 4]], labels=[[0, 1], [0, 1]])
```
map on a Series with datetime64 values may return int64 dtypes rather than int32

```
In [76]: s = Series(date_range('2011-01-02T00:00', '2011-01-02T02:00', freq='H').tz_localize('Asia/Tokyo'))
```

```
In [77]: s
Out[77]:
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
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 [78]: s.map(lambda x: x.hour)
Out[78]:
0  0
1  1
2  2
dtype: int64
```

### 1.4.2.3 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 [79]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')
```

```
In [80]: idx.hour
Out[80]: 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).

### 1.4.2.4 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.Index([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 [81]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                     pd.Timestamp('20160101', tz='US/Eastern')]).unique()
  Out[81]: array([Timestamp('2016-01-01 00:00:00-05:00', tz='US/Eastern')], dtype=object)

  In [82]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                            pd.Timestamp('20160101', tz='US/Eastern')]))
  Out[82]: array([Timestamp('2016-01-01 00:00:00-05:00', tz='US/Eastern')], dtype=object)

  # Index, returns a DatetimeIndex
  In [83]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                 pd.Timestamp('20160101', tz='US/Eastern')]).unique()
  Out[83]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]', freq=None)

  In [84]: pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                            pd.Timestamp('20160101', tz='US/Eastern')]))
  ```
• 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 [85]: pd.Series(list('baabc'), dtype='category').unique()
  Out[85]:
  [b, a, c]
  Categories (3, object): [b, a, c]

  In [86]: pd.unique(pd.Series(list('baabc'), dtype='category'))
  Out[86]:
  [b, a, c]
  Categories (3, object): [b, a, c]

1.4.2.5 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).

1.4.2.6 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 [87]: df = DataFrame({'a': [1, 2, 3]}, 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

  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
```

1.4.2.7 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 [88]: df1 = pd.DataFrame(np.array([1.0], dtype=np.float32, ndmin=2))
In [89]: df1.dtypes
Out[89]:
0 float32
dtype: object
In [90]: df2 = pd.DataFrame(np.array([np.nan], dtype=np.float32, ndmin=2))
In [91]: df2.dtypes
Out[91]:
0 float32
dtype: object
```

Previous Behavior:

```
In [7]: pd.concat([df1, df2]).dtypes
Out[7]:
0 float64
dtype: object
```

New Behavior:

```
In [92]: pd.concat([df1, df2]).dtypes
Out[92]:
0 float32
dtype: object
```

1.4.2.8 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)

1.4.2.9 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 = 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 = 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]: 260

1.4.2.10 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 [93]: df = DataFrame(np.arange(6), columns=['value'], index=MultiIndex.from_product([list('BA'), range(3)]))
In [94]: df
Out[94]:
     value
    B  0  1  2
    A  3  4  5
In [95]: df.index.is_lexsorted()
Out[95]: False
In [96]: df.index.is_monotonic
Out[96]: False

Sorting works as expected

In [97]: df.sort_index()
Out[97]:
In [98]: df.sort_index().index.is_lexsorted()
Out[98]: True
In [99]: df.sort_index().index.is_monotonic
Out[99]: True

However, this example, which has a non-monotonic 2nd level, doesn’t behave as desired.

In [100]: df = pd.DataFrame(
   ....:     {'value': [1, 2, 3, 4]},
   ....:     index=pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
   ....:                          labels=[[0, 0, 1, 1], [0, 1, 0, 1]]))

In [101]: df
Out[101]:
   value
  a  bb  1
      aa  2
  b  bb  3
      aa  4

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 [102]: df.sort_index()
Out[102]:
   value
  a  aa  2
    bb  1
  b  aa  4
    bb  3

In [103]: df.sort_index().index.is_lexsorted()
Out[103]: True
1.4.2.11 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:

```python
In [1]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
In [2]: df.groupby('A').describe()
Out[2]:
         B
      count 2.000000
     mean  1.500000
     std  0.707107
    min  1.000000
   25%  1.250000
   50%  1.500000
   75%  1.750000
   max  2.000000

In [3]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
Out[3]:
      B
    mean  std  amin  amax
   A    1  1.5  0.707107    1   2
      2  3.5  0.707107    3   4
```

New Behavior:

```python
In [105]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
In [106]: df.groupby('A').describe()
```

```plaintext
     count  mean  std  min  25%  50%  75%  max
   A
   1  2.0  1.5  0.707107  1.0  1.25  1.5  1.75  2.0
   2  2.0  3.5  0.707107  3.0  3.25  3.5  3.75  4.0
```

```python
In [107]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
Out[107]:
```
### 1.4.2.12 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)

```python
In [108]: np.random.seed(1234)
In [109]: df = pd.DataFrame(np.random.rand(100, 2),
                   columns=pd.Index(['A', 'B'], name='bar'),
                   index=pd.date_range('20160101',
                       periods=100, freq='D', name='foo'))
In [110]: df.tail()
Out[110]:
   bar    A     B
foo    
2016-04-05 0.640880 0.126205
2016-04-06 0.171465 0.737086
2016-04-07 0.127029 0.369650
2016-04-08 0.604334 0.103104
2016-04-09 0.802374 0.945553
```

Previous Behavior:

```python
In [2]: df.rolling(12).corr()
Out[2]:
<class 'pandas.core.panel.Panel'>
Dimensions: 100 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: 2016-01-01 00:00:00 to 2016-04-09 00:00:00
Major_axis axis: A to B
Minor_axis axis: A to B
```

New Behavior:

```python
In [111]: res = df.rolling(12).corr()
In [112]: res.tail()
Out[112]:
   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
```

Retrieving a correlation matrix for a cross-section
In [113]: df.rolling(12).corr().loc['2016-04-07']
Out[113]:
   A      B
foo bar  
2016-04-07 1.00000 -0.13209
       B  1.00000

1.4.2.13 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 [114]: df = pd.DataFrame({'unparsed_date': ['2014-01-01', '2014-01-01']})
In [115]: df.to_hdf('store.h5', 'key', format='table', data_columns=True)
In [116]: df.dtypes
Out[116]:
unparsed_date object
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 00:00:01.388552400)
^   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')
TypeError: Cannot compare 2014-01-01 00:00:00 of
type <class 'pandas.tslib.Timestamp'> to string column

1.4.2.14 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

  In [117]: left = pd.Index([2, 1, 0])
  In [118]: left
  Out[118]: Int64Index([2, 1, 0], dtype='int64')
  In [119]: right = pd.Index([1, 2, 3])
  In [120]: right
  Out[120]: Int64Index([1, 2, 3], dtype='int64')

Previous Behavior:
In [4]: left.intersection(right)
Out[4]: Int64Index([1, 2], dtype='int64')

New Behavior:

In [121]: left.intersection(right)
Out[121]: Int64Index([2, 1], dtype='int64')

• DataFrame.join and pd.merge

In [122]: left = pd.DataFrame({'a': [20, 10, 0]}, index=[2, 1, 0])
In [123]: left
Out[123]:
   a
2 20
1 10
0 0

In [124]: right = pd.DataFrame({'b': [100, 200, 300]}, index=[1, 2, 3])
In [125]: right
Out[125]:
   b
1 100
2 200
3 300

Previous Behavior:

In [4]: left.join(right, how='inner')
Out[4]:
   a  b
1 10 100
2 20 200

New Behavior:

In [126]: left.join(right, how='inner')
Out[126]:
   a  b
2 20 200
1 10 100

1.4.2.15 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 circumstance. (GH4386)

In [127]: df = DataFrame({'col1': [3, 4, 5],
                    'col2': ['C', 'D', 'E'],
                    'col3': [1, 3, 9]})
In [128]: df
Out[128]:

### 1.4.2.16 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).
- `CParseError` 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 DataFrame 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 `is_month_start` (GH15781)
- `NaT` will now correctly return `np.nan` for `Timedelta` and `Period` accessors such as `days` and `quarter` (GH15782)
- `NaT` will now returns `NaT` for `tz_localize` and `tz_convert` methods (GH15830)
- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `PandasError`, if called with scalar inputs and not axes (GH15541)
- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `pandas.core.common.PandasError`, if called with scalar inputs and not axes; The exception `PandasError` is removed as well. (GH15541)
- The exception `pandas.core.common.AmbiguousIndexError` is removed as it is not referenced (GH15541)

1.4.3 Reorganization of the library: Privacy Changes

1.4.3.1 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 `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are now considered to be PRIVATE. If indicated, a deprecation warning will be issued if you reference theses modules. (GH12588)
Some new subpackages are created with public functionality that is not directly exposed in the top-level namespace: pandas.errors, pandas.plotting and pandas.testing (more details below). Together with pandas.api.types and certain functions in the pandas.io and 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)

1.4.3.2 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',
]```
1.4.3.3 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()

1.4.3.4 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.

1.4.3.5 Other Development Changes

- Building pandas for development now requires cython >= 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).

1.4.4 Deprecations

1.4.4.1 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. To wit, .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.

```python
In [130]: df = pd.DataFrame({'A': [1, 2, 3],
......:            'B': [4, 5, 6]},
......:     index=list('abc'))

In [131]: df
Out[131]:
   A  B
0  1  4
1  2  5
2  3  6
```
Previous Behavior, where you wish to get the 0th and the 2nd elements from the index in the ‘A’ column.

```python
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.

```python
In [132]: df.loc[df.index[[0, 2]], 'A']
Out[132]:
a 1
Name: A, dtype: int64
```

Using `.iloc`. Here we will get the location of the ‘A’ column, then use positional indexing to select things.

```python
In [133]: df.iloc[[0, 2], df.columns.get_loc('A')]
Out[133]:
a 1
Name: A, dtype: int64
```

### 1.4.4.2 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. For more details see Deprecate Panel documentation. (GH13563).

```python
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

```python
In [136]: p.to_frame()
Out[136]:
ItemA  ItemB  ItemC
major minor
2000-01-03 A  0.628776 -1.409432  0.209395
     B    0.988138 -1.347533 -0.896581
     C    -0.938153  1.272395 -0.161137
     D    -0.223019 -0.591863 -1.051539
2000-01-04 A   0.186494  1.422986 -0.592886
```
Convert to an xarray DataArray

```python
In [137]: p.to_xarray()
Out[137]:
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[-0.072608, 0.363565, 1.104352],
       [-1.239072, -1.449567, 0.889157],
       [2.123692, -0.414505, -0.319561]],
       [[0.952478, -2.147855, -1.473116],
        [-0.550603, -0.014752, -0.431550],
        [0.139683, -1.195524, 0.288377],
        [0.122273, -1.425795, -0.619993]],
       [[-1.409432, -1.347533, 1.272395, -0.591863],
        [1.422986, 0.363565, -1.449567, -0.414505],
        [-2.147855, -0.014752, -1.195524, -1.425795]],
       [[-0.592886, 1.104352, 0.889157, -0.319561],
        [-1.473116, -0.43155 , 0.288377, -0.619993]],
       [[ 0.209395, -0.896581, -0.161137, -1.051539],
        [-0.592886, 1.104352, 0.889157, -0.319561],
        [-1.473116, -0.43155 , 0.288377, -0.619993]])
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'
```

1.4.4.3 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 [138]: df = pd.DataFrame({
                        'A': [1, 1, 1, 2, 2],
                        ....:
                        'B': range(5),
                        ....:
                        'C': range(5)}
                        ....:

In [139]: df
Out[139]:
     A  B  C
0  0  0  0
1  1  1  1
```
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).

```python
In [140]: df.groupby('A').agg({'B': 'sum', 'C': 'min'})
Out[140]:
   B  C
A  
  1  3  0
  2  7  3
```

Here’s an example of the first deprecation, passing a dict to a grouped Series. This is a combination aggregation & renaming:

```python
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]:
   foo
A  
  1  3
  2  2
```

You can accomplish the same operation, more idiomatically by:

```python
In [141]: df.groupby('A').B.agg({'count': 'sum'}).rename(columns={'count': 'foo'})
Out[141]:
   foo
A  
  1  3
  2  2
```

Here’s an example of the second deprecation, passing a dict-of-dict to a grouped DataFrame:

```python
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]:
   B  C
foo bar
A  
  1  3  0
  2  7  3
```

You can accomplish nearly the same by:

```python
In [142]: (df.groupby('A')
    ....:   .agg({'B': 'sum', 'C': 'min'})
    ....:   .rename(columns={'B': 'foo', 'C': 'bar'}))
```
1.4.4.4 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)
```

1.4.4.5 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)

1.4.5 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 func-
  tionality 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.items() (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 respec-
  tively (GH10735)

1.4.6 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)

1.4.7 Bug Fixes

1.4.7.1 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)
- Incorrect dtype for `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 a ndarray and not a `Categorical` (GH15903)
- **Bug in** `Index.to_series()` where the index was not copied (and so mutating later would change the original), (GH15949)
- **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)

### 1.4.7.2 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 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)

1.4.7.3 I/O

• 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 multiline headers are specified (GH15376)

• Bug in `pd.read_csv()` with `float_precision='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).

1.4.7.4 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)

1.4.7.5 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)

1.4. v0.20.1 (May 5, 2017)
• 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)

1.4.7.6 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)

1.4.7.7 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 dtype (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)
• 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)

1.4.7.8 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 multiline evals to fail with local variables not on the first line (GH15342)

1.4.7.9 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)

1.5 v0.19.2 (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

1.5.1 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)

### 1.5.2 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)

### 1.5.3 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)
- 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 datetime64 values 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)
- 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 (GH7626)
• 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)

1.6 v0.19.1 (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

1.6.1 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).
1.6.2 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 objectIndex 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 dttype 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 (:issue'14327').
- 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).
1.7 v0.19.0 (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**
  - `merge_asof` for asof-style time-series joining
  - `.rolling()` is now time-series aware
  - `read_csv` has improved support for duplicate column names
  - `read_csv` supports parsing Categorical directly
  - `Categorical Concatenation`
  - `Semi-Month Offsets`
  - `New Index methods`
  - `Google BigQuery Enhancements`
  - `Fine-grained numpy errstate`
  - `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
  - `.to_datetime()` changes
  - Merging changes
  - `.describe()` changes
  - `Period` changes
    * `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
  - `read_csv` will progressively enumerate chunks
  - **Sparse Changes**
    * `int64` and `bool` support enhancements
    * Operators now preserve dtypes
    * Other sparse fixes
  - `Indexer` dtype changes
  - Other API Changes

• **Deprecations**

• **Removal of prior version deprecations/changes**

• **Performance Improvements**

• **Bug Fixes**

### 1.7.1 New features

#### 1.7.1.1 `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.

```python
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  1       a
1  5       b
2 10      c

In [4]: right
   a  right_val
0  1       1
1  2       2
2  3       3
3  6       6
4  7       7

We typically want to match exactly when possible, and use the most recent value otherwise.

```python
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

We can also match rows ONLY with prior data, and not an exact match.

```python
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

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'],
                         'ticker': ['MSFT', 'MSFT',
                                  'GOOG', 'GOOG', 'AAPL'],
                         'price': [51.95, 51.95,
                                   720.77, 720.92, 98.00],
                         ...
                      })
```
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
1 2016-05-25 13:30:00.038  MSFT  51.95     155
2 2016-05-25 13:30:00.048  GOOG  720.77     100
3 2016-05-25 13:30:00.048  GOOG  720.92     100
4 2016-05-25 13:30:00.048  AAPL  98.00     100

In [10]: quotes
Out[10]:
    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

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.

In [11]: pd.merge_asof(trades, quotes,
    ...:     on='time',
    ...:     by='ticker')
Out[11]:
    time    ticker  price  quantity   bid   ask
0 2016-05-25 13:30:00.023  MSFT  51.95      75   51.95  51.96
This returns a merged DataFrame with the entries in the same order as the original left passed DataFrame (trades in this case), with the fields of the quotes merged.

**1.7.1.2 .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](#).

```python
In [12]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
          index=pd.date_range('20130101 09:00:00', periods=5, freq='s'))

In [13]: dft
Out[13]:
   B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```python
In [14]: dft.rolling(2).sum()
Out[14]:
   B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN
```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```python
In [16]: dft.rolling('2s').sum()
```

```python
   B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
```
Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```python
In [17]: dft = DataFrame({'B': [0, 1, 2, np.nan, 4]},
                      index = pd.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')],
                                        name='foo'))

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

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
```

Using the time-specification generates variable windows for this sparse data.

```python
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  NaN
```

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a DataFrame.

```python
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
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

```
3 2013-01-01 09:00:05 NaN
4 2013-01-01 09:00:06 4.0

In [23]: dft.rolling('2s', on='foo').sum()

Out[23]:
       foo
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
```

1.7.1.3 `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)
Out[26]:
   a  b  a.1
0  0  1  2
1  3  4  5
```

1.7.1.4 `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.

```
In [27]: data = 'col1,col2,col3
a,b,1
a,b,2
na,b,2
na,b,2
nc,d,3'

In [28]: pd.read_csv(StringIO(data))
Out[28]:
   col1  col2  col3
0    a     b     1
1    a     b     2
2    c     d     3
```

1.7. v0.19.0 (October 2, 2016)
Individual columns can be parsed as a Categorical using a dict specification

```
In [31]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[31]:
        col1    col2    col3
dtype: category  object  int64
```

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()`.

```
In [32]: df = pd.read_csv(StringIO(data), dtype='category')

In [33]: df.dtypes
Out[33]:
        col1    col2    col3
dtype: category  object  int64

In [34]: df['col3']
Out[34]:
0  1
1  2
2  3
Name: col3, 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, dtype: category
Categories (3, int64): [1, 2, 3]
```
1.7.1.5 Categorical Concatenation

- A function `union_categoricals()` has been added for combining categoricals, see Unioning Categoricals (GH13361, GH:13763, issue:13846, GH14173)

```python
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)

```python
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:
In [43]: pd.concat([s1, s2])
Out[43]:
0  a
1  b
0  b
1  c
dtype: object
```

1.7.1.6 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:
In [45]: 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:
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

Using the anchoring suffix, you can also specify the day of month to use instead of the 15th.

```
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-16')
```

```
In [50]: pd.date_range('2015-01-01', freq='SM-14', periods=4)
```

1.7.1.7 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([['a', 'b', 'c'], ['1', '2', np.nan]])
In [56]: midx
Out[56]: MultiIndex(levels=[[1, 2, 4], [1, 2]],
                labels=[[0, 1, -1, 2], [0, 1, -1, -1]])
In [57]: midx.dropna()
```

```
In [58]: midx.dropna(how='all')
```
Index now supports \texttt{.str.extractall()} which returns a DataFrame, see the docs here\cite{GH10008, GH13156}

\begin{verbatim}
In [59]: idx = pd.Index(['a1a2', 'b1', 'c1'])
In [60]: idx.str.extractall('^[ab]?P<digit>\d$')
Out[60]:
   digit
0   1
1   2
1   1

Index\texttt{.astype()} now accepts an optional boolean argument \texttt{copy}, which allows optional copying if the requirements on dtype are satisfied\cite{GH13209}

1.7.1.8 Google BigQuery Enhancements

- The \texttt{read_gbq()} method has gained the \texttt{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\cite{GH13615}.
- The \texttt{to_gbq()} method now allows the DataFrame column order to differ from the destination table schema\cite{GH11359}.

1.7.1.9 Fine-grained numpy errstate

Previous versions of pandas would permanently silence numpy's ufunc error handling when \texttt{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 \texttt{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 codebase.\cite{GH13109, GH13145}

After upgrading pandas, you may see new \texttt{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 \texttt{numpy.errstate} around the source of the \texttt{RuntimeWarning} to control how these conditions are handled.

1.7.1.10 get\_dummies now returns integer dtypes

The \texttt{pd.get_dummies} function now returns dummy-encoded columns as small integers, rather than floats\cite{GH8725}. This should provide an improved memory footprint.

Previous behavior:

\begin{verbatim}
In [1]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[1]:
a    float64
b    float64
\end{verbatim}
New behavior:

```python
In [61]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[61]:
          a       b       c
dtype: uint8 uint8 uint8
          dtype: object
```

1.7.1.11 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)
```

1.7.1.12 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)
```

```python
['CategoricalDtype', 'DatetimeTZDtype', 'IntervalDtype', 'PeriodDtype', 'infer_dtype', 'is_any_int_dtype', 'is_bool', 'is_bool_dtype', 'is_categorical', 'is_categorical_dtype', 'is_complex', 'is_complex_dtype', 'is_datetime64_any_dtype', 'is_datetime64_dtype',
```

88 Chapter 1. What’s New
'is_datetime64_ns_dtype',
'is_datetime64tz_dtype',
'is_datetime64tz',
'is_dict_like',
'is_dtype_equal',
'is_extension_type',
'is_file_like',
'is_float',
'is_float_dtype',
'is_floating_dtype',
'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',
'is_period_dtype',
'is_re',
'is_re_compilable',
'is_scalar',
'is_sequence',
'is_signed_integer_dtype',
'is_sparse',
'is_string_dtype',
'is_timedelta64_dtype',
'is_timedelta64_ns_dtype',
'is_unsigned_integer_dtype',
'pandas_dtype',
'union_categoricals']

Note: Calling these functions from the internal module pandas.core.common will now show a DeprecationWarning (GH13990)

1.7.1.13 Other enhancements

- Timestamp can now accept positional and keyword parameters similar to datetime.datetime() (GH10758, GH11630)

```python
In [69]: pd.Timestamp(2012, 1, 1)
Out[69]: Timestamp('2012-01-01 00:00:00')
```

```python
In [70]: pd.Timestamp(year=2012, month=1, day=1, hour=8, minute=30)
```

```python
\\\\\\\\\\\\\\\\\\\\\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
Out[72]:
    a     date
  0  0  2015-01-04
  1  1  2015-01-11
  2  2  2015-01-18
  3  3  2015-01-25
  4  4  2015-02-01

In [73]: df.resample('M', on='date').sum()
   a     date
  0  6  2015-01-31
  1  4  2015-02-28

In [74]: df.resample('M', level='d').sum()
   a     d
  0  6  2015-01-31
  1  4  2015-02-28

- 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 (GH13358)
• 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.sort_values(by='row2', axis=1)
```

```
     A  B  C
row1 3 2 4
row2 5 7 8
```

• 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)
• 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).

### 1.7.2 API changes

#### 1.7.2.1 Series.tolist() will now return Python types

`Series.tolist()` will now return Python types in the output, mimicking NumPy .tolist() behavior (GH10904)

```python
In [78]: s = pd.Series([1,2,3])

Previous behavior:

```text
In [7]: type(s.tolist()[0])
Out [7]:
<class 'numpy.int64'>
```

New behavior:

```text
In [79]: type(s.tolist()[0])
Out [79]: int
```

#### 1.7.2.2 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 raise `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).

```python
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
dtype: float64
```

```python
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  3.0
A    3.0
B    4.0
C   NaN
D   NaN
```

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:

```python
In [1]: s1 == s2
Out[1]:
A  False
B   True
C  False
dtype: bool
```

**New behavior (Series):**

```python
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`.

```python
In [86]: s1.values == s2.values
Out[86]: array([False,   True, False], dtype=bool)
```

If you want to compare Series aligning its `.index`, see flexible comparison methods section below:
In [87]: s1.eq(s2)
Out[87]:
A  False
B  True
C  False
D  False
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
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:

In [91]: s1 & s2.reindex_like(s1)
Out[91]:
A  True
B  False
Current Behavior (DataFrame, no change):

```
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  NaN
D  NaN
```

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`.

```
In [95]: s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
In [96]: s2 = pd.Series([2, 2, 2], index=['b', 'c', 'd'])
In [97]: s1.eq(s2)
Out[97]:
  a  False
  b  True
  c  False
  d  False
dtype: bool

In [98]: s1.ge(s2)
Out[98]:
  a  False
  b  True
  c  True
  d  False
dtype: bool
```

Previously, this worked the same as comparison operators (see above).

1.7.2.3 Series type promotion on assignment

A `Series` will now correctly promote its dtype for assignment with incompat values to the current dtype (GH13234)

```
In [99]: s = pd.Series()

Previous behavior:

In [2]: s['a'] = pd.Timestamp("2016-01-01")
```
In [3]: s["b"] = 3.0
TypeError: invalid type promotion

New behavior:

In [100]: s["a"] = pd.Timestamp("2016-01-01")
In [101]: s["b"] = 3.0
In [102]: s
Out[102]:
a  2016-01-01 00:00:00
b  3
dtype: object
In [103]: s.dtype
Out[103]: dtype('O')

1.7.2.4 .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:

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.

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 .to_datetime():

• Bug in pd.to_datetime() when passing integers or floats, and no unit and errors='coerce' (GH13180).
• Bug in pd.to_datetime() when passing invalid datatypes (e.g. bool); will now respect the errors keyword (GH13176).
• Bug in pd.to_datetime() which overflowed on int8, and int16 dtypes (GH13451).
• Bug in pd.to_datetime() raise AttributeError with NaN and the other string is not valid when errors='ignore' (GH12424).
• Bug in pd.to_datetime() did not cast floats correctly when unit was specified, resulting in truncated datetime (GH13834).

1.7.2.5 Merging changes

Merging will now preserve the dtype of the join keys (GH8596).
In [105]: df1 = pd.DataFrame({'key': [1], 'v1': [10]})

In [106]: df1
Out[106]:
   key  v1
0    1  10

In [107]: df2 = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [108]: df2
Out[108]:
   key  v1
0    1  20
1    2  30

Previous behavior:

In [5]: pd.merge(df1, df2, how='outer')
Out[5]:
   key  v1
0   1   10
1   1  20.0
2   2  30.0

In [6]: pd.merge(df1, df2, how='outer').dtypes
Out[6]:
   key  float64
   v1   float64
dtype: object

New behavior:

We are able to preserve the join keys

In [109]: pd.merge(df1, df2, how='outer')
Out[109]:
   key  v1
0    1  10
1    1  20
2    2  30

In [110]: pd.merge(df1, df2, how='outer').dtypes
Out[110]:
   key  int64
   v1   int64
dtype: object

Of course if you have missing values that are introduced, then the resulting dtype will be upcast, which is unchanged from previous.

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

In [112]: pd.merge(df1, df2, how='outer', on='key').dtypes
Out[112]:
   key  int64
   v1_x float64
   v1_y float64
dtype: object
1.7.2.6 .describe() changes

Percentile identifiers in the index of a .describe() output will now be rounded to the least precision that keeps them distinct (GH13104)

In [113]: s = pd.Series([0, 1, 2, 3, 4])
In [114]: 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.

In [3]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[3]:
count    5.000000
mean      2.000000
std       1.581139
min       0.000000
0.0%      0.000000
0.1%      0.002000
0.1%      0.004000
50%       2.000000
99.9%      3.996000
100.0%     3.998000
100.0%     3.999600
max       4.000000
dtype: float64

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       0.000000
0.01%     0.000000
0.05%     0.000200
0.1%      0.000400
50%       2.000000
99.9%      3.996000
99.95%     3.998000
99.99%     3.999600
max       4.000000
dtype: float64
In [116]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])

   count       5.000000
   mean        2.000000
   std         1.581139
   0.01%       0.000400
   0.05%       0.002000
   0.1%        0.004000
   50%         2.000000
   99.9%       3.996000
   99.95%      3.998000
   99.99%      3.999600
   max         4.000000

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)

1.7.2.7 Period changes

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:

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:

In [117]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')
In [118]: pi
Out[118]: PeriodIndex(['2016-08-01'], dtype='period[D]', freq='D')
In [119]: pd.api.types.is_integer_dtype(pi)
Out[119]: False
In [120]: pd.api.types.is_period_dtype(pi)
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:
In [6]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')
In [7]: pi.values
   array([492, 493])

New behavior:

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)

1.7.2.8 Index

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:

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:

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.

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:

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:

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)

1.7.2.9 Index.difference and .symmetric_difference changes

Index.difference and Index.symmetric_difference will now, more consistently, treat NaN values as any other values. (GH13514)
In [132]: idx1 = pd.Index([1, 2, 3, np.nan])
In [133]: idx2 = pd.Index([0, 1, np.nan])

Previous behavior:

In [3]: idx1.difference(idx2)
Out[3]: Float64Index([nan, 2.0, 3.0], dtype='float64')
In [4]: idx1.symmetric_difference(idx2)
Out[4]: Float64Index([0.0, nan, 2.0, 3.0], dtype='float64')

New behavior:

In [134]: idx1.difference(idx2)
Out[134]: Float64Index([2.0, 3.0], dtype='float64')
In [135]: idx1.symmetric_difference(idx2)
Out[135]: Float64Index([0.0, 2.0, 3.0], dtype='float64')

1.7.2.10 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:

In [1]: pd.Index([1, 2, 3]).unique()
Out[1]: array([1, 2, 3])
In [2]: pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo').unique()
Out[2]: 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)

New behavior:

In [136]: pd.Index([1, 2, 3]).unique()
Out[136]: Int64Index([1, 2, 3], dtype='int64')
In [137]: pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo').unique()
Out[137]: 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)

1.7.2.11 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(levels=[['b', 'a', 'c'], ['bar', 'foo']], labels=[[1, 0], [1, 0]])

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
C    object
B    float64
dtype: object

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:

```python
In [147]: df_grouped.index.levels[1]
Out[147]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False, name='C', dtype='category')

In [148]: df_grouped.reset_index().dtypes
   →
A    int64
C    category
B    float64
dtype: object

In [149]: df_set_idx.index.levels[1]
   →
CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False, name='C', dtype='category')

In [150]: df_set_idx.reset_index().dtypes
   →
A    int64
C    category
B    int64
dtype: object
```

### 1.7.2.12 *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).

```python
In [151]: data = 'A,B
0,1
2,3
4,5
6,7'

In [2]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[2]:
   A  B
0  0  1
1  2  3
0  4  5
1  6  7
```

**Previous behavior:**

```python
In [2]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[2]:
   A  B
0  0  1
1  2  3
0  4  5
1  6  7
```

**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
```
1.7.2.13 Sparse Changes

These changes allow pandas to handle sparse data with more dtypes, and for work to make a smoother experience with data handling.

**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)

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](https://pandas.pydata.org/docs) for more details.
Operators now preserve dtypes

- Sparse data structure now can preserve dtype after arithmetic ops (GH13848)

```python
In [155]: s = pd.SparseSeries([0, 2, 0, 1], fill_value=0, dtype=np.int64)

In [156]: s.dtype
Out[156]:

In [157]: s + 1
Out[157]:
```

- Sparse data structure now support astype to convert internal dtype (GH13900)

```python
In [158]: s = pd.SparseSeries([1., 0., 2., 0.], fill_value=0)

In [159]: s
Out[159]:

In [160]: 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], fill_value=np.nan).astype(np.int64)
Out[7]:
```

---

106 Chapter 1. What’s New
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)
- Bug in `SparseDataFrame` doesn’t respect passed `SparseArray` or `SparseSeries` ‘s dtype and fill_value (GH13866)
- Bug in `SparseArray` and `SparseSeries` don’t apply ufunc to fill_value (GH13853)
- Bug in `SparseSeries`.abs incorrectly keeps negative fill_value (GH13853)
- Bug in single row slicing on multi-type `SparseDataFrame` s, types were previously forced to float (GH13917)
- Bug in `SparseSeries` slicing changes integer dtype to float (GH8292)
- Bug in `SparseDataFrame` comparison ops may raise TypeError (GH13001)
- Bug in `SparseDataFrame`.isnull raises ValueError (GH8276)
- Bug in `SparseSeries` representation with bool dtype may raise IndexError (GH13110)
- Bug in `SparseSeries` and `SparseDataFrame` of bool or int64 dtype may display its values like float64 dtype (GH13110)
- Bug in sparse indexing using `SparseArray` with bool dtype may return incorrect result (GH13985)
- Bug in `SparseArray` created from `SparseSeries` may lose dtype (GH13999)
- Bug in `SparseSeries` comparison with dense returns normal Series rather than `SparseSeries` (GH13999)

1.7.2.14 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:**
New behavior:

```python
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int32')
```

```python
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int64')
```

1.7.2.15 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 keyword 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 `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 `CParserError` (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)

### 1.7.3 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-likes** 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)

### 1.7.4 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).

### 1.7.5 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 DataFrameGroupBy.transform(GH12737)
• Improved performance of Index.difference(GH12044)
• Improved performance of RangeIndex.is_monotonic_increasing and is_monotonic_decreasing(GH13749)
• 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)

1.7.6 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) (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 datetme 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 (GH13306)
• 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 coerce `datetime64` and `timedelta` to `object` dtype containing python built-in `datetime` or `timedelta` rather than `Timestamp` or `Timedelta` (GH13626)
- Bug in `PeriodIndex.append` may raise `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 (GH12546)
- 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 datetime-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 dtyped 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 lex-sorted 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 formats key word (GH10690)

• Bug in `DataFrame.iterrows()`, not yielding a `Series` subclasse 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 `datetime` 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 Unbounded-LocalError 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)
1.8 v0.18.1 (May 3, 2016)

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
  - `.groupby(...) syntax with window and resample operations
  - Method chainning improvements
    - `.where()` and `.mask()
    - `.loc[],.iloc[],.ix[]`
    - [] indexing
  - Partial string indexing on `DateTimeIndex when part of a MultiIndex`
  - Assembling Datetimes
  - Other Enhancements

- **Sparse changes**

- **API changes**
  - `.groupby(...) .nth()` changes
  - numpy function compatibility
  - Using `.apply` on `groupby` resampling
  - Changes in `read_csv` exceptions
  - `to_datetime` error changes
  - Other API changes
  - Deprecations

- **Performance Improvements**

- **Bug Fixes**
1.8.1 New features

1.8.1.1 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)

```
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

```
In [4]: dt = datetime(2014, 1, 17, 15)
In [5]: dt + bhour_us
Out[5]: Timestamp('2014-01-17 16:00:00')
```

Tuesday after MLK Day (Monday is skipped because it’s a holiday)

```
In [6]: dt + bhour_us * 2
Out[6]: Timestamp('2014-01-20 09:00:00')
```

1.8.1.2 `.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:

```
...:                      'B': np.arange(40))}
...:
In [8]: df
Out[8]:
   A  B
0  0  0
1  1  1
2  2  2
3  3  3
4  4  4
5  5  5
6  6  6
.. .. ..
33 33
34 34
35 35
36 36
37 37
38 38
```

In [9]: df.groupby('A').apply(lambda x: x.rolling(4).B.mean())

Out[9]:
   A
0  0  NaN
  1  NaN
  2  NaN
  3  1.5
  4  2.5
  5  3.5
  6  4.5
...
3  33  NaN
34  NaN
35  33.5
36  34.5
37  35.5
38  36.5
39  37.5

Name: B, Length: 40, dtype: float64

Now you can do:

In [10]: df.groupby('A').rolling(4).B.mean()

Out[10]:
   A
0  0  NaN
  1  NaN
  2  NaN
  3  1.5
  4  2.5
  5  3.5
  6  4.5
...
3  33  NaN
34  NaN
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 [11]: 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 [12]: df
Out[12]:
   group  val

1.8. v0.18.1 (May 3, 2016)
In [13]: df.groupby('group').apply(lambda x: x.resample('1D').ffill())

Out[13]:

<table>
<thead>
<tr>
<th>group</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2016-01-03</td>
</tr>
<tr>
<td></td>
<td>2016-01-04</td>
</tr>
<tr>
<td></td>
<td>2016-01-05</td>
</tr>
<tr>
<td></td>
<td>2016-01-06</td>
</tr>
<tr>
<td></td>
<td>2016-01-07</td>
</tr>
<tr>
<td></td>
<td>2016-01-08</td>
</tr>
<tr>
<td></td>
<td>2016-01-09</td>
</tr>
<tr>
<td>...</td>
<td>... ...</td>
</tr>
<tr>
<td>2</td>
<td>2016-01-18</td>
</tr>
<tr>
<td></td>
<td>2016-01-19</td>
</tr>
<tr>
<td></td>
<td>2016-01-20</td>
</tr>
<tr>
<td></td>
<td>2016-01-21</td>
</tr>
<tr>
<td></td>
<td>2016-01-22</td>
</tr>
<tr>
<td></td>
<td>2016-01-23</td>
</tr>
<tr>
<td></td>
<td>2016-01-24</td>
</tr>
</tbody>
</table>

[16 rows x 2 columns]

Now you can do:

In [14]: df.groupby('group').resample('1D').ffill()

Out[14]:

<table>
<thead>
<tr>
<th>group</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2016-01-03</td>
</tr>
<tr>
<td></td>
<td>2016-01-04</td>
</tr>
<tr>
<td></td>
<td>2016-01-05</td>
</tr>
<tr>
<td></td>
<td>2016-01-06</td>
</tr>
<tr>
<td></td>
<td>2016-01-07</td>
</tr>
<tr>
<td></td>
<td>2016-01-08</td>
</tr>
<tr>
<td></td>
<td>2016-01-09</td>
</tr>
<tr>
<td>...</td>
<td>... ...</td>
</tr>
<tr>
<td>2</td>
<td>2016-01-18</td>
</tr>
<tr>
<td></td>
<td>2016-01-19</td>
</tr>
<tr>
<td></td>
<td>2016-01-20</td>
</tr>
<tr>
<td></td>
<td>2016-01-21</td>
</tr>
<tr>
<td></td>
<td>2016-01-22</td>
</tr>
<tr>
<td></td>
<td>2016-01-23</td>
</tr>
<tr>
<td></td>
<td>2016-01-24</td>
</tr>
</tbody>
</table>

[16 rows x 2 columns]

1.8.1.3 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

**.where() and .mask()**

These can accept a callable for the condition and other arguments.

```python
In [15]: df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})
```

```python
In [16]: df.where(lambda x: x > 4, lambda x: x + 10)
```

```text
Out[16]:
   A  B  C
0  11 14  7
1  12  5  8
2  13  6  9
```

**.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 [17]: df.loc[lambda x: x.A >= 2, lambda x: x.sum() > 10]
```

```text
Out[17]:
   B  C
0  5  8
1  6  9
```

```python
# callable returns list of labels
In [18]: df.loc[[1, 2], ['A', 'B']]
```

```text
Out[18]:
   A  B
0  2  5
1  3  6
```

**[] indexing**

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 [19]: df['A']
```

```text
Out[19]:
0  1
1  2
2  3
Name: A, dtype: int64
```

Using these methods / indexers, you can chain data selection operations without using temporary variable.
In [20]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [21]: (bb.groupby(['year', 'team'])
...: .sum()
...: .loc[lambda df: df.r > 100]
...: )

Out[21]:

<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>
</tr>
</thead>
<tbody>
<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>10.0</td>
<td>1.0</td>
<td>105</td>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<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>60</td>
</tr>
<tr>
<td></td>
<td>NYN</td>
<td>13</td>
<td>622</td>
<td>1854</td>
<td>240</td>
<td>509</td>
<td>101</td>
<td>3</td>
<td>61</td>
<td>243.0</td>
<td>22.0</td>
<td>4.0</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>SFN</td>
<td>5</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>
</tr>
<tr>
<td></td>
<td>TEX</td>
<td>2</td>
<td>198</td>
<td>729</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>
</tr>
<tr>
<td></td>
<td>TOR</td>
<td>4</td>
<td>459</td>
<td>1408</td>
<td>187</td>
<td>378</td>
<td>96</td>
<td>2</td>
<td>58</td>
<td>223.0</td>
<td>4.0</td>
<td>2.0</td>
<td>190</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>year</th>
<th>team</th>
<th>SO</th>
<th>IBB</th>
<th>HBP</th>
<th>SH</th>
<th>SF</th>
<th>GIDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>CIN</td>
<td>127.0</td>
<td>14.0</td>
<td>1.0</td>
<td>1.0</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>DET</td>
<td>176.0</td>
<td>3.0</td>
<td>10.0</td>
<td>4.0</td>
<td>8.0</td>
<td>28.0</td>
</tr>
<tr>
<td></td>
<td>HOU</td>
<td>212.0</td>
<td>3.0</td>
<td>9.0</td>
<td>16.0</td>
<td>6.0</td>
<td>17.0</td>
</tr>
<tr>
<td></td>
<td>LAN</td>
<td>141.0</td>
<td>8.0</td>
<td>9.0</td>
<td>3.0</td>
<td>8.0</td>
<td>29.0</td>
</tr>
<tr>
<td></td>
<td>NYN</td>
<td>310.0</td>
<td>8.0</td>
<td>9.0</td>
<td>3.0</td>
<td>8.0</td>
<td>29.0</td>
</tr>
<tr>
<td></td>
<td>SFN</td>
<td>188.0</td>
<td>5.0</td>
<td>5.0</td>
<td>2.0</td>
<td>8.0</td>
<td>16.0</td>
</tr>
<tr>
<td></td>
<td>TEX</td>
<td>140.0</td>
<td>4.0</td>
<td>5.0</td>
<td>2.0</td>
<td>8.0</td>
<td>16.0</td>
</tr>
<tr>
<td></td>
<td>TOR</td>
<td>265.0</td>
<td>12.0</td>
<td>16.0</td>
<td>4.0</td>
<td>16.0</td>
<td>38.0</td>
</tr>
</tbody>
</table>

1.8.1.4 Partial string indexing on `DateTimeIndex` when part of a `MultiIndex`  

Partial string indexing now matches on `DateTimeIndex` when part of a `MultiIndex` (GH10331)

In [22]: dft2 = pd.DataFrame(np.random.randn(20, 1),
...: columns=['A'],
...: index=pd.MultiIndex.from_product([pd.date_range('20130101' ...
...:    '20130104'),
...:    ['a', 'b'])))

In [23]: dft2
Out[23]:

<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>b</td>
</tr>
<tr>
<td>2013-01-02 12:00:00 a</td>
</tr>
<tr>
<td>...     ...</td>
</tr>
<tr>
<td>2013-01-04 00:00:00 b</td>
</tr>
</tbody>
</table>
2013-01-04 12:00:00 a 0.749185
   b -0.675521
2013-01-05 00:00:00 a 0.440266
   b 0.688972
2013-01-05 12:00:00 a -0.276646
   b 1.924533

[20 rows x 1 columns]

**In [24]:** dft2.loc['2013-01-05']

    →

A
2013-01-05 00:00:00 a 0.440266
   b 0.688972
2013-01-05 12:00:00 a -0.276646
   b 1.924533

On other levels

**In [25]:** idx = pd.IndexSlice

**In [26]:** dft2 = dft2.swaplevel(0, 1).sort_index()

**In [27]:** dft2

**Out[27]:**

A
a 2013-01-01 00:00:00 0.156998
   2013-01-01 12:00:00 1.057633
   2013-01-02 00:00:00 -0.524627
   2013-01-02 12:00:00 1.910759
   2013-01-03 00:00:00 0.513082
   2013-01-03 12:00:00 1.043945
   2013-01-04 00:00:00 1.459927

... ...

b 2013-01-02 12:00:00 0.787965
   2013-01-03 00:00:00 -0.546416
   2013-01-03 12:00:00 2.107785
   2013-01-04 00:00:00 1.015405
   2013-01-04 12:00:00 -0.675521
   2013-01-05 00:00:00 0.688972
   2013-01-05 12:00:00 1.924533

[20 rows x 1 columns]

**In [28]:** dft2.loc[idx[:, '2013-01-05'], :]

    →

A
a 2013-01-05 00:00:00 0.440266
   2013-01-05 12:00:00 -0.276646
b 2013-01-05 00:00:00 0.688972
   2013-01-05 12:00:00 1.924533

### 1.8.1.5 Assembling Datetimes

`pd.to_datetime()` has gained the ability to assemble datetimes from a passed in DataFrame or a dict.
pandas: powerful Python data analysis toolkit, Release 0.21.0

(GH8158).

```python
In [29]: df = pd.DataFrame({'year': [2015, 2016],
          ....:                'month': [2, 3],
          ....:                'day': [4, 5],
          ....:                'hour': [2, 3]})

In [30]: df
Out[30]:
          day  hour  month  year
0         4      2       2  2015
1         5      3       3  2016

Assembling using the passed frame.

In [31]: pd.to_datetime(df)
Out[31]:
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.

In [32]: pd.to_datetime(df[['year', 'month', 'day']])
Out[32]:
0 2015-02-04
1 2016-03-05
dtype: datetime64[ns]
```

1.8.1.6 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 [33]: idx = pd.Index([1., 2., 3., 4.], dtype='float')
# default, allow_fill=True, fill_value=None
In [34]: idx.take([2, -1])
```

124 Chapter 1. What’s New
• Index now supports `.str.get_dummies()` which returns MultiIndex, see *Creating Indicator Variables* (GH10008, GH10103).

```
In [36]: idx = pd.Index(['a|b', 'a|c', 'b|c'])
In [37]: idx.str.get_dummies('|')
Out[37]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
          labels=[[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)

### 1.8.2 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).

```
In [38]: s = pd.SparseArray([np.nan, np.nan, 1, 2, 3, np.nan, 4, 5, np.nan, 6])
In [39]: s.take(0)
Out[39]: nan
In [40]: s.take([1, 2, 3])
Out[40]:
[nan, 1.0, 2.0]
Fill: nan
IntIndex
Indices: array([1, 2], dtype=int32)
```

• 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 ** 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 `SparseArray.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 `SparseArray.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)

1.8.3 API changes

1.8.3.1 `.groupby(..).nth()` changes

The index in `.groupby(..).nth()` output is now more consistent when the `as_index` argument is passed (GH11039):

```python
In [41]: df = DataFrame({'A' : ['a', 'b', 'a'],
                ....:       'B' : [1, 2, 3]})

In [42]: df
Out[42]:
   A  B
0  a  1
1  b  2
2  a  3

Previous Behavior:
```
```python
In [3]: df.groupby('A', as_index=True)['B'].nth(0)
Out[3]:
   0  1
   1  2
Name: B, dtype: int64
```
```python
In [4]: df.groupby('A', as_index=False)['B'].nth(0)
Out[4]:
   0  1
   1  2
Name: B, dtype: int64
```
New Behavior:

```
In [43]: df.groupby('A', as_index=True)['B'].nth(0)
Out[43]:
      A
a  1
b  2
Name: B, dtype: int64

In [44]: df.groupby('A', as_index=False)['B'].nth(0)
Out[44]:
    0  1
   1  2
Name: B, dtype: int64
```

Furthermore, previously, a `.groupby` would always sort, regardless if `sort=False` was passed with `.nth()`.

```
In [45]: np.random.seed(1234)
In [46]: df = pd.DataFrame(np.random.randn(100, 2), columns=['a', 'b'])
In [47]: df['c'] = np.random.randint(0, 4, 100)

Previous Behavior:

```
In [4]: df.groupby('c', sort=True).nth(1)
Out[4]:
   a   b
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
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524
```

New Behavior:

```
In [48]: df.groupby('c', sort=True).nth(1)
Out[48]:
   a   b
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524
```

```
In [49]: df.groupby('c', sort=False).nth(1)
```

1.8.3.2 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:

```python
In [50]: sp = pd.SparseDataFrame([1, 2, 3])
In [51]: sp
Out[51]:
   0  1
   1  2
   2  3
```

Previous behaviour:

```python
In [2]: np.cumsum(sp, axis=0)
...:
TypeError: cumsum() takes at most 2 arguments (4 given)
```

New behaviour:

```python
In [52]: np.cumsum(sp, axis=0)
Out[52]:
   0  1
   1  3
   2  6
```

1.8.3.3 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).

```python
In [53]: df = pd.DataFrame({'date': pd.to_datetime(['10/10/2000', '11/10/2000']),
                      'value': [10, 13]})

In [54]: df
Out[54]:
      date  value
0 2000-10-10    10
1 2000-11-10    13
```
Previous behavior:

```
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 2000-10-31 value 10 2000-11-30 value 13
```

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
```

### 1.8.3.4 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:

```
In [1]: df = pd.read_csv(StringIO(''), engine='c')
... ValueError: No columns to parse from file
In [2]: df = pd.read_csv(StringIO(''), engine='python')
... StopIteration
```

New behaviour:

```
In [1]: df = pd.read_csv(StringIO(''), engine='c')
... 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)

### 1.8.3.5 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 OutOfBoundsDatetim exception will be raised when an out-of-range value is encountered for that unit when errors='raise'. (GH11758, GH13052, GH13059)

Previous behaviour:

```
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'
```

### 1.8.3.6 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 raise `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)

### 1.8.3.7 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)

### 1.8.4 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)

### 1.8.5 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(...) with a `PeriodIndex` 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 coercable 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 `Float64Index/Int64Index` to an `Int64Index` (GH12881)
• Bug in roundtripping 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.rename() loses name if its dtype is category (GH12835)
- Bug in Series.rename() loses timezone info (GH12835)
- Bug in Series.rename() 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)

**1.9 v0.18.0 (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
  – `.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
1.9.1 New features

1.9.1.1 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)

```python
In [1]: np.random.seed(1234)
In [2]: df = pd.DataFrame({'A' : range(10), 'B' : np.random.randn(10)})
In [3]: df
Out[3]:
   A   B
0  0  0.471435
1  1 -1.190976
2  2  1.432707
3  3 -0.312652
4  4 -0.720589
5  5  0.887163
6  6  0.859588
7  7 -0.636524
8  8  0.015696
9  9 -2.242685
```

Previous Behavior:

```python
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  0.237722
3  2 -0.023640
4  3  0.133155
5  4 -0.048693
6  5  0.342054
7  6  0.370076
8  7  0.079587
9  8 -0.954504
```

New Behavior:

```python
In [4]: r = df.rolling(window=3)
In [5]: r
Out[5]: Rolling [window=3,center=False,axis=0]
```

These show a descriptive repr
with tab-completion of available methods and properties.

The methods operate on the Rolling object itself

They provide getitem accessors

And multiple aggregations
1.9.1.2 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)

```
In [9]: s = pd.Series(np.random.randn(5))
In [10]: s.rename('newname')
Out[10]:
   0    1.150036
   1    0.991946
   2    0.953324
   3   -2.021255
   4   -0.334077
Name: newname, 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
```

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.

1.9.1.3 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]: 80

1.9.1.4 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`.

In [1]: pd.Series(["a1", "b2", "c3"]).str.extract("\[ab\](\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...
Out[1]:
   0  1
   1  2
   2 NaN
dtype: object

Extracting a regular expression with one group returns a `Series` if `expand=False`.

In [16]: pd.Series(["a1", "b2", "c3"]).str.extract("\[ab\](\d)\", expand=False)
Out[16]:
   0  1
   1  2
   2 NaN
dtype: object

It returns a `DataFrame` with one column if `expand=True`.

In [17]: pd.Series(["a1", "b2", "c3"]).str.extract("\[ab\](\d)\", expand=True)
Out[17]:
   0  1
   1  2
   2 NaN

Calling on an `Index` with a regex with exactly one capture group returns an `Index` if `expand=False`.  

1.9. v0.18.0 (March 13, 2016)
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.

In [21]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[21]:
   letter
0    A
1    B
2    C

Calling on an Index with a regex with more than one capture group 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

It returns a DataFrame if expand=True.

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

In summary, extract (expand=True) always returns a DataFrame with a row for every subject string, and a column for every capture group.

1.9.1.5 Addition of str.extractall

The .str.extractall method was added (GH11386). Unlike extract, which returns only the first match.

In [23]: s = pd.Series(["ala2", "b1", "c1"], ["A", "B", "C"])

In [24]: s
Out[24]:
   A  ala2
   B   b1
   C   c1
dtype: object

In [25]: s.str.extract("(?P<letter>[ab])(?P<digit>\d)", expand=False)
Out[25]:
   letter digit
   A    a  1
   B    b  1
   C   NaN  NaN

The extractall method returns all matches.
In [26]: s.str.extractall("(?P<letter>[ab]) (?P<digit>\d)")
Out[26]:
<table>
<thead>
<tr>
<th>letter</th>
<th>digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td></td>
</tr>
<tr>
<td>A 0</td>
<td>a 1</td>
</tr>
<tr>
<td>1 a 2</td>
<td></td>
</tr>
<tr>
<td>B 0</td>
<td>b 1</td>
</tr>
</tbody>
</table>

1.9.1.6 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. (GH1435).

A new, friendlier `ValueError` is added to protect against the mistake of supplying the `sep` as an arg, rather than as a kwarg. (GH11334).

In [27]: pd.Series(['a','b',np.nan,'c']).str.cat(sep=' ')
Out[27]: 'a b c'

In [28]: pd.Series(['a','b',np.nan,'c']).str.cat(sep=' ', na_rep='?')

ValueError: Did you mean to supply a `sep` keyword?

1.9.1.7 Datetimelike rounding

DatetimeIndex, Timestamp, TimedeltaIndex, Timedelta have gained the `.round()`, `.floor()` and `.ceil()` method for datetimelike rounding, flooring and ceiling. (GH4314, GH11963)

Naive datetimes

In [29]: dr = pd.date_range('20130101 09:12:56.1234', periods=3)

In [30]: dr
                      dtype='datetime64[ns]', freq='D')

In [31]: dr.round('s')

→ 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]

→ Timestamp('2013-01-01 09:12:56.123400', freq='D')

In [33]: dr[0].round('10s')

→ Timestamp('2013-01-01 09:13:00')
Tz-aware are rounded, floored and ceiled in local times

```python
In [34]: dr = dr.tz_localize('US/Eastern')
In [35]: dr
Out[35]: DatetimeIndex(['2013-01-01 09:12:56.123400-05:00',
                      '2013-01-02 09:12:56.123400-05:00',
                      '2013-01-03 09:12:56.123400-05:00'],
               dtype='datetime64[ns, US/Eastern]', freq='D')
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

```python
In [37]: t = 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)
```

In addition, .round(), .floor() and .ceil() will be available thru the .dt accessor of Series.

```python
In [40]: 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
dtype: datetime64[ns, US/Eastern]
In [44]: s.dt.round('D')
Out[44]:
0  2013-01-01 00:00:00-05:00
```
1.9.1.8 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  1
   1  2
   2  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=None))
0.0,1
1.0,2
2.0,3
```

1.9.1.9 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:
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    int64
dtype: object

New Behavior:

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
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    int64
dtype: object

When a DataFrame’s integer slice is partially updated with a new slice of floats that could potentially be downcasted to integer without losing precision, the dtype of the slice will be set to float instead of integer.

Previous Behavior:

In [4]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
                      columns=list('abc'),
                      index=[[4,4,8], [8,10,12]])

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.])
New Behavior:

```python
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
0  4  8  1  2  3
1 10  4  5  6
2  8 12  7  8  9
```

1.9.1.10 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.

```python
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
```
1.9.1.11 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.

1.9.1.12 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)

1.9.1.13 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 \_float16 typecode (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 codebase has been PEP-ified (GH12096)

1.9.2 Backwards incompatible API changes

- the leading whitespaces 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)

1.9.2.1 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
dtype: timedelta64[ns]

Timedelta division by floats now works.

In [67]: pd.Timedelta('1s') / 2.0

  Out[67]: Timedelta('0 days 00:00:00.500000')

Subtraction by Timedelta in a Series by a Timestamp works (GH11925)

In [68]: ser = pd.Series(pd.timedelta_range('1 day', periods=3))

In [69]: ser

  Out[69]:
         0  1 days
         1  2 days
         2  3 days
dtype: timedelta64[ns]

In [70]: pd.Timestamp('2012-01-01') - ser

  Out[70]:
         0  2011-12-31
         1  2011-12-30
         2  2011-12-29
dtype: datetime64[ns]

NaT.isoformat() now returns 'NaT'. This change allows pd.Timestamp to rehydrate any timestamp like object from its isoformat (GH12300).

1.9.2.2 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.
1.9.2.3 Signature change for .rank

Series.rank and DataFrame.rank now have the same signature (GH11759)

Previous signature

```
In [3]: pd.Series([0,1]).rank(method='average', na_option='keep',
       ...:                        ascending=True, pct=False)
Out[3]:
          0    1
       1    2
       dtype: float64

In [4]: pd.DataFrame([0,1]).rank(axis=0, numeric_only=None,
       ...:                        method='average', na_option='keep',
       ...:                        ascending=True, pct=False)
Out[4]:
       0    1
       0    1
       1    2
```

New signature

```
In [71]: pd.Series([0,1]).rank(axis=0, method='average', numeric_only=None,
       ...:                        na_option='keep', ascending=True, pct=False)
Out[71]:
       0    1.0
       1    2.0
       dtype: float64

In [72]: pd.DataFrame([0,1]).rank(axis=0, method='average', numeric_only=None,
       ...:                        na_option='keep', ascending=True, pct=False)
```

1.9.2.4 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.

```
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)
```

```
In [76]: d + pd.offsets.QuarterBegin(n=0, startingMonth=1)
```

1.9. v0.18.0 (March 13, 2016)
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')
```

### 1.9.2.5 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
```

**Previous API:**

You would write a resampling operation that immediately evaluates. If a `how` parameter was not provided, it would default to `how='mean'`.

```
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 0.624998 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

```
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.

```
In [82]: r = df.resample('2s')
In [83]: r
Out[83]: DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left, label=left,
       →convention=start, base=0]
```

**Downsampling**

You can then use this object to perform operations. These are downsampling operations (going from a higher frequency to a lower one).

```
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
```

```
In [85]: r.sum()
Out[85]:
          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
```

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.

```python
In [87]: r.agg({'A': 'mean', 'B': 'sum'})
Out[87]:
       A     B
2010-01-01  0.485748  0.894701  
2010-01-01  0.820801  1.588635  
2010-01-01  0.433985  0.629165  
2010-01-01  0.624988  1.219477  
2010-01-01  0.510470  1.068634  
```

These accessors can of course, be combined

```python
In [88]: r[['A','B']].agg(['mean','sum'])
Out[88]:
        mean   sum          mean    sum
2010-01-01  0.485748  0.971495  0.447351  0.894701  
2010-01-01  0.820801  1.641602  0.794317  1.588635  
2010-01-01  0.433985  0.867969  0.314582  0.629165  
2010-01-01  0.624988  1.249976  0.609738  1.219477  
2010-01-01  0.510470  1.020940  0.534317  1.068634  
```

**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=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, 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
```
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
2011-03-31  4
Freq: M, dtype: int64
```

**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:

```python
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]:
          A          B          C          D
2010-01-01 09:00:00  4.857476  4.473507  3.570960  7.936154
2010-01-01 09:00:02  8.208011  7.943173  3.640340  5.310957
2010-01-01 09:00:04  4.339846  3.145823  4.241039  6.257326
2010-01-01 09:00:06  6.249881  6.097384  6.331650  6.124518
2010-01-01 09:00:08  5.104699  5.343172  5.732009  8.069486
```

However, getting and assignment operations directly on a `Resampler` will raise a `ValueError`:

```python
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]:
A   0.433985
B   0.314582
C   0.357096
D   0.531096
dtype: float64
```

The new API will:

```
In [92]: df.resample('2s').min()
Out [92]:
       A    B    C    D
2010-01-01 09:00:00 0.191519 0.272593 0.276464 0.785359
2010-01-01 09:00:02 0.683463 0.712702 0.357817 0.500995
2010-01-01 09:00:04 0.364886 0.013768 0.075381 0.368824
2010-01-01 09:00:06 0.316836 0.568099 0.397203 0.436173
2010-01-01 09:00:08 0.218792 0.143767 0.442141 0.704581
```

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
dtype: float64
```

1.9.2.6 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
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
```
In version 0.18.0, a new `inplace` keyword was added to choose whether the assignment should be done inplace or return a copy.

```python
In [96]: df
Out[96]:
     a  b  c
0  0.0 0.0 0.0
1  2.5 1.3 3.5
2  5.0 2.7 7.0
3  7.5 3.1 10.5
4 10.0 4.4 14.0
```

```python
In [97]: df.eval('d = c - b', inplace=False)
<no output>
```

```python
In [98]: df
<no output>
```

```python
In [99]: df.eval('d = c - b', inplace=True)
Out[99]:
     a  b  c  d
0  0.0 0.0 0.0 0.0
1  2.5 1.3 3.5 2.5
2  5.0 2.7 7.0 5.0
3  7.5 3.1 10.5 7.5
4 10.0 4.4 14.0 10.0
```

**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.
pandas: powerful Python data analysis toolkit, Release 0.21.0

In [101]: df.query('a > 5')
Out[101]:
     a   b   c   d
3   7.5  3.0  10.5  7.5
4  10.0  4.0  14.0  10.0

In [102]: df.query('a > 5', inplace=True)

In [103]: df
Out[103]:
     a   b   c   d
3   7.5  3.0  10.5  7.5
4  10.0  4.0  14.0  10.0

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.

In [104]: df
Out[104]:
     a   b   c   d
3   7.5  3.0  10.5  7.5
4  10.0  4.0  14.0  10.0

In [105]: df.eval(""
       .....:  e = d + a
       .....:  f = e - 22
       .....:  g = f / 2.0""", inplace=True)

In [106]: df
Out[106]:
     a   b   c   d   e   f   g
3   7.5  3.0  10.5  7.5  15.0 -7.0  -3.5
4  10.0  4.0  14.0  10.0  20.0 -2.0  -1.0

1.9.2.7 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, dtype: int64

This will now raise.
In 

- .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 <= 1 (GH12492)

1.9.2.8 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)
• 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).

1.9.2.9 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)

```
In [109]: s = pd.Series([1, 2, 3], index=[4, 5, 6])

In [110]: s
Out[110]:
4  1
5  2
6  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
dtype: int64
```

Previous Behavior:

```
# 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
```
# this is label indexing

```python
In [4]: s.loc[5.0]
```

```
FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
```

```python
Out[4]: 2
```

# .ix would coerce 1.0 to the positional 1, and index

```python
In [5]: s2.ix[1.0] = 10
```

```
FutureWarning: scalar indexers for index type Index should be integers and not floating point
```

```python
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
```

```python
In [114]: s.loc[5.0]
```

```
Out[114]: 2
```

and setting

```python
In [115]: s_copy = s.copy()
```

```python
In [116]: s_copy[5.0] = 10
```

```python
In [117]: s_copy
```

```
Out[117]:
   4   1
  5 10
  6  3
dtype: int64
```

```python
In [118]: s_copy = s.copy()
```

```python
In [119]: s_copy.loc[5.0] = 10
```

```python
In [120]: s_copy
```

```
Out[120]:
   4   1
  5 10
```
Positional setting with `.ix` and a float indexer will ADD this value to the index, rather than previously setting the value by position.

```
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`.

```
In [121]: s.loc[5.0:6]
Out[121]:
5 2
6 3
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
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
dtype: int64
```

### 1.9.2.10 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)

1.9.3 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 whitespace preceding the time zone (GH9714)

1.9.4 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 (GH12232)
• 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 to_numeric where it does not raise if input is more than one dimension (GH11776)
- 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 (GH1193)
- 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 NotImplementedException 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 multi-indexes (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.util.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 multi-indexes 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)
pandas: powerful Python data analysis toolkit, Release 0.21.0

1.10 v0.17.1 (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
- Performance Improvements
- Bug Fixes

1.10.1 New features

1.10.1.1 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.

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. Acesses 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 = DataFrame(np.random.randn(10, 5), columns=list('abcde'))
In [3]: html = df.style.background_gradient(cmap='viridis', low=.5)
```

We can render the HTML to get the following table.
Styler interacts nicely with the Jupyter Notebook. See the documentation for more.

### 1.10.2 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._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.itertuples()` 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 = DataFrame({'A': ['foo']*1000})

In [5]: df['B'] = df['A'].astype('category')

# shows the '+' as we have object dtypes
In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
A   1000 non-null object
B   1000 non-null category
dtypes: category(1), object(1)
memory usage: 9.0+ KB

# we have an accurate memory assessment (but can be expensive to compute this)
In [7]: df.info(memory_usage='deep')

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
A   1000 non-null object
B   1000 non-null category
dtypes: category(1), object(1)
memory usage: 75.4 KB
```

- **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]:
```

1.10. v0.17.1 (November 21, 2015)
0  a
1  a
2  b
3  b
dtype: category
Categories (2, object): [a, b]

In [11]: s.str.contains("a")
Out[11]:
0   True
1   True
2  False
3  False
dtype: bool

In [12]: date = pd.Series(pd.date_range('1/1/2015', periods=5)).astype('category')
In [13]: date
Out[13]:
0  2015-01-01
1  2015-01-02
2  2015-01-03
3  2015-01-04
4  2015-01-05
dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-04, ...

In [14]: date.dt.day
Out[14]:
0   1
1   2
2   3
3   4
4   5
dtype: int64

• **pivot_table** now has a `margins_name` argument so you can use something other than the default of ‘All’ (GH3333)

• 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)

1.10.3 API changes

• raise `NotImplementedError` in `Index.shift` for non-supported index types (GH8038)

• `min` and `max` reductions on `datetime64` and `timedelta64` dtyped 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)

1.10.3.1 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)

1.10.4 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)

1.10.5 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 multi-index 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, (GH11162)

• 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 (GH1124)

• Bug in `df.groupby(...).apply(func)` when a func returns a Series containing a new datetimelike column (GH11324)

• Bug in `pandas.json` when file to load is big (GH11344)

• Bugs in `to_excel` with duplicate columns (GH11007, GH10982, GH10970)

• Fixed a bug that prevented the construction of an empty series of dtype `datetime64[ns, tz]` (GH11245).

• Bug in `read_excel` with multi-index containing integers (GH11317)

• Bug in `to_excel` with openpyxl 2.2+ and merging (GH11408)

• Bug in `DataFrame.to_dict()` produces a np.datetime64 object instead of Timestamp when only datetime is present in data (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 MultiIndexes (GH11600)
- 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)

### 1.11 v0.17.0 (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
    - `strftime`
    - `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

1.11.1 New features

1.11.1.1 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.

```python
In [1]: df = DataFrame({'A': date_range('20130101', periods=3),
                   'B': date_range('20130101', periods=3, tz='US/Eastern'),
                   'C': date_range('20130101', periods=3, tz='CET')})

In [2]: df
Out[2]:
          A          B          C
0 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00+01:00
1 2013-01-02 00:00:00-05:00 2013-01-02 00:00:00+01:00
2 2013-01-03 00:00:00-05:00 2013-01-03 00:00:00+01:00

In [3]: df.dtypes

A  datetime64[ns]
B  datetime64[ns, US/Eastern]
C  datetime64[ns, CET]
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, dtype: datetime64[ns, US/Eastern]

In [5]: df.B.dt.tz_localize(None)

0 2013-01-01
1 2013-01-02
..
This uses a new-dtype representation as well, that is very similar in look-and-feel to its numpy cousin \texttt{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 |

\textbf{Note:} There is a slightly different string repr for the underlying \texttt{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] |

\subsection{1.11.1.2 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 \texttt{groupby}, \texttt{nsmallest}, \texttt{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. \texttt{df.groupby('key').sum()} as well as the \texttt{.sum()} operation.

\begin{verbatim}
N = 1000000
ngroups = 10
df = DataFrame({'key' : np.random.randint(0,ngroups,size=N),
                'data' : np.random.randn(N) })
df.groupby('key')['data'].sum()
\end{verbatim}

Releasing of the GIL could benefit an application that uses threads for user interactions (e.g. \texttt{QT}), or performing multi-threaded computations. A nice example of a library that can handle these types of computation-in-parallel is the \texttt{dask} library.
1.11.1.3 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>(...):

```
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:

```
In [12]: df.plot.<TAB>
df.plot.area   df.plot.barh   df.plot.density df.plot.hist df.plot.line ...
df.plot.scatter 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.

1.11.1.4 Additional methods for dt accessor

**strftime**

We are now supporting a Series.dt.strftime method for datetime-likes to generate a formatted string (GH10110). Examples:

```
# DatetimeIndex
In [13]: s = pd.Series(pd.date_range('20130101', periods=4))
```
In [14]: s
Out[14]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
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
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
dtype: object

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
dtype: object

The string format is as the python standard library and details can be found here

```
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
dtype: timedelta64[ns]
1.11.1.5 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.

```python
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 00:00:00')
```

You can use the multiplied freq in PeriodIndex and period_range.

```python
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]', freq='2D')
In [30]: idx + 1
Out[30]: PeriodIndex(['2015-08-03', '2015-08-05', '2015-08-07', '2015-08-09'], dtype='period[2D]', freq='2D')
```

1.11.1.6 Support for SAS XPORT files

read_sas() provides support for reading SAS XPORT format files. (GH4052).
df = pd.read_sas('sas_xport.xpt')

It is also possible to obtain an iterator and read an XPORT file incrementally.

for df in pd.read_sas('sas_xport.xpt', chunksize=10000)
do something(df)

See the docs for more details.

1.11.1.7 Support for Math Functions in .eval()

eval() now supports calling math functions (GH4893)

df = pd.DataFrame({'a': np.random.randn(10)})
df.eval("b = sin(a)")

The support math functions are sin, cos, exp, log, expm1, log1p, sqrt, sinh, cosh, tanh, arcsin, arccos, arctan, arccosh, arcsinh, arcctanh, abs and arctan2.

These functions map to the intrinsics for the NumExpr engine. For the Python engine, they are mapped to NumPy calls.

1.11.1.8 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.

In [31]: df = pd.DataFrame([[1,2,3,4], [5,6,7,8]],
.....: columns = pd.MultiIndex.from_product([[\'foo\', \'bar\'], [\'a\', \'b\']]),
.....: names = [\'coll\', \'col2\'])),
.....: index = pd.MultiIndex.from_product([[\'j\'], [\'l\', \'k\']]),
.....: names = [\'il\', \'i2\'])))

In [32]: df
coll foo bar
    col2 a b a b
    il i2
    j 1 1 2 3 4
    k 5 6 7 8

In [33]: df.to_excel('test.xlsx')

In [34]: df = pd.read_excel('test.xlsx', header=[0,1], index_col=[0,1])

In [35]: df
coll foo bar
    col2 a b a b
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**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

**New**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

**Warning:** Excel files saved in version 0.16.2 or prior that had index names will still be able to be read in, but the `has_index_names` argument must specified to `True`.

### 1.11.1.9 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`.

**1.11. v0.17.0 (October 9, 2015)***
pandas: powerful Python data analysis toolkit, Release 0.21.0

- 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).

### 1.11.1.10 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({u'': ['UK', u''], u'': ['Alice', u'']})
In [37]: df;
```

```plaintext
  0  Alice  UK
  1 しのぶ  日本
```

```python
In [38]: pd.set_option('display.unicode.east_asian_width', True)
In [39]: df;
```

```plaintext
  0  Alice  UK
  1 しのぶ  日本
```

For further details, see [here](#).

### 1.11.1.11 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  NaN     NaN   2.0      right_only
3  NaN     NaN   2.0      right_only

For more, see the updated 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], 1)
Out[1]:
   0 1 2
  0 1 1 4
  1 2 2 5

New Behavior:

In [46]: pd.concat([foo, bar, baz], 1)
Out[46]:
   foo  0 1
  0  1 1 4
  1  2 2 5

- 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
• Added a DataFrame.round method to round the values to a variable number of decimal places (GH10568).

```python
In [49]: df = pd.DataFrame(np.random.random([3, 3]), columns=['A', 'B', 'C'],
                    index=['first', 'second', 'third'])

In [50]: df
Out[50]:
    A    B    C
first 0.34 0.30 0.42
second 0.68 0.88 0.51
third  0.67 0.59 0.62

In [51]: df.round(2)

Out[51]:
    A  B  C
first 0.34 0.30 0.42
second 0.68 0.88 0.51
third  0.67 0.59 0.62
```

• 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).

```python
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 D
```

```python
In [55]: s.drop_duplicates(keep='last')
Out[55]:
2 C
3 A
4 B
5 D
```

```python
In [56]: s.drop_duplicates(keep=False)
Out[56]:
2 C
5 D
```
• Reindex now has a `tolerance` argument that allows for finer control of *Limits on filling while reindexing* (GH10411):

```python
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]:
          t   x
0.1 2000-01-01 0.0
1.9 2000-01-03 2.0
3.5   NaN  NaN
```

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:

```python
In [59]: df = df.set_index('t')

In [60]: df.reindex(pd.to_datetime(['1999-12-31']),
                   method='nearest',
                   tolerance='1 day')

Out[60]:
          x
1999-12-31 0
```

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).
• Read CSV files from AWS S3 incrementally, instead of first downloading the entire file. (Full file download still required for compressed files in Python 2.) (GH11070, GH11073)
• `pd.read_csv` is now able to infer compression type for files read from AWS S3 storage (GH11070, GH11074).

1.11.2 Backwards incompatible API changes

1.11.2.1 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>

### 1.11.2.2 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]: 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]: to_datetime(['2009-07-31', 'asd'], errors='ignore')
Out[62]: array(['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'`.  

11. v0.17.0 (October 9, 2015)
Consistent Parsing

The string parsing of `to_datetime`, `Timestamp` and `DatetimeIndex` has been made consistent. (GH7599) Prior to v0.17.0, `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]: Timestamp('2012Q2')
Traceback
... 
ValueError: Unable to parse 2012Q2
# Results in today's date.
In [2]: 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:

```python
In [63]: Timestamp('2012Q2')
Out[63]: Timestamp('2012-04-01 00:00:00')
In [64]: Timestamp('2014')
Out[64]: Timestamp('2014-01-01 00:00:00')
In [65]: DatetimeIndex(['2012Q2', '2014'])
Out[65]: DatetimeIndex(['2012-04-01', '2014-01-01'], dtype='datetime64[ns]', freq=None)
```

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
In [67]: Timestamp.now()
Out[67]: Timestamp('2017-10-27 10:37:30.76906')
In [68]: Timestamp.now() + offsets.DateOffset(years=1)
Out[68]: Timestamp('2018-10-27 10:37:30.768416')
```

1.11.2.3 Changes to Index Comparisons

Operator equal on `Index` should behavior similarly to `Series` (GH9947, GH10637)

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)
```
In [3]: pd.Index([1, 2, 3]) == pd.Index([2])
Out[3]: array([False, True, False], dtype=bool)

In [4]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
Out[4]: False

New Behavior:

In [8]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[8]: array([ True, False, False], dtype=bool)

In [9]: pd.Index([1, 2, 3]) == pd.Index([2])
ValueError: Lengths must match to compare

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:

In [69]: np.array([1, 2, 3]) == np.array([1])
Out[69]: array([ True, False, False], dtype=bool)

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

1.11.2.4 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 = Series(range(3))

In [72]: s.iloc[1] = None

In [73]: s
Out[73]:
0  0.0
1 NaN
2  2.0
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

1.11. v0.17.0 (October 9, 2015)
Usually you simply want to know which values are null.

```
In [75]: s.isnull()
Out[75]:
0  False
1  True
2  False
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`.

```
In [76]: None == None
Out[76]: True

In [77]: np.nan == np.nan
Out[77]: False
```

### 1.11.2.5 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:**

```
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
```

```
In [27]:
df_with_missing.to_hdf('file.h5',
                       'df_with_missing',
                       format='table',
                       mode='w')

In [28]: pd.read_hdf('file.h5', 'df_with_missing')
```

```
Out [28]:
          col1  col2
0      0.0  1.0
2      2.0  NaN
```
New Behavior:

```python
In [80]: df_with_missing.to_hdf('file.h5',
                      ....: 'df_with_missing',
                      ....: format='table',
                      ....: mode='w')

In [81]: pd.read_hdf('file.h5', 'df_with_missing')
Out[81]:
      col1  col2
0  0.00  1.00
1 NaN NaN
2  2.00 NaN
```

See the *docs* for more details.

### 1.11.2.6 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`.

```python
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.

```python
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.

### 1.11.2.7 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.
1.11.2.8 Changes to \texttt{bool} passed as \texttt{header} in Parsers

In earlier versions of pandas, if a \texttt{bool} was passed the \texttt{header} argument of \texttt{read_csv}, \texttt{read_excel}, or \texttt{read_html} it was implicitly converted to an integer, resulting in \texttt{header=0} for \texttt{False} and \texttt{header=1} for \texttt{True} (GH6113)

A \texttt{bool} input to \texttt{header} will now raise a \texttt{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
```

1.11.2.9 Other API Changes

- Line and kde plot with \texttt{subplots=True} now uses default colors, not all black. Specify \texttt{color='k'} to draw all lines in black (GH9894)
- Calling the \texttt{.value_counts()} method on a Series with a \texttt{categorical} dtype now returns a Series with a \texttt{CategoricalIndex} (GH10704)
- The metadata properties of subclasses of pandas objects will now be serialized (GH10553).
- \texttt{groupby} using \texttt{Categorical} follows the same rule as \texttt{Categorical.unique} described above (GH10508)
- When constructing \texttt{DataFrame} with an array of \texttt{complex64} dtype previously meant the corresponding column was automatically promoted to the \texttt{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>

### 1.11.2.10 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 a 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).

1.11.2.11 Removal of prior version deprecations/changes

• Removal of na_last parameters from Series.order() and Series.sort(), in favor of na_position. (GH5231)

• Remove 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 = DataFrame(np.random.randn(5,2),columns=list('AB'),index=date_range('20130101',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
```

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  1.120555
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]:
     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
```
• 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, GH4864)
• 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)

1.11.3 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 (GH10287)
• 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)

1.11.4 Bug Fixes

• Bug in incorrection computation of .mean() on timedelta64[ns] because of overflow (GH9442)
• Bug in .isin on older numpies (:issue: 11232)
• 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 picking 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 multi-index (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 algos.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 (issue: 11016)
• 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 (GH8984)
• Bug in DataFrame.plot raises ValueError when color name is specified by multiple characters (GH10387)
• Bug in left and right align 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 `plot` result 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)
pandas: powerful Python data analysis toolkit, Release 0.21.0

• 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)

1.12 v0.16.2 (June 12, 2015)

This is a minor bug-fix release from 0.16.1 and includes a 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

1.12.1 New features

1.12.1.1 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
The logic flows from inside out, and function names are separated from their keyword arguments. This can be rewritten as

```python
(df.pipe(h)
    .pipe(g, arg1=1)
    .pipe(f, arg2=2, arg3=3)
)
```

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.poisson takes (formula, data)
In [3]: (bb.query('h > 0')
    ...: .assign(ln_h = lambda df: np.log(df.h))
    ...: .pipe((sm.poisson, 'data'), 'hr ~ ln_h + year + g + C(lg)')
    ...: .fit()
    ...: .summary()
    ...: )
```

```
Optimization terminated successfully.
Current function value: 2.116284
Iterations 24
```

```
Out[3]:
<class 'statsmodels.iolib.summary.Summary'>
```

```
Poisson Regression Results
==============================================================================
Dep. Variable: hr  No. Observations: 68
Model: Poisson  Df Residuals: 63
Method: MLE  Df Model: 4
Date: Fri, 27 Oct 2017  Pseudo R-squ.: 0.6878
Time: 10:37:31  Log-Likelihood: -143.91
converged: True  LL-Null: -460.91
LLR p-value: 6.774e-136
===============================================================================
                  coef    std err          z      P>|z|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept      -1267.3636     457.867     -2.768     0.006    -2164.767    -369.960
C(lg)[T.NL]     -0.2057     0.1011     -2.044     0.041    -0.403       -0.008
ln_h             0.9280     0.1909      4.866     0.000       0.554       1.302
year             0.6301     0.2278      2.762     0.006       0.183       1.077
g              0.0099     0.0040      2.754     0.006       0.003       0.017
```

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)

1.12.1.2 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 here.

- axis parameter of DataFrame.quantile now accepts also index and column. (GH9543)

1.12.2 API Changes

- Holiday now raises NotImplementedError if both offset and observance are used in the constructor instead of returning an incorrect result (GH10217).

1.12.3 Performance Improvements

- Improved Series.resample performance with dtype=datetime64[ns] (GH7754)
- Increase performance of str.split when expand=True (GH10081)

1.12.4 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 funcs 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)

• 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 arithmetics (GH9926)

• Bug in DataFrame construction from nested dict with datetime64 (GH10160)

• Bug in Series construction from dict with datetime64 keys (GH9456)

• 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 NaT (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 timerule (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 DataFrame's, 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).
1.13 v0.16.1 (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**

---

**Warning:** In pandas 0.17.0, the sub-package `pandas.io.data` will be removed in favor of a separately installable package. See [here for details](#) (GH8961)

### 1.13.1 Enhancements

#### 1.13.1.1 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 = DataFrame({'A': np.arange(6),
                  ...:                  'B': 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 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
groupby operations on the index will preserve the index nature as well

```python
In [10]: df2.groupby(level=0).sum()
Out[10]:
   A  B
a  4  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.

```python
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]:
Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [14]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde')))
Out[14]:
   A  B
a 0.0
a 1.0
a 5.0
e NaN

In [15]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde'))).index
Out[15]:
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)

### 1.13.1.2 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 or 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 = Series([0,1,2,3,4,5])
# When no arguments are passed, returns 1
In [2]: example_series.sample()
Out[2]:
3 3
dtype: int64
# One may specify either a number of rows:
In [3]: example_series.sample(n=3)
Out[3]:
5 5
1 1
4 4
dtype: int64
# Or a fraction of the rows:
In [4]: example_series.sample(frac=0.5)
Out[4]:
4 4
1 1
0 0
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
3 3
5 5
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, 0, None, np.nan]
In [8]: example_series.sample(n=1, weights=example_weights2)
Out[8]:
0 0
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 = 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]:
   col1 weight_column
0    9           0.5
1    8           0.4
2    7           0.1
1.13.1.3 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`.

```python
In [11]: idx = 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:

```python
In [13]: idx = Index([ 'a1', 'a2', 'b1', 'b2'])
In [14]: s = Series(range(4), index=idx)
In [15]: s
Out[15]:
a1 0
a2 1
b1 2
b2 3
dtype: int64
In [16]: idx.str.startswith('a')
Out[16]: array([ True, False, False, False], dtype=bool)
In [17]: s[s.index.str.startswith('a')]
```

- The following new methods are accesible via `.str` accessor to apply the function to each values. (GH9766, GH9773, GH10031, GH10045, GH10052)

```plaintext
<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)

```python
In [18]: s = 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]
```

1.13. v0.16.1 (May 11, 2015)
• Improved `extract` and `get_dummies` methods for `Index.str` (GH9980)

1.13.1.4 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)

• `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)
• 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)

### 1.13.2 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)

### 1.13.2.1 Deprecations

• Series.str.split's return_type keyword was removed in favor of expand (GH9847)

### 1.13.3 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]
Length: 4, Freq: D, Timezone: US/Eastern

In [5]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')
Out[5]: <class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-04-14 00:00:00-04:00]
Length: 104, Freq: D, Timezone: US/Eastern
```

New Behavior

```python
In [30]: pd.set_option('display.width', 80)
In [31]: pd.Index(range(4), name='foo')
Out[31]: RangeIndex(start=0, stop=4, step=1, name='foo')

In [32]: pd.Index(range(30), name='foo')
Out[32]: RangeIndex(start=0, stop=30, step=1, name='foo')

In [33]: pd.Index(range(104), name='foo')
Out[33]: RangeIndex(start=0, stop=104, step=1, name='foo')

In [34]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar')
CategoricalIndex(['a', 'bb', 'ccc', 'dddd'], categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar', dtype='category')

In [35]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd']*10, ordered=True, name='foobar')
CategoricalIndex(['a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', '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, name='foobar', dtype='category')

In [36]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd']*100, ordered=True, name='foobar')
```
CategoricalIndex(['a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', ...
'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc',
'dddda'],
categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar',
dtype='category', length=400)

In [37]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')

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 [38]: pd.date_range('20130101', periods=25, freq='D')

DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
'2013-01-25'],
dtype='datetime64[ns]', freq='D')

In [39]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')

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',
'2013-01-11 00:00:00-05:00', '2013-01-12 00:00:00-05:00',
'2013-01-13 00:00:00-05:00', '2013-01-14 00:00:00-05:00',
'2013-01-15 00:00:00-05:00', '2013-01-16 00:00:00-05:00',
'2013-01-17 00:00:00-05:00', '2013-01-18 00:00:00-05:00',
'2013-01-19 00:00:00-05:00', '2013-01-20 00:00:00-05:00',
'2013-01-21 00:00:00-05:00', '2013-01-22 00:00:00-05:00',
'2013-01-23 00:00:00-05:00', '2013-01-24 00:00:00-05:00',
'2013-01-25 00:00:00-05:00', '2013-01-26 00:00:00-05:00',
'2013-01-27 00:00:00-05:00', '2013-01-28 00:00:00-05:00',
'2013-01-29 00:00:00-05:00', '2013-01-30 00:00:00-05:00',
'2013-01-31 00:00:00-05:00'],
dtype='datetime64[ns, US/Eastern]', name='foo', length=104, freq='D')

1.13.4 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)

1.13.5 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.precision (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 whitespace plus one non-space character to be skipped. (GH9710)

• Bug in C csv parser causing spurious NaNs when data started with newline followed by whitespace. (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.__getitem__` 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 multi-indexed 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)
1.14 v0.16.0 (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
  - Other enhancements
- **Backwards incompatible API changes**
  - Changes in Timedelta
  - Indexing Changes
  - Categorical Changes
  - Other API Changes
  - Deprecations
  - Removal of prior version deprecations/changes
- **Performance Improvements**
- **Bug Fixes**
1.14.1 New features

1.14.1.1 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.

```
In [1]: iris = 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

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

Above was an example of inserting a precomputed value. We can also pass in a function to be evaluated.

```
```
In [4]: iris.assign(sepal_ratio = lambda x: (x['SepalWidth'] / x['SepalLength'])).head()
Out[4]:
     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

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.

```
In [5]: (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[5]: <matplotlib.axes._subplots.AxesSubplot at 0x137e0f668>
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

1.14.1.2 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:

```
In [6]: from numpy import nan
In [7]: s = Series([3.0, nan, 1.0, 3.0, nan, nan])
In [8]: s.index = 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 [9]: s
Out[9]:
   A  B  C  D
0  1  2  a  0  3.0
   1  NaN
1  1  2  b  0  1.0
   1  3.0
2  1  2  b  0  NaN
   1  NaN
dtype: float64

# SparseSeries
In [10]: ss = s.to_sparse()
In [11]: ss
Out[11]:
   A  B  C  D
0  1  2  a  0  3.0
   1  NaN
```
<table>
<thead>
<tr>
<th>1</th>
<th>b</th>
<th>0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>NaN</td>
</tr>
</tbody>
</table>

dtype: float64
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 2], dtype=int32)

In [12]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
                                      column_levels=['C', 'D'],
                                      sort_labels=False)

In [13]: A
Out[13]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [14]: A.todense()

In [15]: rows

In [16]: columns

The from_coo method is a convenience method for creating a SparseSeries from a scipy.sparse.coo_matrix:

In [17]: from scipy import sparse

In [18]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
                         shape=(3, 4))

In [19]: A
Out[19]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [20]: A.todense()

In [21]: ss = SparseSeries.from_coo(A)

In [22]: ss
### 1.14.1.3 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>isalnum</th>
<th>isalpha</th>
<th>isdigit</th>
<th>isdigit</th>
<th>isspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>islower</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>isupper</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>istitle</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>isnumeric</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>isdecimal</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>find</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>rfind</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>ljust</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>rjust</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>zfill</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

```python
In [23]: s = Series(['abcd', '3456', 'EFGH'])
```

```python
In [24]: s.str.isalpha()
Out[24]:
0     True
1    False
2     True
dtype: bool
```

```python
In [25]: s.str.find('ab')
Out[25]:
0     0
1    -1
2    -1
```

- `Series.str.pad()` and `Series.str.center()` now accept `fillchar` option to specify filling character (GH9352)

```python
In [26]: s = Series(['12', '300', '25'])
```

```python
In [27]: s.str.pad(5, fillchar='_')
Out[27]:
0     ___12
1    __300
2    ___25
```

- Added `Series.str.slice_replace()`, which previously raised `NotImplementedError` (GH8888)

```python
In [28]: s = Series(['ABCD', 'EFGH', 'IJK'])
```

```python
In [29]: s.str.slice_replace(1, 3, 'X')
Out[29]:
0    AXD
1    EXH
2      IX
```
1.14.14 Other enhancements

- Reindex now supports method='nearest' for frames or series with a monotonic increasing or decreasing index (GH9258):

```python
In [31]: df = pd.DataFrame([x: range(5)])
In [32]: df.reindex([0.2, 1.8, 3.5], method='nearest')
Out[32]:
 x
0.2 0
1.8 2
3.5 4
```

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)

```python
# 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)
pandas: powerful Python data analysis toolkit, Release 0.21.0

- 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 for more (GH9636).

## 1.14.2 Backwards incompatible API changes

### 1.14.2.1 Changes in Timedelta

In v0.15.0 a new scalar type `Timedelta` was introduced, that is a sub-class of `datetime.timedelta`. Mentioned here 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 $10 \times 3600 + 11 \times 60 + 12 = 36672$.

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

```python
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

```python
In [33]: t = pd.Timedelta('1 day, 10:11:12.100123')
In [34]: t.days
Out[34]: 1
In [35]: t.seconds
Out[35]: 36672
In [36]: t.microseconds
Out[36]: 100123
```

Using `.components` allows the full component access

```python
In [37]: t.components
Out[37]: Components(days=1, hours=10, minutes=11, seconds=12, milliseconds=100, microseconds=123, nanoseconds=0)
```
1.14.2.2 Indexing Changes

The behavior of a small sub-set of edge cases for using \texttt{.loc} have changed (GH8613). Furthermore we have improved the content of the error messages that are raised:

- Slicing with \texttt{.loc} where the start and/or stop bound is not found in the index is now allowed; this previously would raise a \texttt{KeyError}. This makes the behavior the same as \texttt{.ix} in this case. This change is only for slicing, not when indexing with a single label.

```python
In [39]: df = DataFrame(np.random.randn(5,4),
                   columns=list('ABCD'),
                   index=date_range('20130101',periods=5))

In [40]: df
Out[40]:
       A         B         C         D
2013-01-01 -0.322795  0.841675  2.390961  0.076200
2013-01-02 -0.566446  0.036142 -2.074978  0.247792
2013-01-03 -0.897157 -0.136795  0.018289  0.755414
2013-01-04  0.215269  0.841009 -1.445810 -1.401973
2013-01-05 -0.100918 -0.548242 -0.144620  0.354020

In [41]: s = Series(range(5),[-2,-1,1,2,3])

In [42]: s
Out[42]:
-2 0
-1 1
 1 2
 2 3
 3 4
dtype: int64
```

Previous Behavior

```python
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 in the [index]'
```

New Behavior

```python
In [43]: df.loc['2013-01-02':'2013-01-10']
Out[43]:
       A         B         C         D
2013-01-02 -0.566446  0.036142 -2.074978  0.247792
2013-01-03 -0.897157 -0.136795  0.018289  0.755414
2013-01-04  0.215269  0.841009 -1.445810 -1.401973
2013-01-05 -0.100918 -0.548242 -0.144620  0.354020
```

1.14. v0.16.0 (March 22, 2015) 217
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 (Int64Index)
```

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
```

### 1.14.2.3 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 = 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
1 1
2 2
dtype: category
Categories (3, int64): [0, 1, 2]

New Behavior

In [45]: s = Series([0,1,2], dtype='category')

In [46]: s
Out[46]:
0 0
1 1
2 2
dtype: category
Categories (3, int64): [0, 1, 2]

In [47]: s.cat.ordered
Out[47]: False

In [48]: s = s.cat.as_ordered()

In [49]: s
Out[49]:
0 0
1 1
2 2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [50]: s.cat.ordered
Out[50]: True

# you can set in the constructor of the Categorical
In [51]: s = Series(Categorical([0,1,2],ordered=True))

In [52]: s
Out[52]:
0 0
1 1
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 = 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 = 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]
```

### 1.14.2.4 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:

  ```
  In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
  Out[2]: {'f':"[3.0,4.2]","i":[1.0,2.0]}
  ```

  Now each column is serialised using its correct dtype:

  ```
  In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
  Out[2]: '{"f":[3.0,4.2],"i":[0,2.0]}'
  ```

- `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`.

  Previous Behavior

<table>
<thead>
<tr>
<th>In [2]:</th>
<th>pd.Series([0,1,2,3], list('abcd'))</th>
<th>pd.Series([4,4,4,4], list('abcd'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out [2]:</td>
<td>a  True</td>
<td>b  True</td>
</tr>
<tr>
<td></td>
<td>dtype: bool</td>
<td></td>
</tr>
</tbody>
</table>

  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.

<table>
<thead>
<tr>
<th>In [2]:</th>
<th>pd.Series([0,1,2,3], list('abcd'))</th>
<th>pd.Series([4,4,4,4], list('abcd'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out [2]:</td>
<td>a  4</td>
<td>b  5</td>
</tr>
<tr>
<td></td>
<td>dtype: int64</td>
<td></td>
</tr>
</tbody>
</table>

• 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 [54]: | p = pd.Series([0, 1]) |
| In [55]: | p / 0 |
| Out [55]: | 0  NaN |
|         | 1  inf |
|         | dtype: float64 |
• 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:

```
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:

```
In [57]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
Out[57]: 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).

### 1.14.2.5 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 using rplot to seaborn: rplot docs.

- 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 thru 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)

### 1.14.2.6 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)

1.14.3 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)

1.14.4 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)

1.14. v0.16.0 (March 22, 2015)
• 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 multi-index 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 thru 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 stubnames 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 [1]: df1 = DataFrame({'x': Series(['a','b','c']), 'y': Series(['d','e','f']))
In [2]: df2 = df1[['x']]
In [3]: df2['y'] = ['g', 'h', 'i']
```

## 1.15 v0.15.2 (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

### 1.15.1 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
     0.123943  x     0.119381
     0.738523  z     0.587304
In [3]: df.index.lexsort_depth
1
# in prior versions this would raise a KeyError
# will now show a PerformanceWarning
In [4]: df.loc[(1, 'z')]
   jolie     jim  joe
   0.738523  1     z
```
# lexically sorting

In [5]: df2 = df.sort_index()

In [6]: df2
Out[6]:
      jolie
   jim  joe
0  x  0.123943
  x  0.119381
1  y  0.587304
  z  0.738523

In [7]: df2.index.lexsort_depth
Out[7]: 2

In [8]: df2.loc[(1,'z')]
Out[8]:
       jolie
   jim  joe
1  z  0.738523

• 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:

1.15. v0.15.2 (December 12, 2014)
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
# 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:

In [18]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])
In [19]: s.loc['c':'a':-1]
Out[19]:
   c
0  2
1  4
2  6
1.15.2 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/to from HDF5 (GH7621). Queries work the same as if it was an object array. However, the category dtype 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})
  ```

- 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
  dtype: bool
  ```

- Panel now supports the all and any aggregation functions. (GH8302):

  ```python
  In [22]: p = pd.Panel(np.random.rand(2, 5, 4) > 0.1)
  In [23]: p.all()
  Out[23]:
  0 1
  0  True  True
  1  True  True
  2 False False
  3  True  True
  ```

- Added support for utcffromtimestamp(), fromtimestamp(), and combine() on Timestamp class (GH5351).

- Added Google Analytics (pandas.io.ga) basic documentation (GH8835). See ‘here-http://pandas.pydata.org/pandas-docs/version/0.15.2/remote_data.html#remote-data-ga’__.
• 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 datatypes of non-NA values for columns that contain NA values and have dtype `object` (GH8778).

### 1.15.3 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)

### 1.15.4 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 multi-index 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 outputing a Multindex 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; buggly 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 whitespace 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 timeseries, would show labels on multiple axes (GH3964).
- 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)

### 1.16 v0.15.1 (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**

#### 1.16.1 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 = Series(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
     3 2013-01-04
     4 2013-01-05
dtype: datetime64[ns]
```

previous behavior:

```python
In [6]: s.dt.hour
Out[6]:
0  0
1  0
2 -1
3  0
4  0
dtype: int64
```

current behavior:

```python
In [4]: s.dt.hour
Out[4]:
0  0.0
1  0.0
2  NaN
3  0.0
```
• groupby with as_index=False will not add erroneous extra columns to result (GH8582):

    np.random.seed(2718281)
    df = pd.DataFrame(np.random.randint(0, 100, (10, 2)),
                      columns=['jim', 'joe'])
    df.head()
    ts = pd.Series(5 * np.random.randint(0, 3, 10))

    df.groupby(ts, as_index=False).max()

    df = pd.DataFrame({'jim': range(5), 'joe': range(5, 10)})
    df
    gr = df.groupby(df['jim'] < 2)

    previous behavior (excludes 1st column from output):

    df.groupby(ts, as_index=False).max()

    current behavior:

    df.groupby(ts, as_index=False).max()

• groupby will not erroneously exclude columns if the column name conflicts with the grouper name (GH8112):

    df = pd.DataFrame({'jim': range(5), 'joe': range(5, 10)})
    df
    gr = df.groupby(df['jim'] < 2)

    previous behavior (excludes 1st column from output):
In [4]: gr.apply(sum)
Out[4]:
   joe
  jim
False  24
True   11

current behavior:

In [13]: gr.apply(sum)
Out[13]:
   joe
  jim
False  9  24
True   1  11

• 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
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
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 callsYYMMDD or putsYYMMDD. 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:

```python
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 Expire Type Symbol  Last
1                     80  2014-11-14 call AAPL141114C00080000  29.05
2                     84  2014-11-14 call AAPL141114C00084000  24.80
3                     85  2014-11-14 call AAPL141114C00085000  24.05
4                     86  2014-11-14 call AAPL141114C00086000  22.76
5                     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]:
                     Strike Expire Type Symbol  Last
1                     109  2014-11-22 call AAPL1411122C00109000  1.48
2                     109  2014-11-28 call AAPL1411128C00109000  1.79
3                     110  2014-11-14 call AAPL1411114C00110000  0.55
4                     110  2014-11-22 call AAPL1411122C00110000  1.02
5                     110  2014-11-28 call AAPL1411128C00110000  1.32
```

- 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).

### 1.16.2 Enhancements

- `concat` permits a wider variety of iterables of pandas objects to be passed as the first parameter (GH8645):

```python
In [17]: from collections import deque
In [18]: df1 = pd.DataFrame([[1, 2, 3]])
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

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
  1  2
  2  3
  0  4
  1  5
  2  6

• 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 = 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 11040
A 8000
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)

### 1.16.3 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 (issue: 8703)
- Bug in slicing a multi-index 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).

1.17 v0.15.0 (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
  – 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 refactorings 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
1.17.1 New features

1.17.1.1 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 = 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]:
 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
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, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
In [7]: df.sort_values("grade")
Out[7]:
    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
In [8]: df.groupby("grade").size()
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>grade</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>very bad</td>
<td>1</td>
</tr>
<tr>
<td>bad</td>
<td>0</td>
</tr>
<tr>
<td>medium</td>
<td>0</td>
</tr>
<tr>
<td>good</td>
<td>2</td>
</tr>
<tr>
<td>very good</td>
<td>3</td>
</tr>
</tbody>
</table>

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`.

1.17.1.2 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 = 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.
Construct a scalar

```python
In [9]: Timedelta('1 days 06:05:01.00003')
Out[9]: Timedelta('1 days 06:05:01.000030')

In [10]: Timedelta('15.5us')
Out[10]: Timedelta('0 days 00:00:00.000015')

In [11]: Timedelta('1 hour 15.5us')
Out[11]: Timedelta('0 days 01:00:00.000015')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [12]: Timedelta('-1us')
Out[12]: Timedelta('-1 days +23:59:59.999999')

# a NaT
In [13]: Timedelta('nan')
Out[13]: NaT
```

Access fields for a Timedelta

```python
In [14]: td = 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

```python
In [18]: TimedeltaIndex(['1 days','1 days, 00:00:05','2 days 00:00:02'])
Out[18]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:02'], dtype='timedelta64[ns]', freq=None)
```

Constructing a TimedeltaIndex with a regular range

```python
In [19]: 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]: timedelta_range(start='1 days',end='2 days',freq='30T')
```

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.
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', '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00', '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00', '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00', '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00', '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00', '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00', '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00', '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00', '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00', '2 days 00:00:00'],
dtype='timedelta64[ns]', freq='30T')

You can now use a TimedeltaIndex as the index of a pandas object

In [21]: s = Series(np.arange(5),
    index = 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, dtype: int64

You can select with partial string selections

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, dtype: int64

Finally, the combination of TimedeltaIndex with DatetimeIndex allow certain combination operations that are NaT preserving:

In [25]: tdi = 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 = 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()

In [30]: (dti - tdi).tolist()

• 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.

1.17.1.3 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 = dict({ t: np.random.randint(100, size=n).astype(t)
.....:     for t in dtypes})
.....:

In [34]: df = 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):
bool 5000 non-null bool
complex128 5000 non-null complex128
datetime64[ns] 5000 non-null datetime64[ns]
float64 5000 non-null float64
int64 5000 non-null int64
object 5000 non-null object
timedelta64[ns] 5000 non-null timedelta64[ns]
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: 289.1+ KB

Additionally memory_usage() is an available method for a dataframe object which returns the memory usage of each column.
In [37]: df.memory_usage(index=True)
Out[37]:
    Index       80
             bool     5000
             complex128  80000
           datetime64[ns]  40000
             float64     4000
              int64     40000
               object    40000
          timedelta64[ns]  40000
             categorical 10920
   dtype: int64

1.17.1.4 .dt accessor

Series has gained an accessor to succinctly return datetime like properties for the values of the Series, if its a
datetime/period like Series. (GH7207) This will return a Series, indexed like the existing Series. See the docs

```python
# datetime
In [38]: s = Series(date_range('20130101 09:10:12',periods=4))

In [39]: s
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
   dtype: datetime64[ns]

In [40]: s.dt.hour
   →
0   9
1   9
2   9
3   9
dtype: int64

In [41]: s.dt.second
   →
0   12
1   12
2   12
3   12
dtype: int64

In [42]: s.dt.day
   →
0   1
1   2
2   3
3   4
dtype: int64
```
This enables nice expressions like this:

```python
In [44]: s[s.dt.day==2]
Out[44]:
1 2013-01-02 09:10:12
```

You can easily produce tz aware transformations:

```python
In [45]: stz = s.dt.tz_localize('US/Eastern')
In [46]: stz
tz
```

You can also chain these types of operations:

```python
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
dtype: datetime64[ns, US/Eastern]
```

The `.dt` accessor works for period and timedelta dtypes.

```python
# period
In [49]: s = Series(period_range('20130101', periods=4, freq='D'))
In [50]: s
dtype: object
In [51]: s.dt.year
```

```python
0 2013
1 2013
2 2013
3 2013
dtype: int64
```
In [52]: s.dt.day

    0  1
    1  2
    2  3
    3  4
dtype: int64

# timedelta
In [53]: s = Series(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
dtype: timedelta64[ns]

In [55]: s.dt.days
Out[55]:
   0  1
   1  1
   2  1
   3  1
dtype: int64

In [56]: s.dt.seconds
Out[56]:
   0   5
   1   6
   2   7
   3   8
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

1.17.1.5 Timezone handling improvements

- \texttt{tz\_localize(\texttt{None})} for \texttt{tz-aware Timestamp} and \texttt{DatetimeIndex} now removes timezone holding local time, previously this resulted in \texttt{Exception} or \texttt{TypeError} (GH7812)

In [58]: ts = Timestamp('2014-08-01 09:00', 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 = DatetimeIndex(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='H')

• 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)

1.17.1.6 Rolling/Expanding Moments improvements

• rolling_min(), rolling_max(), rolling_cov(), and rolling_corr() now return objects with all NaN when len(arg) < min_periods <= window rather than raising. (This makes all rolling functions consistent in this behavior). (GH7766)

Prior to 0.15.0

In [64]: s = Series([10, 11, 12, 13])

In [15]: rolling_min(s, window=10, min_periods=5)
ValueError: min_periods (5) must be <= window (4)

New behavior
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 (window-1)/2 entries.

Now the final (window-1)/2 entries of the result are calculated as if the input arg were followed by
  (window-1)/2 NaN values (or with shrinking windows, in the case of rolling_apply()). (GH7925, GH8269)

Prior behavior (note final value is NaN):

In [7]: 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 = sum([2, 3, NaN])):

In [7]: rolling_sum(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 (mean=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 = Series([10.5, 8.8, 11.4, 9.7, 9.3])

Behavior prior to 0.15.0:

In [39]: 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 = 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(Series([None, 1., 8.]), com=2.)
Out[7]:
0   NaN
1   1.0
2   5.2
dtype: float64
```

```
In [8]: pd.ewma(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(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 0 rather than 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 = Series([1., 2., 0., 4.])
```
In [89]: ewmvar(s, com=2., bias=False)
Out[89]:
0  -2.775558e-16
1  3.000000e-01
2  9.556787e-01
3  3.585799e+00
dtype: float64

In [90]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
Out[90]:
0  1.25
1  1.25
2  1.25
3  1.25
dtype: float64

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 [14]: pd.ewmvar(s, com=2., bias=False)
Out[14]:
0    NaN
1  0.500000
2  1.210526
3  4.089069
dtype: float64

In [15]: pd.ewmvar(s, com=2., bias=False) / pd.ewmvar(s, com=2., bias=True)
Out[15]:
0    NaN
1  2.083333
2  1.583333
3  1.425439
dtype: float64

See *Exponentially weighted moment functions* for details. (GH7912)

1.17.1.7 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')
pd.read_sql_table('table', engine, schema='other_schema')
```

- Added support for writing NaN values with `to_sql` (GH2754).
- Added support for writing datetime64 columns with `to_sql` for all database flavors (GH7103).
1.17.2 Backwards incompatible API changes

1.17.2.1 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:

- df.loc[[3]] would return a frame reindexed by 3 (with all np.nan values)
- df.loc[[3],:] would raise KeyError.

Both will now raise a KeyError. The rule is that at least 1 indexer must be found when using a list-like and .loc (GH7999)

Furthermore in prior versions these were also different:

- df.loc[[1,3]] would return a frame reindexed by [1,3]
- df.loc[[1,3],:] would raise KeyError.

Both will now return a frame reindex by [1,3]. E.g.

```python
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
```

This can also be seen in multi-axis indexing with a `Panel`.

```
In [70]: p = Panel(np.arange(2*3*4).reshape(2,3,4),
       ....:     items=['ItemA','ItemB'],
       ....:     major_axis=[1,2,3],
       ....:     minor_axis=['A','B','C','D'])

In [71]: p
Out[71]:
<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
```

The following would raise `KeyError` prior to 0.15.0:

```
In [5]:
Out[5]:
   ItemA  ItemD
  1   3  NaN
  2   7  NaN
  3  11  NaN
```

Furthermore, `.loc` will raise `KeyError` if no values are found in a multi-index with a list-like indexer:

```
In [72]: s = Series(np.arange(3),dtype='int64'),
       ....:     index=MultiIndex.from_product([['A'],['foo','bar','baz']],
       ....:     names=['one','two'])
       ....: ).sort_index()

In [73]: s
Out[73]:
one two
A bar 1
  baz 2
  foo 0
dtype: int64
In [74]: try:
   ....:     s.loc[['D']]
   ....:     except KeyError as e:
   ....:         print("KeyError: " + str(e))
   ....:     \n\n            \n            "[D'] not in index"
```

- Assigning values to `None` now considers the dtype when choosing an ‘empty’ value (GH7941).

1.17  v0.15.0 (October 18, 2014)  253
Previously, assigning to None in numeric containers changed the dtype to object (or errored, depending on the call). It now uses NaN:

```
In [75]: s = Series([1, 2, 3])
In [76]: s.loc[0] = None
In [77]: s
Out[77]:
0   NaN
1    2.0
2    3.0
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 [78]: s = Series(['a', 'b', 'c'])
In [79]: s.loc[0] = None
In [80]: s
Out[80]:
0  None
1    b
2    c
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 [81]: s = Series([1, 2, 3])
In [82]: s2 = s
In [83]: 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

```python
# the original object
In [84]: s
Out[84]:
0    2.5
1    3.5
2    4.5
dtype: float64

# a reference to the original object
In [85]: s2
Out[85]:
0    2.5
1    3.5
2    4.5
dtype: float64
```

- Made both the C-based and Python engines for `read_csv` and `read_table` ignore empty lines in input as well as whitespace-filled lines, as long as `sep` is not whitespace. 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 [86]: i = date_range('1/1/2011', periods=3, freq='10s', tz='US/Eastern')
In [87]: i
Out[87]:
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 [88]: df = DataFrame( {'a' : i } )
In [89]: df
Out[89]:
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
In [90]: df.dtypes

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</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>dtype: object</td>
</tr>
</tbody>
</table>
```

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 = 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 the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#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):

```python
In [91]: df = DataFrame([[True, 1],[False, 2]],
   ...:    columns=['female','fitness'])
   ...
In [92]: df
Out[92]:
   female  fitness
0     True      1
1    False      2
In [93]: df.dtypes
Out[93]:
   female    bool
   fitness   int64
dtype: object
# dtypes are now preserved
In [95]: df
Out[95]:
   female  fitness
0     True      1
1    False      2
2    False      2
In [96]: df.dtypes
Out[96]:
   female    bool
   fitness   int64
dtype: object
```
• 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)

1.17.2.2 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 similary 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.

1.17.2.3 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)

```python
# +
Index(['a','b','c']) + Index(['b','c','d'])

# should be replaced by
Index(['a','b','c']).union(Index(['b','c','d']))
```
The `infer_types` argument to `read_html()` now has no effect and is deprecated (GH7762, GH7032).

1.17.2.4 Removal of prior version deprecations/changes

- Remove `DataFrame.delevel` method in favor of `DataFrame.reset_index`

1.17.3 Enhancements

Enhancements in the importing/exporting of Stata files:

- Added support for bool, uint8, uint16 and uint32 datatypes 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 [97]: df = DataFrame={['catA': ['foo', 'foo', 'bar'] * 8, 
    ....:     'catB': ['a', 'b', 'c', 'd'] * 6, 
    ....:     'numC': np.arange(24), 
    ....:     'numD': np.arange(24.) + .5})

In [98]: df.describe(include=['object'])
Out[98]:
```
### Requesting all columns is possible with the shorthand ‘all’

```python
In [100]: df.describe(include='all')
```

```plaintext
Out[100]:
<table>
<thead>
<tr>
<th></th>
<th>catA</th>
<th>catB</th>
<th>numC</th>
<th>numD</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>24</td>
<td>24</td>
<td>24.000000</td>
<td>24.000000</td>
</tr>
<tr>
<td>unique</td>
<td>2</td>
<td>4</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>top</td>
<td>foo</td>
<td>d</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>freq</td>
<td>16</td>
<td>6</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>mean</td>
<td>NaN</td>
<td>NaN</td>
<td>11.500000</td>
<td>12.000000</td>
</tr>
<tr>
<td>std</td>
<td>NaN</td>
<td>NaN</td>
<td>7.071068</td>
<td>7.071068</td>
</tr>
<tr>
<td>min</td>
<td>NaN</td>
<td>NaN</td>
<td>0.000000</td>
<td>0.500000</td>
</tr>
<tr>
<td>25%</td>
<td>NaN</td>
<td>NaN</td>
<td>5.750000</td>
<td>6.250000</td>
</tr>
<tr>
<td>50%</td>
<td>NaN</td>
<td>NaN</td>
<td>11.500000</td>
<td>12.000000</td>
</tr>
<tr>
<td>75%</td>
<td>NaN</td>
<td>NaN</td>
<td>17.250000</td>
<td>17.750000</td>
</tr>
<tr>
<td>max</td>
<td>NaN</td>
<td>NaN</td>
<td>23.000000</td>
<td>23.500000</td>
</tr>
</tbody>
</table>
```

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.

```python
In [101]: df = DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'], 'C': [1, 2, 3]})
      ........
In [102]: pd.get_dummies(df)
```

```plaintext
Out[102]:
<table>
<thead>
<tr>
<th></th>
<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>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- 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)

```python
In [103]: business_dates = date_range(start='4/1/2014', end='6/30/2014', freq='B')
In [104]: df = DataFrame(1, index=business_dates, columns=['a', 'b'])
# get the first, 4th, and last date index for each month
In [105]: df.groupby((df.index.year, df.index.month)).nth([0, 3, -1])
Out[105]:
     a  b
2014 4 1 1
      4 1 1
      4 1 1
      5 1 1
      5 1 1
      6 1 1
      6 1 1

• Period and PeriodIndex supports addition/subtraction with timedelta-likes (GH7966)

    If Period freq is D,H,T,S,L,U,N, timedelta-like can be added if the result can have same freq. Otherwise, only the same offsets can be added.

```
- Added experimental compatibility with openpyxl for versions \( \geq 2.0 \). 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)

- Index.isin now supports a level argument to specify which index level to use for membership tests (GH7892, GH7890)
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, False, False, True], dtype=bool)

• Index now supports duplicated and drop_duplicates. (GH4060)

In [118]: idx = Index([1, 2, 3, 4, 1, 2])
In [119]: idx
Out[119]: Int64Index([1, 2, 3, 4, 1, 2], dtype='int64')

In [120]: idx.duplicated()
Out[120]: array([False, False, False, False, True, True], dtype=bool)

In [121]: idx.drop_duplicates()
Out[121]: Int64Index([1, 2, 3, 4], dtype='int64')

• add copy=True argument to pd.concat to enable pass thru of complete blocks (GH8252)
• Added support for numpy 1.8+ data types (bool_, int_, float_, string_) for conversion to R dataframe (GH8400)

1.17.4 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)

1.17.5 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 multi-index slicing with missing indexers (GH7866)

• Bug in multi-index slicing with various edge cases (GH8132)

• Regression in multi-index 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 multi-index 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 DateFormatter 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 multi-index 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.

1.18 v0.14.1 (July 11, 2014)

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

1.18.1 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):

```
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]: Timestamp('2014-01-31 00:00:00')

# 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)

### 1.18.2 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):

```
In [3]: import pandas.tseries.offsets as offsets
In [4]: day = offsets.Day()
In [5]: day.apply(Timestamp('2014-01-01 09:00'))
Out[5]: Timestamp('2014-01-02 09:00:00')
In [6]: day = offsets.Day(normalize=True)
In [7]: day.apply(Timestamp('2014-01-01 09:00'))
Out[7]: Timestamp('2014-01-02 00:00:00')
```

1.18. v0.14.1 (July 11, 2014)
pandas: powerful Python data analysis toolkit, Release 0.21.0

- **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 **NotImplementedError** 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)

```python
In [8]: rng = date_range('3/6/2012 00:00', periods=10, freq='D', ...:
   ...:     tz='dateutil/Europe/London')
   ...
In [9]: rng.tz
Out[9]: 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** whitelist, 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)

### 1.18.3 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)
1.18.4 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 http/2 and the Google apiclient and oauth2client API client libraries which should be more stable and, therefore, reliable than bq.py. See the docs. (GH6937).

1.18.5 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 multi-index 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 timeops 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 multi-index 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 multi-index 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() / eval 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 (GH7131).
• Bug in isnull() when mode.use_inf_as_null == True where isnull wouldn’t test True when it encountered an inf/-inf (GH7135).
• 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 .ixgetitem should always return a Series (GH7150)
• Bug in multi-index slicing with incomplete indexers (GH7399)
• Bug in multi-index 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 (:issue’7306’)
• 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** with 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).

### 1.19 v0.14.0 (May 31, 2014)

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](#).

---

1.19. v0.14.0 (May 31, 2014) 271
– Display interface changes, See Here
– MultiIndexing Using Slicers, See Here.
– Ability to join a singly-indexed DataFrame with a multi-indexed 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)

1.19.1 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)

In [1]: df1 = DataFrame(np.random.randn(5,2),columns=list('AB'))

In [2]: df1
Out[2]:
         A         B
0  1.583584 -0.438313
1 -0.402537 -0.780572
2 -0.141685  0.542241
3  0.370966 -0.251642
4  0.787484  1.666563

In [3]: df1.iloc[:,2:3]

Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [4]: df1.iloc[:,1:3]

→ B
0 -0.438313
1 -0.780572
2  0.542241
3 -0.251642
4  1.666563

In [5]: df1.iloc[4:6]

→
A  B
4 0.787484 1.666563

These are out-of-bounds selections

- df1.iloc[[4,5,6]]
  IndexError: positional indexers are out-of-bounds

- df1.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)::] 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.
pandas: powerful Python data analysis toolkit, Release 0.21.0

- 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)

```
In [6]: i = pd.Index([1, 2, 3, 'a', 'b', 'c'])
In [7]: i[[0,1,2]]
Out[7]: Index([1, 2, 3], dtype='object')
In [8]: i.drop(['a', 'b', 'c'])
Out[8]: Index([1, 2, 3], dtype='object')
```

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
```

# New behavior
```
In [12]: mi
Out[12]: MultiIndex(levels=[['a', 'b'], ['c', 'd']], labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
In [13]: df_multi.set_index(mi)
   0   1
a c 0.471435 -1.190976
d  1.432707 -0.312652
b c -0.720589  0.887163
d  0.859588 -0.636524
```

This also applies when passing multiple indices to `set_index:`
• pairwise keyword was added to the statistical moment functions rolling_cov, rolling_corr, ewm cov, ewm corr, expanding_cov, expanding_corr to allow the calculation of moving window covariance and correlation matrices (GH4950). See Computing rolling pairwise covariances and correlations in the docs.

• 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)

1.19. v0.14.0 (May 31, 2014) 275
• **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
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)

### 1.19.2 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  6  7  8  9 10 11 ...
2001-01-03 12 13 14 15 16 17 ...
2001-01-04 18 19 20 21 22 23 ...
2001-01-05 24 25 26 27 28 29 ...
2001-01-06 30 31 32 33 34 35 ...
2001-01-07 36 37 38 39 40 41 ...
2001-01-08 42 43 44 45 46 47 ...
2001-01-09 48 49 50 51 52 53 ...
2001-01-10 54 55 56 57 58 59 ...
[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  3  4
2001-01-01  0  1  2  3  4 ...
2001-01-02  5  6  7  8  9 ...
2001-01-03 10 11 12 13 14 ...
2001-01-04 15 16 17 18 19 ...
2001-01-05 20 21 22 23 24 ...
2001-01-06 25 26 27 28 29 ...
2001-01-07 30 31 32 33 34 ...
2001-01-08 35 36 37 38 39 ...
2001-01-09 40 41 42 43 44 ...
2001-01-10 45 46 47 48 49 ...
[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

1.19. v0.14.0 (May 31, 2014)
• 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__ setting (GH4553)

### 1.19.3 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)

### 1.19.4 Groupby API Changes

More consistent behaviour for some groupby methods:

• `groupby head` and `tail` now act more like `filter` rather than an aggregation:
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

In [22]: g.apply(lambda x: x.head(1))  # used to simply fall-through

Out[22]:
   A  B
A 1  2
5  6

- groupby head and tail respect column selection:

In [23]: g[['B']].head(1)
Out[23]:
   B
0  2
2  6

- 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

In [24]: df = 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

# this is equivalent to g.first()
In [27]: g.nth(0, dropna='any')
Out[27]:
   B
A
1  4.0
5  6.0

# this is equivalent to g.last()
In [28]: g.nth(-1, dropna='any')
Out[28]:
   B
A
1  4.0
5  6.0
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
In [31]: gf.nth(0, dropna='any')
Out[31]:
   A  B
0  1  4.0
2  5  6.0
```

- groupby will now not return the grouped column for non-cython functions (GH5610, GH5614, GH6732), as its already the index

```
In [32]: df = 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]:
   A
0  1
1  2
In [35]: g.describe()
Out[35]:
   B
   count  mean    std   min   25%  50%  75%   max
   A
0  2.0  1.0  0.0  1.0  1.0  1.0  1.0  1.0  4.0
1  2.0  5.0  0.0  5.0  5.0  5.0  5.0  5.0  7.0

```

- passing as_index will leave the grouped column in-place (this is not change in 0.14.0)

```
In [36]: df = 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  2  2
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  1.0  4.0
1  2.0  5.0  0.0  5.0  5.0  5.0  5.0  5.0  7.0
```

• 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.

1.19.5 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:')
```

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)
```

You can read data from a database by specifying the table name:

```python
In [44]: df = pd.read_sql_table('db_table', engine)
Out[44]:
    A  B
0  1  a
1  2  b
2  3  c
```

or by specifying a sql query:
In [45]: pd.read_sql_query('SELECT * FROM db_table', engine)
Out[45]:
   A B
0 1 a
1 2 b
2 3 c

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).

### 1.19.6 MultiIndexing Using Slicers

In 0.14.0 we added a new way to slice multi-indexed objects. You can slice a multi-index 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:

```
df.loc[(slice('A1','A3'),.....),:]
```

rather than this:

```
df.loc[(slice('A1','A3'),.....)]
```

**Warning:** You will need to make sure that the selection axes are fully lexsorted!
In [46]: def mklbl(prefix,n):
    ....:     return ["%s%s" % (prefix,i) for i in range(n)]

In [47]: index = MultiIndex.from_product([mklbl('A',4),
    ....:     mklbl('B',2),
    ....:     mklbl('C',4),
    ....:     mklbl('D',2)])

In [48]: columns = MultiIndex.from_tuples([('a','foo'),('a','bar'),
    ....:     ('b','foo'),('b','bah')],
    ....:     names=['lvl0', 'lvl1'])

In [49]: df = 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 lvl1
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 21 20 23 22
C3 D0 25 24 27 26
... ... ... ... ...
A3 B1 C1 D1 229 228 231 230
C1 D0 233 232 235 234
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 multi-index slicing using slices, lists, and labels.

In [51]: df.loc[(slice('A1','A3'),slice(None), ['C1','C3']),:]
Out[51]:
lvl0 lvl1
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 109 108 111 110
C3 D0 121 120 123 122
D1 125 124 127 126
... ... ... ... ...
A3 B0 C1 D1 205 204 207 206...
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
     D1  44  46
     C3 D0  56  58
... ... ...
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
[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
     D1  84  86
     C3 D0  88  90
... ... ...
B1 C0 D1 100 102
     C1 D0 104 106
     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']]

_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
  D1 44 46
  C3 D0 56 58
...
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
[32 rows x 2 columns]

Using a boolean indexer you can provide selection related to the values.

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

You can also specify the axis argument to .loc to interpret the passed slicers on a single axis.

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
  D1 13 12 15 14
  C3 D0 25 24 27 26
  D1 29 28 31 30
B1 C1 D0 41 40 43 42
  D1 45 44 47 46
  C3 D0 57 56 59 58
...
A3 B0 C1 D1 205 204 207 206
  C3 D0 217 216 219 218
  D1 221 220 223 222
B1 C1 D0 233 232 235 234
Furthermore you can set the values using these methods

```
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  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
  D1 21 20 23 22
  C3  D0 -10 -10 -10 -10
...
A3  B1  C0  D1  229 228 231 230
  C1  D0 -10 -10 -10 -10
  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]
```

You can use a right-hand-side of an alignable object as well.

```
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 9000 8000 11000 10000
  D1 13000 12000 15000 14000
  C2  D0 17 16 19 18
  D1 21 20 23 22
  C3  D0 25000 24000 27000 26000
...
A3  B1  C0  D1  229 228 231 230
  C1  D0 233000 232000 235000 234000
  D1 237000 236000 239000 238000
  C2  D0 241 240 243 242
  D1 245 244 247 246
  C3  D0 249000 248000 251000 250000
```
1.19.7 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).
  - 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.
1.19.8 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)

1.19.9 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]: 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]: 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]: 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
```
• 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)

• 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.

1.19.10 Known Issues

• OpenPyXL 2.0.0 breaks backwards compatibility (GH7169)

1.19.11 Enhancements

• DataFrame and Series will create a MultiIndex object if passed a tuples dict, See the docs (GH3323)
pandas: powerful Python data analysis toolkit, Release 0.21.0

In [66]: 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]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a b</td>
<td>a b</td>
</tr>
<tr>
<td>A</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>C</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>D</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>9.0</td>
</tr>
</tbody>
</table>

- 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 multi-indexed DataFrame (GH3662)

See the docs. Joining multi-index DataFrames on both the left and right is not yet supported ATM.

In [67]: household = DataFrame(dict(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]:

<table>
<thead>
<tr>
<th>household_id</th>
<th>male</th>
<th>wealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>196087.3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>316478.7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>294750.0</td>
</tr>
</tbody>
</table>

In [69]: portfolio = DataFrame(dict(household_id = [1,2,2,3,3,3,4],
......:
    asset_id = ["nl0000301109","nl0000289783", "gb00b03mlx29", "gb00b03mlx29","1u0197800237", "n10000289965",np.nan],
......:
    name = ["ABN Amro","Robeco","Royal Dutch Shell","Royal Dutch Shell","AAB 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]:

290 Chapter 1. What’s New
• *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 = DataFrame({
   ....:     'Branch': ['A A A A A B'].split(),
   ....:     'Buyer': ['Carl Mark Carl Carl Joe Joe'].split(),
   ....:     'Quantity': [1, 3, 5, 1, 8, 1],
   ....:     'Date': [datetime.datetime(2013,11,1,13,0),
              datetime.datetime(2013,9,1,13,5),
              datetime.datetime(2013,10,1,20,0),
              datetime.datetime(2013,10,2,10,0),
              datetime.datetime(2013,11,1,20,0),
              datetime.datetime(2013,10,2,10,0)],
   ....:     'PayDay': [datetime.datetime(2013,10,4,0,0),
                datetime.datetime(2013,9,5,10,0),
                datetime.datetime(2013,9,5,10,0),
                datetime.datetime(2013,9,5,10,0),
                datetime.datetime(2013,9,5,10,0),
                datetime.datetime(2013,9,5,10,0)]})

In [74]: df
```

```
Out[74]:
          Branch  Buyer     Date       PayDay
   0          A  Carl  2013-11-01 13:00:00 2013-10-04 00:00:00 1
   1          A  Mark  2013-09-01 13:05:00 2013-10-15 13:05:00 3
   2          A  Carl  2013-10-01 20:00:00 2013-09-05 20:00:00 5
   3          A  Carl  2013-10-02 10:00:00 2013-11-02 10:00:00 1
   4          A   Joe  2013-11-01 20:00:00 2013-10-07 20:00:00 8
   5          B   Joe  2013-10-02 10:00:00 2013-09-05 10:00:00 1

In [75]: pivot_table(df, index=Grouper(freq='M', key='Date'),
   ....:     columns=Grouper(freq='M', key='PayDay'),
   ....:     values='Quantity', aggfunc=np.sum)

```

• 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)

```
In [76]: prng = period_range('2013-01-01 09:00', periods=100, freq='H')

In [77]: ps = Series(np.random.randn(len(prng)), index=prng)

In [78]: ps
```

```
Out[78]:
2013-01-01 09:00    0.015696
2013-01-01 10:00    -2.242685
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>11:00</td>
<td>1.150036</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>12:00</td>
<td>0.991946</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>13:00</td>
<td>0.953324</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>14:00</td>
<td>-2.021255</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>15:00</td>
<td>-0.334077</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013-01-05</td>
<td>06:00</td>
<td>0.566534</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>07:00</td>
<td>0.503592</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>08:00</td>
<td>0.285296</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>09:00</td>
<td>0.484288</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>10:00</td>
<td>1.363482</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>11:00</td>
<td>-0.781105</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>12:00</td>
<td>-0.468018</td>
</tr>
</tbody>
</table>

Freq: H, Length: 100, dtype: float64

```
In [79]: ps['2013-01-02']

Out[79]:
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>00:00</td>
<td>0.553439</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>01:00</td>
<td>1.318152</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>02:00</td>
<td>-0.469305</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>03:00</td>
<td>0.675554</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>04:00</td>
<td>-1.817027</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>05:00</td>
<td>-0.183109</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>06:00</td>
<td>1.058969</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013-01-02</td>
<td>17:00</td>
<td>0.076200</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>18:00</td>
<td>-0.566446</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>19:00</td>
<td>0.036142</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>20:00</td>
<td>-2.074978</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>21:00</td>
<td>0.247792</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>22:00</td>
<td>-0.897157</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>23:00</td>
<td>-0.136795</td>
</tr>
</tbody>
</table>

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)
1.19.12 Performance

- Performance improvement when converting `DatetimeIndex` to floating ordinals using `DatetimeConverter` (GH6636)
- Performance improvement for `DataFrame.shift` (GH5609)
- Performance improvement in indexing into a multi-indexed 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).

1.19.13 Experimental

There are no experimental changes in 0.14.0

1.19.14 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 zeroes (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 multi-index 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 (GH6338).
- Perf issue in concatting 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 MultiIndexes 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 re-ordering 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 metacharacters were being treated as regexes 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 with numpy < 1.7.2 when reading long strings from HDFStore (GH6166)
• 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 adding 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 datetlike 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 (GH6878)
• 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 whitespace (GH3374)
• Bug in C parser with delim_whitespace=True and \r-delimited lines
• Bug in python parser with explicit multi-index 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 nans 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, multi-index 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 multi-index (GH7199)
• Fix a bug where invalid eval/query operations would blow the stack (GH5198)

1.20 v0.13.1 (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:

```python
In [1]: df = DataFrame(dict(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 = DataFrame(dict(A = np.array(['foo','bar','bah','foo','bar'])))
In [5]: df.loc[0,'A'] = np.nan
In [6]: df
Out[6]:
   A
A 0  NaN
    1 bar
    2 bah
    3 foo
    4 bar

1.20.1 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)

In [7]: max_info_rows = pd.get_option('max_info_rows')
In [8]: df = DataFrame(dict(A = np.random.randn(10),
                        B = np.random.randn(10),
                        C = 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):
A 7 non-null float64
B 10 non-null float64
C 7 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 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):
A 7 non-null float64
B 10 non-null float64
C 7 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 bytes

- Add `show_dimensions` display option for the new DataFrame repr to control whether the dimensions print.
In [14]: df = DataFrame([[1, 2], [3, 4]])

In [15]: pd.set_option('show_dimensions', False)

In [16]: df
Out[16]:
0 1
0 1 2
1 3 4

In [17]: pd.set_option('show_dimensions', True)

In [18]: df
Out[18]:
0 1
0 1 2
1 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:

<table>
<thead>
<tr>
<th>age</th>
<th>today</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>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 = DataFrame([ Timestamp('20010101'),
                     Timestamp('20040601') ], columns=['age'])

In [20]: df['today'] = 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]

### 1.20.2 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:

In [23]: s = Series(['a', 'a|b', np.nan, 'a|c'])

In [24]: s.str.get_dummies(sep='|')
• 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.

```
In [25]: df = DataFrame({'col': ['foo', 0, np.nan]})
In [26]: df2 = DataFrame({'col':[np.nan, 0, 'foo']}, index=[2,1,0])
In [27]: df.equals(df2)
Out[27]: False
```

```
In [28]: df.equals(df2.sort_index())
Out[28]: True
```

```
In [29]: import pandas.core.common as com
In [30]: com.array_equivalent(np.array([0, np.nan]), np.array([0, np.nan]))
Out[30]: True
In [31]: np.array_equal(np.array([0, np.nan]), np.array([0, np.nan]))
```

• `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:

```
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)
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.

```
In [35]: empty.apply(applied_func, reduce=True)
Out[35]:
```

```
1.20.3 Prior Version Deprecations/Changes

There are no announced changes in 0.13 or prior that are taking effect as of 0.13.1

1.20.4 Deprecations

There are no deprecations of prior behavior in 0.13.1

1.20.5 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 @lexical 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
do = 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):

  ```python
  In [37]: shades = ['light', 'dark']
  In [38]: colors = ['red', 'green', 'blue']
  In [39]: MultiIndex.from_product([shades, colors], names=['shade', 'color'])
  Out[39]:
  MultiIndex(levels=[['dark', 'light'], ['blue', 'green', 'red']],
             labels=[[1, 1, 1, 0, 0, 0], [2, 1, 0, 2, 1, 0]],
             names=['shade', 'color'])
  ```

- Panel `apply()` will work on non-ufuncs. See the docs.
In [40]: import pandas.util.testing as tm

In [41]: panel = tm.makePanel(5)

In [42]: panel
Out[42]:
<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 [43]: panel['ItemA']

```
   A    B    C    D
2000-01-03  0.694103  1.893534 -1.735349 -0.850346
2000-01-04  0.678630  0.639633  1.210384  1.176812
2000-01-05  0.239556 -0.962029  0.797435 -0.524336
2000-01-06  0.151227 -2.085266  0.379811  0.700908
2000-01-07  0.816127  1.930247  0.702562  0.984188
```
[5 rows x 4 columns]

Specifying an apply that operates on a Series (to return a single element)

In [44]: panel.apply(lambda x: x.dtype, axis='items')
Out[44]:
          A      B      C      D
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

[5 rows x 4 columns]

A similar reduction type operation

In [45]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[45]:
     ItemA  ItemB  ItemC
A 2.579643  3.062757  0.379252
B 1.416120 -1.960855  0.923558
C 0.595222 -1.079772 -3.118269
D 1.487226 -0.734611 -1.979310

[4 rows x 3 columns]

This is equivalent to

In [46]: panel.sum('major_axis')
Out[46]:
     ItemA  ItemB  ItemC
A 2.579643  3.062757  0.379252
B 1.416120 -1.960855  0.923558
C 0.595222 -1.079772 -3.118269
D 1.487226 -0.734611 -1.979310
A transformation operation that returns a Panel, but is computing the z-score across the major_axis

```python
In [47]: result = panel.apply(
    ....:     lambda x: (x-x.mean())/x.std(),
    ....:     axis='major_axis')
    ....:

In [48]: result
Out[48]:
<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: A to D

In [49]: result['ItemA']

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>0.595800</td>
<td>0.907552</td>
<td>-1.556260</td>
<td>-1.244875</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.544058</td>
<td>0.200868</td>
<td>0.915883</td>
<td>0.953747</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.924165</td>
<td>-0.701810</td>
<td>0.569325</td>
<td>-0.891290</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-1.219530</td>
<td>-1.334852</td>
<td>-0.418654</td>
<td>0.437589</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.003837</td>
<td>0.928242</td>
<td>0.489705</td>
<td>0.744830</td>
</tr>
</tbody>
</table>
```

- Panel `apply()` operating on cross-sectional slabs. (GH1148)

```python
In [50]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T

In [51]: result = panel.apply(f, axis = ['items','major_axis'])

In [52]: result
Out[52]:
<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: ItemA to ItemC

In [53]: result.loc[:, :, 'ItemA']

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>0.331409</td>
<td>1.071034</td>
<td>-0.914540</td>
<td>-0.510587</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.741017</td>
<td>-0.118794</td>
<td>0.383277</td>
<td>0.537212</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.065042</td>
<td>-0.767353</td>
<td>0.655436</td>
<td>0.069467</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.027932</td>
<td>-0.569477</td>
<td>0.908202</td>
<td>0.610585</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.116434</td>
<td>1.133591</td>
<td>0.871287</td>
<td>1.004064</td>
</tr>
</tbody>
</table>
```

This is equivalent to the following
1.20.6 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)

1.20.7 Experimental

There are no experimental changes in 0.13.1

1.20.8 Bug Fixes

See V0.13.1 Bug Fixes for an extensive list of bugs that have been fixed in 0.13.1.

See the full release notes or issue tracker on GitHub for a complete list of all API changes, Enhancements and Bug Fixes.
1.21 v0.13.0 (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**

1.21.1 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
index.levels = [[1, 2, 3, 4], [1, 2, 4, 4]]

# now, you use the set_levels or set_labels methods
index = 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 = index.set_names(['bob', 'cranberry'])

# and all methods take an inplace kwarg - but return None
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]: Series(arr) // Series(arr2)
Out[6]:
0 0
1 0
2 1
3 4
dtype: int64
```

**True Division**

```python
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.

```python
if df:
    ....
df1 and df2
s1 and s2
```
Added the `.bool()` method to NDFrame objects to facilitate evaluating of single-element boolean Series:

```
In [1]: Series([True]).bool()
Out[1]: True

In [2]: Series([False]).bool()
    Out[2]: False

In [3]: DataFrame([[True]]).bool()
    Out[3]: True

In [4]: 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 = 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)
  ...                # A value is trying to be set on a copy of a slice from a DataFrame.
    SettingWithCopyWarning:   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)
1.21.2 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 then None. This activates truncated display ("...") of long sequences in various places. (GH3391)

1.21.3 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.

1.21.4 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 = Series([1,2,3])

In [11]: s
Out[11]:
0   1
1   2
2   3
Length: 3, dtype: int64


In [13]: s
Out[13]:
0   1.0
1   2.0
2   3.0
5   5.0
Length: 4, dtype: float64
```
In [14]: dfi = DataFrame(np.arange(6).reshape(3,2),
                columns=['A','B'])

In [15]: dfi
Out[15]:
    A  B
0  0  1
1  2  3
2  4  5
[3 rows x 2 columns]

This would previously KeyError

In [16]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [17]: dfi
Out[17]:
    A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
[3 rows x 3 columns]

This is like an append operation.

In [18]: dfi.loc[3] = 5

In [19]: dfi
Out[19]:
    A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
[4 rows x 3 columns]

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 [21]: p
Out[21]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 2 (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 B

In [22]: p.loc[ :, :, 'C'] = 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']

<table>
<thead>
<tr>
<th></th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-12</td>
<td>30.0</td>
<td>32.0</td>
</tr>
<tr>
<td>2001-01-13</td>
<td>30.0</td>
<td>32.0</td>
</tr>
<tr>
<td>2001-01-14</td>
<td>30.0</td>
<td>32.0</td>
</tr>
<tr>
<td>2001-01-15</td>
<td>30.0</td>
<td>32.0</td>
</tr>
</tbody>
</table>

[4 rows x 2 columns]

1.21.5 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 [25]: index = Index([1.5, 2, 3, 4.5, 5])

  In [26]: index
  Out[26]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

  In [27]: s = Series(range(5), index=index)

  In [28]: s
  Out[28]:
  1.5 0
  2.0 1
  3.0 2
  4.5 3
  5.0 4
  Length: 5, 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 [29]: s[3]
  Out[29]: 2

  In [30]: s.loc[3]
  Out[30]: 2

  The only positional indexing is via iloc

  In [31]: s.iloc[3]
  Out[31]: 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 [32]: s[2:4]
Out[32]:
2.0 1
3.0 2
Length: 2, dtype: int64

In [33]: s.loc[2:4]
Out[33]:
2.0 1
3.0 2
Length: 2, dtype: int64

In [34]: s.iloc[2:4]
Out[34]:
3.0 2
4.5 3
Length: 2, dtype: int64

In float indexes, slicing using floats are allowed.

In [35]: s[2.1:4.6]
Out[35]:
3.0 2
4.5 3
Length: 2, dtype: int64

In [36]: s.loc[2.1:4.6]
Out[36]:
3.0 2
4.5 3
Length: 2, dtype: int64

In 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`.

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

1.21.6 HDFStore API Changes

- Query Format Changes. A much more string-like query format is now supported. See the docs.
In [37]: path = 'test.h5'

In [38]: dfq = DataFrame(randn(10,4),
    ....:     columns=list('ABCD'),
    ....:     index=date_range('20130101',periods=10))

In [39]: dfq.to_hdf(path,'dfq',format='table',data_columns=True)

Use boolean expressions, with in-line function evaluation.

In [40]: read_hdf(path,'dfq',
    ....:     where="index>Timestamp('20130104') & columns=['A', 'B']")

Out[40]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.057633</td>
<td>-0.791489</td>
</tr>
<tr>
<td>1</td>
<td>1.910759</td>
<td>0.787965</td>
</tr>
<tr>
<td>2</td>
<td>1.043945</td>
<td>2.107785</td>
</tr>
<tr>
<td>3</td>
<td>0.749185</td>
<td>-0.675521</td>
</tr>
<tr>
<td>4</td>
<td>-0.276646</td>
<td>1.924533</td>
</tr>
<tr>
<td>5</td>
<td>0.226363</td>
<td>-2.078618</td>
</tr>
</tbody>
</table>

[6 rows x 2 columns]

Use an inline column reference

In [41]: read_hdf(path,'dfq',
    ....:     where="A>0 or C>0")

Out[41]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.414505</td>
<td>-1.425795</td>
<td>0.209395</td>
<td>-0.592886</td>
</tr>
<tr>
<td>1</td>
<td>-1.473116</td>
<td>-0.896581</td>
<td>1.104352</td>
<td>-0.431550</td>
</tr>
<tr>
<td>2</td>
<td>-0.161137</td>
<td>0.889157</td>
<td>0.288377</td>
<td>-1.051539</td>
</tr>
<tr>
<td>3</td>
<td>-0.319561</td>
<td>-0.619993</td>
<td>0.156998</td>
<td>-0.571455</td>
</tr>
<tr>
<td>4</td>
<td>1.057633</td>
<td>-0.791489</td>
<td>-0.524627</td>
<td>0.071878</td>
</tr>
<tr>
<td>5</td>
<td>1.910759</td>
<td>0.787965</td>
<td>0.513082</td>
<td>-0.546416</td>
</tr>
<tr>
<td>6</td>
<td>1.043945</td>
<td>2.107785</td>
<td>1.459927</td>
<td>1.015405</td>
</tr>
<tr>
<td>7</td>
<td>0.749185</td>
<td>-0.675521</td>
<td>0.440266</td>
<td>0.688972</td>
</tr>
<tr>
<td>8</td>
<td>-0.276646</td>
<td>1.924533</td>
<td>0.411204</td>
<td>0.890765</td>
</tr>
<tr>
<td>9</td>
<td>0.226363</td>
<td>-2.078618</td>
<td>-0.387886</td>
<td>-0.087107</td>
</tr>
</tbody>
</table>

[10 rows x 4 columns]

- 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 [47]: 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 [48]: path = 'test.h5'

In [49]: df = DataFrame(randn(10,2))

In [50]: store1 = HDFStore(path)

In [51]: store2 = HDFStore(path)

In [52]: store1.append('df',df)

In [53]: store2.append('df2',df)

In [54]: store1
Out[54]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [55]: store2

Out[55]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [56]: store1.close()

In [57]: store2
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [58]: store2.close()

In [59]: store2
Out[59]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

• 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 thru store creation arguments; can be used to support in-memory stores

1.21.7 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').

1.21.8 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 outtype. Returns an array of column-keyed dictionaries. (GH4936)

• NaN handing in get_dummies (GH4446) with dummy_na

```python
# previously, nan was erroneously counted as 2 here
# now it is not counted at all
In [60]: get_dummies([1, 2, np.nan])
Out[60]:
   1.0  2.0
0   1   0
1   0   1
```
2 0 0
[3 rows x 2 columns]

# unless requested
In [61]: get_dummies([1, 2, np.nan], dummy_na=True)

Out[61]:
    0  1  2  NaN
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
[3 rows x 3 columns]

• timedelta64[ns] operations. See the docs.

**Warning**: Most of these operations require numpy >= 1.7

Using the new top-level to_timedelta, you can convert a scalar or array from the standard timedelta format (produced by to_csv) into a timedelta type (np.timedelta64 in nanoseconds).

In [62]: to_timedelta('1 days 06:05:01.00003')
Out[62]: Timedelta('1 days 06:05:01.000030')

In [63]: to_timedelta('15.5us')
Out[63]: Timedelta('0 days 00:00:00.000015')

In [64]: to_timedelta(['1 days 06:05:01.00003','15.5us','nan'])
Out[64]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT], dtype='timedelta64[ns]', freq=None)

In [65]: to_timedelta(np.arange(5),unit='s')
Out[65]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'], dtype='timedelta64[ns]', freq=None)

In [66]: to_timedelta(np.arange(5),unit='d')
Out[66]: 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 dtype Series. This is frequency conversion. See the docs for the docs.

In [67]: from datetime import timedelta
In [68]: td = Series(date_range('20130101',periods=4))-Series(date_range('20121201',periods=4))
In [69]: td[2] += np.timedelta64(timedelta(minutes=5,seconds=3))
In [70]: td[3] = np.nan
In [71]: td
Out [71]:
0  31 days 00:00:00
1  31 days 00:00:00
2  31 days 00:05:03
3    NaT
Length: 4, dtype: timedelta64[ns]

# to days
In [72]: td / np.timedelta64(1,'D')
Out [72]:
0  31.000000
1  31.000000
2  31.003507
3    NaN
Length: 4, dtype: float64

In [73]: td.astype('timedelta64[D]')
Out [73]:
0  31.0
1  31.0
2  31.0
3    NaN
Length: 4, dtype: float64

# to seconds
In [74]: td / np.timedelta64(1,'s')
Out [74]:
0  2678400.0
1  2678400.0
2  2678703.0
3    NaN
Length: 4, dtype: float64

In [75]: td.astype('timedelta64[s]')
Out [75]:
0  2678400.0
1  2678400.0
2  2678703.0
3    NaN
Length: 4, dtype: float64

Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series

In [76]: td * -1
Out [76]:
0  -31 days +00:00:00
1  -31 days +00:00:00
2  -32 days +23:54:57
3    NaN
Length: 4, dtype: timedelta64[ns]

In [77]: td * Series([1,2,3,4])
Absolute DateOffset objects can act equivalently to timedeltas

```
In [78]: from pandas import offsets

In [79]: td + offsets.Minute(5) + offsets.Milli(5)
Out[79]:
0  31 days 00:05:00.005000
1  31 days 00:05:00.005000
2  31 days 00:10:03.005000
3       NaT
Length: 4, dtype: timedelta64[ns]
```

Fillna is now supported for timedeltas

```
In [80]: td.fillna(0)
Out[80]:
0  31 days 00:00:00
1  31 days 00:00:00
2  31 days 00:05:03
3       0 days
Length: 4, dtype: timedelta64[ns]
```

You can do numeric reduction operations on timedeltas.

```
In [82]: td.mean()
Out[82]: Timedelta('31 days 00:01:41')

In [83]: td.quantile(.1)
```

- `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.

```
In [84]: Series(['a1', 'b2', 'c3']).str.extract('^[ab](\d)')
Out[84]:
0  1
1  2
```
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 [85]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
Out[85]:
   0 1
0  a 1
1  b 2
2  NaN NaN
```

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 [86]: Series(['a1', 'b2', 'c3']).str.extract(
    ....:   '(?P<letter>[ab])(?P<digit>\d)')
Out[86]:
   letter digit
0      a   1
1      b   2
2  NaN    NaN
```

and optional groups can also be used.

```python
In [87]: Series(['a1', 'b2', '3']).str.extract(
    ....:   '(?P<letter>[ab])?(?P<digit>\d)')
Out[87]:
   letter digit
0      a   1
1      b   2
2  NaN   3
```

- `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 [88]: date_range('2013-01-01', periods=5, freq='5N')
Out[88]:
dtype='datetime64[ns]', freq='5N')

or with frequency as offset
In [89]: date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
Out[89]:
dtype='datetime64[ns]', freq='5N')

Timestamps can be modified in the nanosecond range
In [90]: t = Timestamp('20130101 09:01:02')
In [91]: t + pd.tseries.offsets.Nano(123)
Out[91]:
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 [92]: dfi = DataFrame({'A': [1, 2, 3, 4], 'B': ['a', 'b', 'f', 'n']})
In [93]: dfi
Out[93]:
   A  B
0  1  a
1  2  b
2  3  f
3  4  n
[4 rows x 2 columns]
In [94]: other = DataFrame({'A': [1, 3, 3, 7], 'B': ['e', 'f', 'f', 'e']})
In [95]: mask = dfi.isin(other)
In [96]: mask
Out[96]:
   A  B
0  True False
1  False False
2  True True
3  False False
[4 rows x 2 columns]
In [97]: dfi[mask.any(1)]
• 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 [98]: df = DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
                   ....:   'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})

In [99]: df.interpolate()
Out[99]:
    A    B
0   1.0  0.25
1  2.1  1.50
2  3.4  2.75
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40
```

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:

```python
In [100]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])

In [101]: ser.interpolate(limit=2)
Out[101]:
0    1.0
Name: 0, dtype: float64
```
1 3.0
2 5.0
3 7.0
4 NaN
5 11.0
Length: 5, dtype: float64

- Added `wide_to_long` panel data convenience function. See the docs.

```
In [102]: np.random.seed(123)
In [103]: 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 [104]: df['id'] = df.index
In [105]: df
Out[105]:
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
[3 rows x 6 columns]
In [106]: wide_to_long(df, ['A', 'B'], i='id', j='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
[6 rows x 3 columns]
```

- `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)

### 1.21.9 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,
For more details, see the 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,

```
In [111]: df = DataFrame(randn(10, 2), columns=['a', 'b'])
In [112]: df.eval('a + b')
```

```
Out[112]:
    0   -0.685204
    1     1.589745
    2     0.325441
    3    -1.784153
    4    -0.432893
    5     0.171850
    6     1.895919
    7     3.065587
    8    -0.092759
    9     1.391365
Length: 10, 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,

```
In [113]: n = 20
In [114]: df = DataFrame(np.random.randint(n, size=(n, 3)), columns=['a', 'b', 'c'])
In [115]: df.query('a < b < c')
```

```
Out[115]:
    a  b  c
11  1  5  8
15  8 16 19
[2 rows x 3 columns]
```

selects all the rows of `df` where `a < b < c` evaluates to `True`. For more details see the 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.

In [116]: df = DataFrame(np.random.rand(5,2),columns=list('AB'))
In [117]: df.to_msgpack('foo.msg')
In [118]: pd.read_msgpack('foo.msg')

Out[118]:
      A       B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns]
In [119]: s = Series(np.random.rand(5),index=date_range('20130101',periods=5))
In [120]: pd.to_msgpack('foo.msg', df, s)
In [121]: pd.read_msgpack('foo.msg')

Out[121]:
[      A       B]
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns], 2013-01-01 0.022321
2013-01-02 0.227025
2013-01-03 0.383282
2013-01-04 0.193225
2013-01-05 0.110977
Freq: D, Length: 5, dtype: float64]

You can pass iterator=True to iterator over the unpacked results

In [122]: for o in pd.read_msgpack('foo.msg', iterator=True):
.....: print(o)
.....:
      A       B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns]
2013-01-01 0.022321
2013-01-02 0.227025
2013-01-03 0.383282
2013-01-04 0.193225
2013-01-05 0.110977
Freq: D, Length: 5, dtype: float64
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,"n
        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 = pandas.concat([df2.min(), df2.mean(), df2.max()],
        axis=1,keys=['Min Tem', 'Mean Temp', 'Max Temp'])

The resulting DataFrame is:

```
A patch is scheduled for the week of 10/14/13.

### 1.21.10 Internal Refactoring

In 0.13.0 there is a major refactoring primarily to subclass `Series` from `NDFrame`, which is the base class currently for `DataFrame` and `Panel`, to unify methods and behaviors. Series formerly subclassed directly from `ndarray`. (GH4080, GH3862, GH816)

**Warning:** There are two potential incompatibilities from < 0.13.0

- Using certain numpy functions would previously return a `Series` if passed a `Series` as an argument. This seems only to affect `np.ones_like`, `np.empty_like`, `np.diff` and `np.where`. These now return ndarrays.

```python
In [123]: s = Series([1,2,3,4])

Numpy Usage

In [124]: np.ones_like(s)
Out[124]: array([1, 1, 1, 1])

In [125]: np.diff(s)
Out[125]: array([1, 1, 1])

In [126]: np.where(s>1,s,np.nan)
Out[126]: array([nan, 2., 3., 4.])

Pandonic Usage

In [127]: Series(1,index=s.index)
Out[127]:
0  1
1  1
2  1
3  1
Length: 4, dtype: int64

In [128]: s.diff()
Out[128]:
0  NaN
1  1.0
2  1.0
3  1.0
Length: 4, dtype: float64

In [129]: s.where(s>1)
Out[129]:
0  NaN
1  2.0
2  3.0
3  4.0
Length: 4, 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,pop`
    • `__iter__,keys,contains,len,neg,invert`
    • `convert_objects,as_blocks,as_matrix,values`
    • `__getstate__,__setstate__(compat remains in frame/panel)`
    • `__getattribute__,__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` (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 [130]: s = Series([1,2,3],index=list('abc'))
In [131]: s.b
Out [131]: 2
In [132]: s.a = 5
In [133]: s
Out [133]:
    a  5
    b  2
    c  3
Length: 3, dtype: int64
```

1.21.11 Bug Fixes

See `V0.13.0 Bug Fixes` for an extensive list of bugs that have been fixed in 0.13.0.

See the full release notes or issue tracker on GitHub for a complete list of all API changes, Enhancements and Bug Fixes.

1.22 v0.12.0 (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 multi-indexes 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.
### 1.22.1 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 corrects a numpy bug that treats integer and float dtypes differently.

```python
In [1]: p = 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

[3 rows x 2 columns]
In [3]: p % p
   first  second
 0  0.0     NaN
 1  0.0     NaN
 2  0.0     0.0

[3 rows x 2 columns]
In [4]: p / p
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
˓→

0
1
2

first
1.0
1.0
1.0

second
NaN
NaN
1.0

[3 rows x 2 columns]

In [5]: p / 0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
˓→

0
1
2

first
inf
inf
inf

second
NaN
NaN
inf

[3 rows x 2 columns]

• 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 [6]: df2 = DataFrame([{"val1": 1, "val2" : 20}, {"val1":1, "val2": 19},
...:
{"val1":1, "val2": 27}, {"val1":1, "val2": 12}])
...:
In [7]: def func(dataf):
...:
return dataf["val2"]
...:

- dataf["val2"].mean()

# squeezing the result frame to a series (because we have unique groups)
In [8]: df2.groupby("val1", squeeze=True).apply(func)
Out[8]:
0
0.5
1
-0.5
2
7.5
3
-7.5
Name: 1, Length: 4, dtype: float64
# no squeezing (the default, and behavior in 0.10.1)
In [9]: df2.groupby("val1").apply(func)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
˓→

val2
val1
1

0

1

0.5 -0.5

2

3

7.5 -7.5

[1 rows x 4 columns]

• 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 plently of alternatives. This preserves the iloc API to be purely positional
based.
In [10]: df = DataFrame(lrange(5), list('ABCDE'), columns=['a'])

1.22. v0.12.0 (July 24, 2013)

331


In [11]: mask = (df.a % 2 == 0)

In [12]: mask
Out[12]:
A   True
B  False
C   True
D  False
E   True
Name: a, Length: 5, dtype: bool

# this is what you should use
In [13]: df.loc[mask]

    a
A  0
C  2
E  4

[3 rows x 1 columns]

# this will work as well
In [14]: df.iloc[mask.values]

    a
A  0
C  2
E  4

[3 rows x 1 columns]

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
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

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 astypes 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 baseclass 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)

1.22.2 I/O 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

    In [15]: df = DataFrame({'a': range(3), 'b': list('abc')})

    In [16]: print(df)
    
    a   b
    0   0 a
    1   1 b
    2   2 c

    [3 rows x 2 columns]

    In [17]: html = df.to_html()
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 [20]: from pandas.util.testing import makeCustomDataframe as mkdf
In [21]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)
In [22]: df.to_csv('mi.csv')
In [23]: 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 [24]: pd.read_csv('mi.csv', header=[0,1,2,3], index_col=[0,1])
```
• 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

```
In [25]: path = 'store_iterator.h5'
In [26]: DataFrame(randn(10,2)).to_hdf(path,'df',table=True)
In [27]: for df in 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
      3 -0.626968 -0.875772
      4 -0.930687 -0.218983
      5  0.949965 -0.442354
      6 -0.402985  1.111358
      7 -0.241527 -0.670477
      8  0.049355  0.632633
      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

### 1.22.3 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

```
In [25]: df = DataFrame({'a': list('ab..'), 'b': [1, 2, 3, 4]})
In [26]: df.replace(regex=r'\s*\.\s*', value=np.nan)
Out[26]:
   a    b
0  1.5 -1.2
1  1.5 -1.2
```
to replace all occurrences of the string '. ' with zero or more instances of surrounding whitespace with NaN.

Regular string replacement still works as expected. For example, you can do

```
In [27]: df.replace('.', np.nan)
Out[27]:
 a  b
0   a  1
1   b  2
2  NaN  3
3  NaN  4
```

This replaces 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:

```
In [28]: pd.get_option('a.b')
Out[28]: 2

In [29]: pd.get_option('b.c')
Out[29]: 3

In [30]: pd.set_option('a.b', 1, 'b.c', 4)

In [31]: pd.get_option('a.b')
Out[31]: 1

In [32]: pd.get_option('b.c')
Out[32]: 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.

```
In [33]: sf = Series([1, 1, 2, 3, 3, 3])

In [34]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[34]:
3  3
4  3
5  3
Length: 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.
```python
In [35]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [36]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[36]:
   A  B
0  2  b
1  3  b
2  4  b
3  5  b
[4 rows x 2 columns]
```

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.

```python
In [37]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[37]:
   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
[8 rows x 2 columns]
```

- 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 date-time.min and datetime.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

### 1.22.4 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.

```python
In [38]: from pandas.tseries.offsets import CustomBusinessDay
doctest:6: FutureWarning: elementwise comparison failed; returning scalar instead
In [39]: from datetime import datetime
# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [40]: weekmask_egypt = 'Sun Mon Tue Wed Thu'
# They also observe International Workers' Day so let's
```
# add that for a couple of years
In [41]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]

In [42]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [43]: dt = datetime(2013, 4, 30)

In [44]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00

In [45]: dts = date_range(dt, periods=5, freq=bday_egypt)

In [46]: print(Series(dts.weekday, dts).map(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, Length: 5, dtype: object

## 1.22.5 Bug Fixes

- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have 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,
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 a `AttributeConflictWarning` 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.itertuples()` now works with frames with duplicate column names (GH3873)

1.22. v0.12.0 (July 24, 2013)
pandas: powerful Python data analysis toolkit, Release 0.21.0

- Bug in non-unique indexing via loc (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.

1.23 v0.11.0 (April 22, 2013)

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*

### 1.23.1 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 indicies 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*.

### 1.23.2 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.
### 1.23.3 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 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')

In [2]: df1
Out[2]:
     A
0  1.392665
1 -0.123497
2 -0.402761
3 -0.246604
4 -0.288433
5 -0.763434
6   2.069526
7 -1.203569

[8 rows x 1 columns]

In [3]: df1.dtypes

Out[3]:
   A
float32
Length: 1, dtype: object

In [4]: df2 = DataFrame(dict( A = Series(randn(8),dtype='float16'),
                         ...: B = Series(randn(8)),
                         ...: C = Series(randn(8),dtype='uint8'))

In [5]: df2
Out[5]:
     A         B         C
0  0.591797 -0.038605       0
1  0.841309 -0.460478       1
2 -0.500977 -0.310458       0
3 -0.816406  0.866493  254
4 -0.207031  0.245972       0
5 -0.664062  0.319442       1
6  0.580566  1.378512       1
7 -0.965820  0.292502  255

[8 rows x 3 columns]

In [6]: df2.dtypes

Out[6]:
    A    B    C
float16 float64 uint8
Length: 3, dtype: object

# here you get some upcasting
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```
1.23.4 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')
```

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
Length: 5, dtype: object
```

# same, but specific dtype conversion
```
In [15]: df3['D'] = df3['D'].astype('float16')
```
Forcing Date coercion (and setting NaT when not datelike)

In [18]: from datetime import datetime

In [19]: s = Series([datetime(2001,1,1,0,0), 'foo', 1.0, 1,
   ....:                Timestamp('20010104'), '20010105'],dtype='O')
   ....:

In [20]: s.convert_objects(convert_dates='coerce')

Out[20]:
0 2001-01-01
1 NaT
2 NaT
3 NaT
4 2001-01-04
5 2001-01-05
Length: 6, dtype: datetime64[ns]

### 1.23.5 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]: DataFrame([[1,2]],columns=['a']).dtypes
Out[21]:
a    int64
Length: 1, dtype: object

In [22]: DataFrame({'a' : [1,2] }).dtypes

In [23]: DataFrame({'a' : 1 }, index=range(2)).dtypes


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  1  0  0  1  1
1  1  0  1  1  1
2  0  0  0  1  1
3 -1  0 254 1  1
4  0  0  0  1  1
5 -1  0  1  1  1
6  2  1  1  1  1
7 -2  0  255 1  1

[8 rows x 5 columns]

In [27]: dfi.dtypes

A  int32
B  int32
C  int32
D  int64
E  int32
Length: 5, dtype: object

In [28]: casted = dfi[dfi>0]

In [29]: casted
Out[29]:
   A  B  C  D  E
0  1.0 NaN NaN  1  1
1 NaN NaN  1.0  1  1
2 NaN NaN NaN  1  1
3 NaN NaN 254.0  1  1
4 NaN NaN NaN NaN  1
5 NaN NaN  1.0  1  1
6  2.0  1.0  1.0  1  1
7 NaN NaN 255.0  1  1

[8 rows x 5 columns]

In [30]: casted.dtypes

A  float64
B  float64
C  float64
D  int64
E  int32
Length: 5, dtype: object
```
While float dtypes are unchanged.

```python
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
Length: 5, dtype: object

In [34]: casted = df4[df4>0]

In [35]: casted
Out [35]:
   A  B  C  D  E
0  1.98  NaN  NaN  1.0  1
1  0.72  NaN  1.0  1.0  1
2  NaN  NaN  NaN  1.0  1
3  NaN  0.87  254.0  1.0  1
4  NaN  0.25  NaN  1.0  1
5  NaN  0.32  1.0  1.0  1
6  2.65  1.38  1.0  1.0  1
7  NaN  0.29  255.0  1.0  1
[8 rows x 5 columns]

In [36]: casted.dtypes

1.23.6 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)

```python
In [37]: df = DataFrame(randn(6,2),date_range('20010102',periods=6),columns=['A','B'])

In [38]: df['timestamp'] = Timestamp('20010103')

In [39]: df
Out [39]:
   A      B  timestamp
0  0.22  2.63  2001-01-02
1  0.43 -1.17  2001-01-02
2 -0.62 -0.90  2001-01-02
3  2.33 -1.67  2001-01-02
4 -0.68 -3.39  2001-01-02
5 -0.00  0.70  2001-01-02

In [40]: df.dtypes
Out [40]:
A    float64
B    float64
C    float64
D    float16
E     int32
Length: 5, dtype: object
```
```
2001-01-04 -0.270630 -1.685677 2001-01-03
2001-01-05 -0.440747 -0.115070 2001-01-03
2001-01-06 -0.632102 -0.585977 2001-01-03
2001-01-07 -1.444787 -0.201135 2001-01-03

[6 rows x 3 columns]

# datetime64[ns] out of the box
In [40]: df.get_dtype_counts()

Out[40]:
       datetime64[ns]  1
float64              2
Length: 2, dtype: int64

# use the traditional nan, which is mapped to NaT internally
In [41]: df.loc[df.index[2:4], ['A','timestamp']] = np.nan

In [42]: df
Out[42]:
   A         B       timestamp
0  2001-01-02  1.023958  0.660103 2001-01-03
1  2001-01-03  1.236475 -2.170629 2001-01-03
2  2001-01-04 NaN    1.685677    NaT
3  2001-01-05 NaN    0.115070    NaT
4  2001-01-06 -0.632102 -0.585977 2001-01-03
5  2001-01-07 -1.444787 -0.201135 2001-01-03

[6 rows x 3 columns]
```

Astype conversion on `datetime64[ns]` to `object`, implicitly converts NaT to np.nan

```
In [43]: import datetime

In [44]: s = Series([datetime.datetime(2001, 1, 2, 0, 0) for i in range(3)])

In [45]: s.dtype
Out[45]: dtype('<M8[ns]')

In [46]: s[1] = np.nan

In [47]: s
Out[47]:
0  2001-01-02
1  NaT
2  2001-01-02
Length: 3, dtype: datetime64[ns]

In [48]: s.astype('M8[ns]')
Out[48]:
0  2001-01-02
1  NaT
2  2001-01-02
Length: 3, dtype: datetime64[ns]

In [49]: s = s.astype('O')

In [50]: s
Out[50]:
0  2001-01-02 00:00:00
1  NaT
```
1.23.7 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

1.23.8 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
    ```python
    In [52]: df = DataFrame(dict(A=lrange(5), B=lrange(5)))
    In [53]: df.to_hdf('store.h5','table',append=True)
    In [54]: read_hdf('store.h5', 'table', where = ['index>2'])
    Out[54]:
    A  B
    3  3  3
    4  4  4
    [2 rows x 2 columns]
    ```
  - 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 sup-
    port 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 [58]: df = DataFrame(dict(A = ts))

In [59]: df['2001']
Out[59]:
   A
2001-10-31  0.663256
2001-11-30  0.079126
2001-12-31  0.587699
[3 rows x 1 columns]

• Squeeze to possibly remove length 1 dimensions from an object.

In [60]: p = Panel(randn(3,4,4),items=['ItemA','ItemB','ItemC'],
   ....:   major_axis=date_range('20010102',periods=4),
   ....:   minor_axis=['A','B','C','D'])

In [61]: p.reindex(items=['ItemA']).squeeze()

In [62]: p.reindex(items=['ItemA'],minor=['B']).squeeze()

• 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
pandas: powerful Python data analysis toolkit, Release 0.21.0

- 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.

1.24 v0.10.1 (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 `inplace` option has been reverted and deprecation warnings added.

1.24.1 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)
1.24.2 New features

- MySQL support for database (contribution from Dan Allan)

1.24.3 HDFStore

You may need to upgrade your existing data files. Please visit the compatibility 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 = HDFStore('store.h5')
In [2]: df = DataFrame(randn(8, 3), index=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
0  1.885136 -0.183873  2.550850   foo     cool
1  0.180759 -1.117089  0.061462   foo     cool
2 -0.294467 -0.591411 -0.876691   foo     cool
3  3.127110  1.451130  0.045152   foo     cool
4 -0.242846  1.195819  1.533294   NaN     cool
5  0.820521 -0.281201  1.651561   NaN     cool
6 -0.034086  0.252394 -0.498772   foo     cool
7 -2.290958 -1.601262 -0.256718    bar     cool
[8 rows x 5 columns]

# 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]:
       A         B         C    string  string2
2000-01-04  3.127110  1.451130  0.045152   foo     cool
2000-01-07  -0.034086  0.252394 -0.498772   foo     cool
[2 rows x 5 columns]

# this is in-memory version of this type of selection
In [10]: df[(df.B > 0) & (df.string == 'foo')]
   ...
   A         B         C    string  string2
2000-01-04  3.127110  1.451130  0.045152   foo     cool
2000-01-07  -0.034086  0.252394 -0.498772   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'] = 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
0  2000-01-01 1.885136 -0.183873  foo    cool  2001-01-02
1  2000-01-02 0.180759 -1.117089  foo    cool  2001-01-02
2  2000-01-03 -0.294467 0.591411 -0.876691  foo    cool  2001-01-02
3  2000-01-04  NaN     NaN     NaN     NaN    NaN     NaN
4  2000-01-05 -0.242846 1.195819 1.533294  NaN     cool  2001-01-02
5  2000-01-06 0.820521 -0.281201 1.651561  NaN     cool  2001-01-02
6  2000-01-07 -0.034086 0.252394 -0.498772  foo    cool  2001-01-02
7  2000-01-08 -2.290958 -1.601262 -0.256718  bar    cool  2001-01-02

[8 rows x 6 columns]
```

```
In [17]: df_mixed1.get_dtype_counts()
...

   →
datetime64[ns]     1
float64           3
object            2
Length: 3, 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)

```
In [18]: store.select('df', columns=['A', 'B'])
Out[18]:
   A    B
0  1.885136 -0.183873
1  0.180759 -1.117089
2 -0.294467  0.591411
3  3.127110  1.451130
4 -0.242846  1.195819
5  0.820521 -0.281201
6 -0.034086  0.252394
7 -2.290958 -1.601262
```
HDFStore now serializes multi-index dataframes when appending tables.

```python
In [19]: index = 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 = DataFrame(np.random.randn(10, 3), index=index,
                      columns=['A', 'B', 'C'])

In [21]: df
Out[21]:
     A       B       C
foo bar
foo one  0.239369  0.174122 -1.131794
two -1.948006  0.980347 -0.674429
three -0.361633 -0.761218  1.768215
bar one  0.152288 -0.862613 -0.210968
two -0.859278  1.498195  0.462413
baz two -0.647604  1.511487 -0.727189
three -0.342928 -0.007364  1.427674
qux one  0.104020  2.052171 -1.230963
two -0.019240 -1.713238  0.838912
three -0.637855  0.215109 -1.515362

[10 rows x 3 columns]

In [22]: store.append('mi', df)

In [23]: store.select('mi')
Out[23]:
     A       B       C
foo bar
foo one  0.239369  0.174122 -1.131794
two -1.948006  0.980347 -0.674429
three -0.361633 -0.761218  1.768215
bar one  0.152288 -0.862613 -0.210968
two -0.859278  1.498195  0.462413
baz two -0.647604  1.511487 -0.727189
three -0.342928 -0.007364  1.427674
qux one  0.104020  2.052171 -1.230963
two -0.019240 -1.713238  0.838912
three -0.637855  0.215109 -1.515362

[10 rows x 3 columns]

# the levels are automatically included as data columns
In [24]: store.select('mi', "foo=='bar'")
     A       B       C
foo bar
```
bar one  0.152288 -0.862613 -0.210968
    two -0.859278  1.498195  0.462413
[2 rows x 3 columns]

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.

```python
In [25]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
      columns=['A', 'B', 'C', 'D', 'E', 'F'])
      ....:
      ....:
In [26]: df_mt['foo'] = 'bar'

# you can also create the tables individually
In [27]: store.append_to_multiple({ 'df1_mt' : ['A','B'], 'df2_mt' : None }, df_mt, ˓
   selector = 'df1_mt')
In [28]: store
Out[28]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# indiviual tables were created
In [29]: store.select('df1_mt')

       A  B
2000-01-01 1.586924 -0.447974
2000-01-02 -0.102206  0.870302
2000-01-03  1.249874  1.458210
2000-01-04 -0.616293  1.458210
2000-01-05 -0.431163  0.016640
2000-01-06  0.800353 -0.451572
2000-01-07  1.239198  0.185437
2000-01-08 -0.040863  0.290110
[8 rows x 2 columns]
In [30]: store.select('df2_mt')

       C  D  E  F  foo
2000-01-01  0.630925 -0.071659 -1.277640  bar
2000-01-02  1.275280  0.199212  1.673018  bar
2000-01-03 -0.710542  0.825392  1.557329  bar
2000-01-04  0.132104  0.580923 -0.128750  bar
2000-01-05  0.904578 -1.645852 -0.888714  bar
2000-01-06  0.831767  0.228760  0.932498 -2.200069  bar
2000-01-07 -0.540770 -0.370038  1.298390  bar
2000-01-08 -0.096145  1.717830 -0.462446 -0.112019  bar
[8 rows x 5 columns]

# as a multiple
In [31]: store.select_as_multiple(['df1_mt','df2_mt'], where = [ 'A>0','B>0' ], ˓
   selector = 'df1_mt')
    
```
## 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 automatically 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 expectedrows that PyTables will expect. 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 (-series) and inversion (~series) 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 businessday 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.

1.25 v0.10.0 (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.

1.25.1 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

1.25.2 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', periods=6))
```
In [3]: df
Out[3]:
    0    1    2    3
2000-01-01 -0.134024 -0.205969 1.348944 -1.198246
2000-01-02 -1.626124 0.982041 0.059493 -0.460111
2000-01-03 -1.565401 -0.025706 0.942864 2.502156
2000-01-04 -0.302741 0.261551 -0.066342 0.897097
2000-01-05  0.268766 -1.225092 0.582752 -1.490764
2000-01-06 -0.639757 -0.952750 -0.892402 0.505987
[6 rows x 4 columns]

# deprecated now
In [4]: df - df[0]

[6 rows x 10 columns]

# Change your code to
In [5]: df.sub(df[0], axis=0) # align on axis 0 (rows)

[6 rows x 4 columns]
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 = Series(np.arange(len(dates)), index=dates)
In [3]: series
Out[3]:
2000-01-01 00:00:00    0
2000-01-01 04:00:00    1
2000-01-01 08:00:00    2
2000-01-01 12:00:00    3
2000-01-01 16:00:00    4
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
```
• 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:

```python
In [6]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])
```

```python
In [7]: pd.isnull(s)
```

```python
Out[7]:
0  False
1  False
2  False
3  False
Length: 4, dtype: bool
```

```python
In [8]: s.fillna(0)
```

```python
Out[8]:
0  1.500000
1  inf
2  3.400000
3  -inf
Length: 4, dtype: float64
```

```python
In [9]: pd.set_option('use_inf_as_null', True)
```

```python
In [10]: pd.isnull(s)
```

```python
Out[10]:
0  False
1  True
2  False
3  True
Length: 4, dtype: bool
```

```python
In [11]: s.fillna(0)
```

```python
Out[11]:
0  1.5
1  0.0
2  3.4
3  0.0
Length: 4, dtype: float64
```

```python
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 \( X0, X1, \ldots \)) can be reproduced by specifying `prefix='X'`:

```python
In [6]: data= 'a,b,c\n1,Yes,2\n3,No,4'
```
In [7]: `print(data)
   a,b,c
   1,Yes,2
   3,No,4`

In [8]: `pd.read_csv(StringIO(data), header=None)`
\------
Out[8]:
  0 1 2
  0 a b c
  1 1 Yes 2
  2 3 No 4
[3 rows x 3 columns]

In [9]: `pd.read_csv(StringIO(data), header=None, prefix='X')`
\------
\------
Out[9]:
   X0  X1  X2
   0 a  b  c
   1 1  Yes 2
   2 3  No 4
[3 rows x 3 columns]

• 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 [10]: `print(data)
   a,b,c
   1,Yes,2
   3,No,4`

In [11]: `pd.read_csv(StringIO(data))`  
\------
Out[11]:
  a b c
  0 1 Yes 2
  1 3 No 4
[2 rows x 3 columns]

In [12]: `pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])`  
\------
\------
Out[12]:
   a  b  c
   0 1  True 2
   1 3  False 4
[2 rows x 3 columns]

• 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 [13]: `s = Series([np.nan, 1., 2., np.nan, 4])`

In [14]: `s`
Out[14]:
0   NaN
1    1.0
2    2.0
3   NaN
4    4.0
Length: 5, dtype: float64

In [15]: s.fillna(0)
Out[15]:
0    0.0
1    1.0
2    2.0
3    0.0
4    4.0
Length: 5, dtype: float64

In [16]: s.fillna(method='pad')
Out[16]:
0   NaN
1    1.0
2    2.0
3    2.0
4    4.0
Length: 5, dtype: float64

Convenience methods ffill and bfill have been added:

In [17]: s.ffill()
Out[17]:
0   NaN
1    1.0
2    2.0
3    2.0
4    4.0
Length: 5, dtype: float64

• 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

In [18]: def f(x):
   ....:     return Series([ x, x**2 ], index = ['x', 'x^2'])
   ....:

In [19]: s = Series(np.random.rand(5))

In [20]: s
Out[20]:
0    0.717478
1    0.815199
2    0.452478
3    0.848385
4    0.235477
Length: 5, dtype: float64

In [21]: s.apply(f)
New API functions for working with pandas options (GH2097):

- `get_option` / `set_option` - get/set the value of an option. Partial names are accepted.
- `reset_option` - reset one or more options to their default value. Partial names are accepted.
- `describe_option` - print a description of one or more options. When called with no arguments, print all registered options.

Note: `set_printoptions` / `reset_printoptions` are now deprecated (but functioning), the print options now live under “display.XYZ”. For example:

```python
In [22]: get_option("display.max_rows")
Out[22]: 15
```

- `to_string()` methods now always return unicode strings (GH2224).

### 1.25.3 New features

#### 1.25.4 Wide DataFrame Printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```python
In [23]: wide_frame = DataFrame(randn(5, 16))

In [24]: wide_frame
```

```
  0    1     2     3     4     5     6
0 -0.681624 0.191356 1.180274 -0.834179 0.703043 0.166568 -0.583599
1  0.441522 -0.316864 -0.017062 1.570114 -0.360875 -0.880096 0.235532
2 -0.412451 -0.462580 0.422194 0.288403 -0.487393 -0.777639 0.055865
3 -0.277255 1.331263 0.585174 -0.568825 -0.719412 1.191340 -0.456362
4 -1.642511 0.432560 1.218080 -0.564705 -0.581790 0.286071 0.048725
   7    8     9    10    11    12    13
0 -1.201796 -1.422811 -0.882554 1.209871 -0.941235 0.863067 -0.336232
1  0.207232 -1.983857 -1.702547 -1.621234 -0.906840 1.014601 -0.475108
2  1.383381 0.085638 0.246392 0.965887 0.246354 -0.727728 -0.094414
3  0.089931 0.776079 0.752889 -1.195795 -1.425911 -0.548829 0.774225
4  1.002440 1.276582 0.054399 0.241963 -0.471786 0.314510 -0.059986
   14   15
0 -0.976847 0.033862
1 -0.358944 1.262942
2 -0.276854 0.158399
3  0.740501 1.510263
```
4 -2.069319 -1.115104  
[5 rows x 16 columns]  

The old behavior of printing out summary information can be achieved via the ‘expand_frame_repr’ print option:

```python
In [25]: pd.set_option('expand_frame_repr', False)
```

```python
In [26]: wide_frame
```

```python
Out[26]:
       0       1       2       3       4       5       6       7
0 -0.681624  0.191356  1.180274 -0.834179  0.703043  0.166568 -0.583599 -1.201796
   -422811 -0.882554  1.209871 -0.941235  0.863067 -0.336232 -0.976847  0.033862
1  0.441522 -0.316864 -0.017062  1.570114 -0.360875 -0.880096  0.235532  0.207232
   -983857 -1.702547 -1.621234 -0.906840  1.014601 -0.475108 -0.358944  1.262942
2 -0.412451 -0.462580  0.422194  0.288403 -0.487393 -0.776393  0.055865  1.383381
   -085638  0.246392  0.965887  0.246354 -0.727728 -0.094414 -0.276854  0.158399
3 -0.277255  1.331263  0.585174 -0.568825 -0.719412  1.191340 -0.456362  0.089931
   -776079  0.752889 -1.195795 -1.425911 -0.548829  0.774225  0.740501  1.510263
4 -1.642511  0.432560  1.218080 -0.564705 -0.581790  0.286071  0.048725  1.002440
   -276582  0.054399  0.241963 -0.471786  0.314510 -0.059986 -2.069319 -1.115104
```

[5 rows x 16 columns]  

The width of each line can be changed via ‘line_width’ (80 by default):

```python
In [27]: pd.set_option('line_width', 40)
```

```python
In [28]: wide_frame
```

```python
Out[28]:
     0       1       2       3       4       5       6       7
0 -0.681624  0.191356  1.180274 -0.834179  0.703043  0.166568 -0.583599
   -422811 -0.882554  1.209871 -0.941235  0.863067 -0.336232 -0.976847
1  0.441522 -0.316864 -0.017062  1.570114 -0.360875 -0.880096  0.235532  0.207232
   -983857 -1.702547 -1.621234 -0.906840  1.014601 -0.475108 -0.358944
2 -0.412451 -0.462580  0.422194  0.288403 -0.487393 -0.776393  0.055865
   -085638  0.246392  0.965887  0.246354 -0.727728 -0.094414
3 -0.277255  1.331263  0.585174 -0.568825 -0.719412  1.191340 -0.456362
   -776079  0.752889 -1.195795 -1.425911 -0.548829  0.774225
4 -1.642511  0.432560  1.218080 -0.564705 -0.581790  0.286071  0.048725
   -276582  0.054399  0.241963 -0.471786  0.314510 -0.059986
```

[5 rows x 16 columns]  

1.25. v0.10.0 (December 17, 2012)
1.25.5 Updated PyTables Support

Docs for PyTables Table format & several enhancements to the api. Here is a taste of what to expect.

In [29]: store = HDFStore('store.h5')

In [30]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
....:           columns=['A', 'B', 'C'])

In [31]: df
Out[31]:
          A         B         C
2000-01-01 -0.369325 -1.502617 -0.376280
2000-01-02  0.511936 -0.116412 -0.625256
2000-01-03 -0.550627  1.261433 -0.552429
2000-01-04  1.695803 -1.025917 -0.910942
2000-01-05  0.426805 -0.131749  0.432600
2000-01-06  0.044671 -0.341265  1.844536
2000-01-07 -2.036047  0.000830 -0.955697
2000-01-08 -0.898872 -0.725411  0.059904

[8 rows x 3 columns]

# appending data frames
In [32]: df1 = df[0:4]

In [33]: df2 = df[4:]

In [34]: store.append('df', df1)

In [35]: store.append('df', df2)

In [36]: store
Out[36]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# selecting the entire store
In [37]: store.select('df')
Out[37]:
          A         B         C
2000-01-01 -0.369325 -1.502617 -0.376280
2000-01-02  0.511936 -0.116412 -0.625256
2000-01-03 -0.550627  1.261433 -0.552429
2000-01-04  1.695803 -1.025917 -0.910942
2000-01-05  0.426805 -0.131749  0.432600
2000-01-06  0.044671 -0.341265  1.844536
2000-01-07 -2.036047  0.000830 -0.955697
2000-01-08 -0.898872 -0.725411  0.059904

[8 rows x 3 columns]

In [38]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
....:         major_axis=date_range('1/1/2000', periods=5),
....:         minor_axis=['A', 'B', 'C', 'D'])
In [39]: wp
Out[39]:
<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 [40]: store.append('wp', wp)

# selecting via A QUERY
In [41]: store.select('wp', "major_axis>20000102 and minor_axis=['A','B']")
Out[41]:
<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 [42]: store.remove('wp', "major_axis>20000103")

In [43]: store.select('wp')
Out[43]:
<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 [44]: del store['df']
In [45]: store
Out[45]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

Enhancements

• added ability to hierarchical keys

In [46]: store.put('foo/bar/bah', df)
In [47]: store.append('food/orange', df)
In [48]: store.append('food/apple', df)
In [49]: store
Out[49]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# remove all nodes under this level
In [50]: store.remove('food')

In [51]: store
Out[51]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

• added mixed-dtype support!

In [52]: df['string'] = 'string'
In [53]: df['int'] = 1
In [54]: store.append('df', df)
In [55]: df1 = store.select('df')
In [56]: df1
Out[56]:
   A     B    C      string    int
0 2000-01-01 -0.369325 -1.502617 string 1
1 2000-01-02  0.511936 -0.116412 string 1
2 2000-01-03 -0.550627  1.261433 string 1
3 2000-01-04  1.695803 -1.025917 string 1
4 2000-01-05  0.426805 -0.131749 string 1
5 2000-01-06  0.044671 -0.341265 string 1
6 2000-01-07 -2.036047  0.000830 string 1
7 2000-01-08 -0.898872 -0.725411 string 1

[8 rows x 5 columns]
In [57]: df1.get_dtype_counts()

Out[57]:
˓→
float64 3
int64   1
object  1
Length: 3, dtype: int64

• performance improvements on table writing
• support for arbitrarily indexed dimensions
• SparseSeries now has a density property (GH2384)
• enable Series.str.strip/lstrip/rstrip 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.

### 1.25.6 N Dimensional Panels (Experimental)

Adding experimental support for Panel4D and factory functions to create n-dimensional named panels. Docs for NDim. Here is a taste of what to expect.

```
In [58]: p4d = Panel4D(randn(2, 2, 5, 4),
   ....:   labels=['Label1','Label2'],
   ....:   items=['Item1', 'Item2'],
   ....:   major_axis=date_range('1/1/2000', periods=5),
   ....:   minor_axis=['A', 'B', 'C', 'D'])
   ....:
In [59]: p4d
```

```
<class 'pandas.core.panelnd.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.

### 1.26 v0.9.1 (November 14, 2012)

This is a bugfix 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.

#### 1.26.1 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)

```
In [2]: df = DataFrame(np.random.randint(0, 2, (6, 3)), columns=['A', 'B', 'C'])
In [3]: df.sort(['A', 'B'], ascending=[1, 0])
```

```
  A  B  C
3  0  1  1
```
**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 = DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])
In [3]: df.rank()
Out[3]:
        A    B    C
0  3.00  1.0  3.0
1  2.00  2.0  1.0
2  NaN   NaN  NaN
3  NaN   NaN  NaN
4  NaN   NaN  NaN
5  1.00  3.0  2.0

[6 rows x 3 columns]
In [4]: df.rank(na_option='top')

        A    B    C
0  6.00  4.0  6.0
1  5.00  5.0  4.0
2  2.00  2.0  2.0
3  2.00  2.0  2.0
4  2.00  2.0  2.0
5  4.00  6.0  5.0

[6 rows x 3 columns]
In [5]: df.rank(na_option='bottom')

       A    B    C
0  3.00  1.0  3.0
1  2.00  2.0  1.0
2  5.00  5.0  5.0
3  5.00  5.0  5.0
4  5.00  5.0  5.0
5  1.00  3.0  2.0

[6 rows x 3 columns]
```

- 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.
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 \(\text{NaN}\). This is accomplished via the new method `DataFrame.where`. In addition, `where` takes an optional `other` argument for replacement.
Furthermore, \texttt{where} now aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analagous to partial setting via \texttt{.ix} (but on the contents rather than the axis labels)

\begin{verbatim}
In [12]: df2 = df.copy()
In [13]: df2[ df2[1:4] > 0 ] = 3
In [14]: df2
Out[14]:
   A        B        C
0  1.744738  -0.356939  0.092791
1  3.000000   3.000000  3.000000
2  3.000000  -0.404023  -1.115882
3  3.000000   3.000000  -1.775758
4  1.303175   0.025683  -1.795489

[5 rows x 3 columns]
\end{verbatim}

\textit{DataFrame.mask} is the inverse boolean operation of \texttt{where}.

\begin{verbatim}
In [15]: df.mask(df<=0)
Out[15]:
   A        B        C
0  1.744738 Nan   0.092791
1  1.226237  1.909179  0.195946
2  0.481559 Nan   Nan
3  2.093925  0.010808  1.775758
4  1.303175  0.025683 Nan

[5 rows x 3 columns]
\end{verbatim}

- Enable referencing of Excel columns by their column names (GH1936)

\begin{verbatim}
In [16]: xl = 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
2000-01-03  0.980269  3.685731 -0.364217  1.159738
2000-01-04  1.047916  0.041232  0.161812  0.212549
2000-01-05  0.498581  0.731168  0.537677  1.346270
2000-01-06  1.120202  1.567621  0.003641  0.675253
2000-01-07 -0.487094  0.571455  1.116139  0.103469
2000-01-10  0.836649  0.246462  0.588543  1.062782
2000-01-11 -0.157161  1.340307  1.195778  1.097007

[7 rows x 4 columns]
\end{verbatim}
• 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)

1.26.2 API changes

• Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

```
In [1]: prng = period_range('2012Q1', periods=2, freq='Q')
In [2]: s = 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
     2012-06 NaN
Freq: M, dtype: float64
```

• Period.end_time now returns the last nanosecond in the time interval (GH2124, GH2125, GH1764)

```
In [18]: p = 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]: data = 'A,B,C
00001,001,5
00002,002,6'
In [21]: read_csv(StringIO(data), converters={'A' : lambda x: x.strip()})
Out[21]:
     A  B  C
0  00001  1  5
1  00002  2  6
[2 rows x 3 columns]
```

See the full release notes or issue tracker on GitHub for a complete list.

1.27 v0.9.0 (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.

### 1.27.1 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)

### 1.27.2 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:

```
In [1]: data = '0,0,1
   :1,1,0
   :0,1,0'
In [2]: df = read_csv(StringIO(data), header=None)
In [3]: df
Out[3]:
     0 1 2
0   0 0 1
1   1 1 0
2   0 1 0
3 rows x 3 columns
```

- 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 [4]: s1 = Series([1, 2, 3])
In [5]: s1
Out[5]:
0    1
1    2
2    3
Length: 3, dtype: int64
In [6]: s2 = Series(s1, index=['foo', 'bar', 'baz'])
In [7]: s2
```
See the full release notes or issue tracker on GitHub for a complete list.

1.28  v0.8.1 (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.

1.28.1 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)
1.28.2 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

1.29 v0.8.0 (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.

1.29.1 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)

1.29.2 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 codebases 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.

1.29.3 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 codebase. 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 localtime. Time zone conversions are therefore essentially free. User needs to know very little about pytz library now; only time zone names as 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

### 1.29.4 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 `:ref'dayfirst <io.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`

1.29.5 New plotting methods

`Series.plot` now supports a `secondary_y` option:

```python
In [1]: plt.figure()
Out[1]: <matplotlib.figure.Figure at 0x1298a9a58>

In [2]: fx['FR'].plot(style='g')
                   \                                   
                   ----> AxesSubplot at 0x12b046ba8

Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0x129c83f98>

In [3]: fx['IT'].plot(style='k--', secondary_y=True)
                   \                                   
                   ----> AxesSubplot at 0x129c83f98
```

Vytautas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, ’kde’ is a new option:

```python
In [4]: s = Series(np.concatenate((np.random.randn(1000),
                             ...
                             np.random.randn(1000) + 0.5 + 3)))
```
In [5]: plt.figure()
Out[5]: <matplotlib.figure.Figure at 0x1207ff9e8>

In [6]: s.hist(normed=True, alpha=0.2)
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1298a9668>

In [7]: s.plot(kind='kde')
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1298a9668>

See the plotting page for much more.

### 1.29.6 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.

### 1.29.7 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 [8]: import datetime
In [9]: rng = date_range('1/1/2000', periods=10)
In [10]: rng[5]
Out[10]: Timestamp('2000-01-06 00:00:00', freq='D')
In [11]: isinstance(rng[5], datetime.datetime)
Out[11]: True
In [12]: rng_asarray = np.asarray(rng)
In [13]: scalar_val = rng_asarray[5]
In [14]: type(scalar_val)
Out[14]: 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 `asobject` property of `DatetimeIndex` produces an array of `Timestamp` objects:

```python
In [15]: stamp_array = rng.asobject
In [16]: stamp_array
```
To get an array of proper `datetime.datetime` objects, use the `to_pydatetime` method:

```
In [18]: dt_array = rng.to_pydatetime()

In [19]: dt_array
Out[19]:
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)
```

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 [21]: rng = date_range('1/1/2000', periods=10)

In [22]: rng
Out[22]:
DateTimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-09', '2000-01-10'],
             dtype='datetime64[ns]', freq='D')

In [23]: np.asarray(rng)

In [24]: converted = np.asarray(rng, dtype=object)

In [25]: converted[5]
```
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.

1.30 v.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.

1.30.1 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)
```
- Add `stacked` argument to Series and DataFrame's `plot` method for stacked bar plots.

```python
df.plot(kind='bar', stacked=True)
```
Add log x and y scaling options to DataFrame.plot and Series.plot
Add kurt methods to Series and DataFrame for computing kurtosis

1.30.2 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 = Series(['Steve', np.nan, 'Joe'])
In [2]: series == 'Steve'
Out[2]:
0    True
1   False
2   False
Length: 3, dtype: bool

In [3]: series != 'Steve'
    \\n\n\nOut[3]:
0   False
1    True
2    True
Length: 3, dtype: bool
```

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()]
Out[5]:
0    Steve
Length: 1, dtype: object
```
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.

### 1.30.3 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:

```python
In [6]: df = 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 [7]: df
Out[7]:
   A  B          C          D
0  foo  one  1.075059 -0.449141
1  bar  one  0.785676  1.443014
2  foo  two  0.958157  0.612324
3  bar  three 1.477773 -0.178818
4  foo  two -1.006023  0.612324
5  bar  two -1.506997 -0.550981
6  foo  one  1.218042 -2.043335
7  foo  three -0.565878  0.753539
[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  0.252151  1.562274 -1.506997 -0.360661  0.785676  1.131724
  foo   5.0  0.335871  1.039915 -1.006023 -0.565878  0.958157  1.075059

  max
A
  bar  1.477773
  foo  1.218042

[2 rows x 8 columns]

In [10]: grouped.apply(lambda x: x.sort_values()[-2:]) # top 2 values

A
  bar  1  0.785676
       3  1.477773
  foo  0  1.075059
       6  1.218042
Name: C, Length: 4, dtype: float64
```
1.31 v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

1.31.1 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)

1.31.2 Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)

1.32 v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

1.32.1 New features

- Add to_clipboard function to pandas namespace for writing objects to the system clipboard (GH774)
- Add iteritems 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

1.32.2 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)
1.33 v0.7.0 (February 9, 2012)

1.33.1 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 = DataFrame(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  -0.409608  0.539495  0.163276  0.051646
std    1.397779  0.968808  0.874489  0.719651
min   -2.539411 -0.737206 -1.202276 -1.050435
25%   -1.202202  0.021308 -0.368812 -0.383608
50%   -0.384480  0.306124  0.211431  0.165586
75%    0.186280  1.024039  0.730744  0.494457
max    2.524998  2.533114  1.334428  1.147396
[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.

### 1.33.2 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:

```python
In [3]: s = Series(randn(10), index=range(0, 20, 2))
In [4]: s
Out [4]:
0 -0.543429
2  1.425447
4 -0.408795
6 -1.489348
8 -1.166408
10 -0.481205
12 -0.810355
14 -0.985491
16 -0.336246
18 -0.629058
Length: 10, dtype: float64

In [5]: s[0]
\→ -0.54342898765020686

In [6]: s[2]
\→ 1.4254474252163707

In [7]: s[4]
\→ -0.40879476802408349
```
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 = DataFrame(randn(8, 4), index=range(0, 16, 2))
```

```
In [4]: df
Out[4]:
    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   0.61941
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>

### 1.33.3 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 = Series(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
a   1.628992
```

![Image](image.png)

This change also has the same impact on DataFrame:

```
In [3]: df = DataFrame(randn(8, 4), index=range(0, 16, 2))
```

```
In [4]: df
Out[4]:
    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   0.61941
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>

### 1.33.3 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 = Series(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
a   1.628992
```
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
  c -0.539890
  e  0.073308
  g -1.182230
  k -0.243550
  m -0.276183
dtype: float64
In [6]: s2.ix['b':'h']
Out[6]:
  c -0.539890
  e  0.073308
  g -1.182230
dtype: float64
```

### 1.33.4 Changes to Series [] operator

As an 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 = Series(randn(6), index=list('acegkm'))
In [9]: s
Out[9]:
a -0.297788
c  0.499769
e  0.810531
g  0.414649
k -1.551478
m  1.012459
Length: 6, dtype: float64
In [10]: s[['m', 'a', 'c', 'e']]
   m  1.012459
   a -0.297788
c  0.499769
e  0.810531
Length: 4, dtype: float64
```
In the case of integer indexes, the behavior will be exactly as before (shadowing \texttt{ndarray}): 

In [13]: s = Series(randn(6), index=range(0, 12, 2))  
In [14]: s[[4, 0, 2]]  
Out[14]:  
\begin{verbatim}
   4   0.928877
  0   1.171752
  2   0.026488
Length: 3, dtype: float64
\end{verbatim} 

In [15]: s[1:5]  
Out[15]:  
\begin{verbatim}
   2   0.026488
  4   0.928877
  6  -1.264991
  8   0.419449
Length: 4, dtype: float64
\end{verbatim} 

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use \texttt{ix}.

1.33.5 Other API Changes

- The deprecated \texttt{LongPanel} class has been completely removed

- If \texttt{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 \texttt{df[col].sort()} instead of the side-effect free method \texttt{df[col].order()} (GH316)

- Miscellaneous renames and deprecations which will (harmlessly) raise \texttt{FutureWarning}

- \texttt{drop} added as an optional parameter to \texttt{DataFrame.reset_index} (GH699)

1.33.6 Performance improvements

- \textit{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 \texttt{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
dataarray 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 compati-
• 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)

1.34 v.0.6.1 (December 13, 2011)

1.34.1 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*

**1.34.2 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)

**1.35 v.0.6.0 (November 25, 2011)**

**1.35.1 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)

1.35.2 Performance Enhancements

• VBENCH Cythonized cache_readonly, resulting in substantial micro-performance enhancements throughout the codebase (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)

1.36 v.0.5.0 (October 24, 2011)

1.36.1 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)
• **Added** & 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)
• TODO: DOCS ABOUT TAKE METHODS

### 1.36.2 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

### 1.37 v.0.4.3 through v0.4.1 (September 25 - October 9, 2011)

#### 1.37.1 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_dtypes_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)

### 1.37.2 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
The easiest way for the majority of users 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, various Linux distributions, or a development version are also provided.

2.1 Python version support

Officially Python 2.7, 3.5, and 3.6.

2.2 Installing pandas

2.2.1 Installing pandas 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, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

After running a simple 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.

An additional advantage of installing with Anaconda is that you don’t require admin rights to install it, it will install in the user’s home directory, and this also makes it trivial to delete Anaconda at a later date (just delete that folder).

2.2.2 Installing pandas 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 (these are analogous to a virtualenv but they also allow you to specify precisely which Python version to install also). 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 require any packages that are available to pip but not conda, simply install pip, and use pip to install these packages:

```
conda install pip
pip install django
```

### 2.2.3 Installing from PyPI

pandas can be installed via pip from PyPI.

```
pip install pandas
```

This will likely require the installation of a number of dependencies, including NumPy, will require a compiler to compile required bits of code, and can take a few minutes to complete.

### 2.2.4 Installing using your Linux distribution’s package manager.

The commands in this table will install pandas for Python 2 from your distribution. To install pandas for Python 3 you may need to use the package **python3-pandas**.
### 2.2.5 Installing from source

See the [contributing documentation](https://pandas.pydata.org/pandas-docs/stable/getting_started/00_introduction.html) for complete instructions on building from the git source tree. Further, see [creating a development environment](https://pandas.pydata.org/pandas-docs/stable/getting_started/00_introduction.html) if you wish to create a pandas development environment.

### 2.2.6 Running the test suite

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

```sh
>>> import pandas as pd
>>> pd.test()
```

```
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-packages\pandas
====================================================================
platform win32 -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: C:\Users\TP\Documents\Python\pandasdev\pandas, inifile: setup.cfg
collected 12145 items / 3 skipped

..........................................................S......
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S
...............................................................S

====================================================================
12130 passed, 12 skipped in 368.339 seconds
```

### 2.3 Dependencies

- `setuptools`

---

**Distribution** | **Status** | **Download / Repository Link** | **Install method**
--- | --- | --- | ---
Debian | stable | official Debian repository | `sudo apt-get install python-pandas`
Debian & Ubuntu (latest packages) | unstable | NeuroDebian | `sudo apt-get install python-pandas`
Ubuntu | stable | official Ubuntu repository | `sudo apt-get install python-pandas`
Ubuntu (daily builds) | unstable | PythonXY PPA; activate by: `sudo add-apt-repository ppa:pythonxy/pythonxy-devel` & `sudo apt-get update` | `sudo apt-get install python-pandas`
OpenSuse | stable | OpenSuse Repository | `zypper in python-pandas`
Fedora | stable | official Fedora repository | `dnf install python-pandas`
Centos/RHEL | stable | EPEL repository | `yum install python-pandas`
• **NumPy**: 1.9.0 or higher
• **python-dateutil**: 1.5 or higher
• **pytz**: Needed for time zone support

### 2.3.1 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 2.4.6 or higher.
- **bottleneck**: for accelerating certain types of nan evaluations. bottleneck uses specialized cython routines to achieve large speedups. If installed, must be Version 1.0.0 or higher.

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

### 2.3.2 Optional Dependencies

- **Cython**: Only necessary to build development version. Version 0.23 or higher.
- **SciPy**: miscellaneous statistical functions, Version 0.14.0 or higher
- **xarray**: pandas like handling for > 2 dims, needed for converting Panels to xarray objects. Version 0.7.0 or higher is recommended.
- **PyTables**: necessary for HDF5-based storage. Version 3.0.0 or higher required, Version 3.2.1 or higher highly recommended.
- **Feather Format**: necessary for feather-based storage, version 0.3.1 or higher.
- **Apache Parquet**, either pyarrow (>= 0.4.1) or fastparquet (>= 0.0.6) for parquet-based storage. The snappy and brotli are available for compression support.
- **SQLAlchemy**: for SQL database support. Version 0.8.1 or higher recommended. Besides SQLAlchemy, you also need a database specific driver. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs. Some common drivers are:
  - **psycopg2**: for PostgreSQL
  - **pymysql**: for MySQL.
  - **SQLite**: for SQLite, this is included in Python’s standard library by default.
- **matplotlib**: for plotting, Version 1.4.3 or higher.
- **For Excel I/O**:
  - **xlr/ xlwt**: Excel reading (xlr) and writing (xlwt)
  - **openpyxl**: openpyxl version 1.6.1 or higher (but lower than 2.0.0), or version 2.2 or higher, for writing .xlsx files (xlr >= 0.9.0)
  - **XlsxWriter**: Alternative Excel writer
- **Jinja2**: Template engine for conditional HTML formatting.
- **s3fs**: necessary for Amazon S3 access (s3fs >= 0.0.7).
- **blosc**: for msgpack compression using blosc
• One of PyQt4, PySide, pygtk, xsel, or xclip: necessary to use `read_clipboard()`. Most package managers on Linux distributions will have xclip and/or xsel immediately available for installation.

• For Google BigQuery I/O - see here

• Backports.lzma: Only for Python 2, for writing to and/or reading from an xz compressed DataFrame in CSV; Python 3 support is built into the standard library.

• One of the following combinations of libraries is needed to use the top-level `read_html()` function:
  – BeautifulSoup4 and html5lib (Any recent version of html5lib is okay.)
  – BeautifulSoup4 and lxml
  – BeautifulSoup4 and html5lib and lxml
  – Only lxml, although see HTML Table Parsing 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.
  – You may need to install an older version of BeautifulSoup4: Versions 4.2.1, 4.1.3 and 4.0.2 have been confirmed for 64 and 32-bit Ubuntu/Debian

Note:
  – if you’re on a system with apt-get you can do

    ```bash
    sudo apt-get build-dep python-lxml
    ```

    to get the necessary dependencies for installation of lxml. This will prevent further headaches down the line.

Note: Without the optional dependencies, many useful features will not work. Hence, it is highly recommended that you install these. A packaged distribution like Anaconda, or Enthought Canopy may be worth considering.
pandas: powerful Python data analysis toolkit, Release 0.21.0
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
  - Creating a development environment
  - Creating a Windows development environment
  - Making changes
- Contributing to the documentation
  - About the pandas documentation
  - How to build the pandas documentation
    - Requirements
    - Building the documentation
    - Building master branch documentation
- Contributing to the code base
  - Code standards
    - C (cpplint)
    - Python (PEP8)
    - Backwards Compatibility
  - Testing With Continuous Integration
  - Test-driven development/code writing
    - Writing tests
    - Transitioning to pytest
* Using pytest
  - Running the test suite
  - Running the performance test suite
  - Documenting your code

• Contributing your changes to pandas
  - Committing your code
  - Combining commits
  - Pushing your changes
  - Review your code
  - Finally, make the pull request
  - Delete your merged branch (optional)

3.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome.

If you are simply looking to start working with the pandas codebase, navigate to the GitHub “issues” tab and start looking through interesting issues. There are a number of issues listed under Docs and Difficulty Novice where you could start out.

Or maybe through using pandas you have an idea of your own or are looking for something in the documentation and thinking ‘this can be improved’...you can do something about it!

Feel free to ask questions on the mailing list or on Gitter.

3.2 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. Because many versions of pandas are supported, knowing version information will also identify improvements made since previous versions. Trying the bug-producing code out on the master 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.

### 3.3 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.

#### 3.3.1 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’s documentation.
- Matthew Brett’s Pydagogue.

#### 3.3.2 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.

#### 3.3.3 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 git://github.com/pandas-dev/pandas.git
```

This creates the directory *pandas-yourname* and connects your repository to the upstream (main project) *pandas* repository.

#### 3.3.4 Creating a branch

You want your master 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:
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.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest pandas git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

### 3.3.5 Creating a development environment

An easy way to create a pandas development environment is as follows.

- Install either Anaconda or miniconda
- Make sure that you have cloned the repository
- cd to the pandas source directory

Tell conda to create a new environment, named `pandas_dev`, or any other name you would like for this environment, by running:

```
conda create -n pandas_dev --file ci/requirements_dev.txt
```

For a python 3 environment:

```
conda create -n pandas_dev python=3 --file ci/requirements_dev.txt
```

**Warning:** If you are on Windows, see [here for a fully compliant Windows environment](#).

This will create the new environment, and not touch any of your existing environments, nor any existing python installation. It will install all of the basic dependencies of pandas, as well as the development and testing tools. If you would like to install other dependencies, you can install them as follows:

```
conda install -n pandas_dev -c pandas pytables scipy
```

To install all pandas dependencies you can do the following:

```
conda install -n pandas_dev -c conda-forge --file ci/requirements_all.txt
```

To work in this environment, Windows users should activate it as follows:

```
activate pandas_dev
```

Mac OSX / Linux users should use:

```
source activate pandas_dev
```
You will then see a confirmation message to indicate you are in the new development environment.

To view your environments:

```bash
conda info -e
```

To return to your home root environment in Windows:

```bash
deactivate
```

To return to your home root environment in OSX / Linux:

```bash
source deactivate
```

See the full conda docs here.

At this point you can easily do an in-place install, as detailed in the next section.

### 3.3.6 Creating a Windows development environment

To build on Windows, you need to have compilers installed to build the extensions. You will need to install the appropriate Visual Studio compilers, VS 2008 for Python 2.7, VS 2010 for 3.4, and VS 2015 for Python 3.5 and 3.6.

For Python 2.7, you can install the mingw compiler which will work equivalently to VS 2008:

```bash
conda install -n pandas_dev libpython
```

or use the Microsoft Visual Studio VC++ compiler for Python. Note that you have to check the x64 box to install the x64 extension building capability as this is not installed by default.

For Python 3.4, you can download and install the Windows 7.1 SDK. Read the references below as there may be various gotchas during the installation.

For Python 3.5 and 3.6, you can download and install the Visual Studio 2015 Community Edition.

Here are some references and blogs:

- https://cowboyprogrammer.org/building-python-wheels-for-windows/
- https://blog.ionelmc.ro/2014/12/21/compiling-python-extensions-on-windows/

### 3.3.7 Making changes

Before making your code changes, it is often necessary to build the code that was just checked out. There are two primary methods of doing this.

1. The best way to develop pandas is to build the C extensions in-place by running:

   ```bash
   python setup.py build_ext --inplace
   ```

   If you startup the Python interpreter in the pandas source directory you will call the built C extensions

2. Another very common option is to do a develop install of pandas:
3.4 Contributing to the documentation

If you’re not the developer type, contributing to the documentation is still of huge value. You don’t even have to be an expert on pandas to do so! Something as simple as rewriting small passages for clarity as you reference the docs is a simple but effective way to contribute. The next person to read that passage will be in your debt!

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 simple way to ensure it will help the next person.

3.4.1 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 pandas/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 the Numpy Docstring Standard, which is used widely in the Scientific Python community. This standard specifies the format of the different sections of the docstring. See this document for a detailed explanation, or look at some of the existing functions to extend it in a similar manner.

- 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

  ```python
  x = 2
  x**3
  ```

  will be rendered as:
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.

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/contributing.rst. Then, to generate CONTRIBUTING.md, use pandoc with the following command:

```bash
pandoc doc/source/contributing.rst -t markdown_github > CONTRIBUTING.md
```

The utility script scripts/api_rst_coverage.py can be used to compare the list of methods documented in doc/source/api.rst (which is used to generate the API Reference page) and the actual public methods. This will identify methods documented in doc/source/api.rst that are not actually class methods, and existing methods that are not documented in doc/source/api.rst.

### 3.4.2 How to build the pandas documentation

#### 3.4.2.1 Requirements

First, you need to have a development environment to be able to build pandas (see the docs on creating a development environment above). Further, to build the docs, there are some extra requirements: you will need to have sphinx and ipython installed. numpydoc is used to parse the docstrings that follow the Numpy Docstring Standard (see above), but you don’t need to install this because a local copy of numpydoc is included in the pandas source code. nbsphinx is required to build the Jupyter notebooks included in the documentation.

If you have a conda environment named pandas_dev, you can install the extra requirements with:

```bash
conda install -n pandas_dev sphinx ipython nbconvert nbformat
conda install -n pandas_dev -c conda-forge nbsphinx
```

Furthermore, it is recommended to have all optional dependencies. installed. This is not strictly necessary, but be aware that you will see some error messages when building the docs. This happens because all the code in the documentation is executed during the doc build, and so code examples using optional dependencies will generate errors. Run pd.show_versions() to get an overview of the installed version of all dependencies.

**Warning:** You need to have sphinx version >= 1.3.2.

#### 3.4.2.2 Building the documentation

So how do you build the docs? Navigate to your local pandas/doc/ directory in the console and run:

```bash
python make.py html
```

Then you can find the HTML output in the folder pandas/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.

### 3.4. Contributing to the documentation
If you want to do a full clean build, do:

```
python make.py clean
python make.py html
```

Starting with pandas 0.13.1 you can tell `make.py` to compile only a single section of the docs, greatly reducing the turn-around time for checking your changes. You will be prompted to delete .rst files that aren’t required. This is okay because the prior versions of these files can be checked out from git. However, you must make sure not to commit the file deletions to your Git repository!

```
#omit autosummary and API section
python make.py clean
python make.py --no-api

# compile the docs with only a single section, that which is in indexing.rst
python make.py clean
python make.py --single indexing
```

For comparison, a full documentation build may take 10 minutes, a `--no-api` build may take 3 minutes and a single section may take 15 seconds. Subsequent builds, which only process portions you have changed, will be faster. Open the following file in a web browser to see the full documentation you just built:

```
pandas/docs/build/html/index.html
```

And you’ll have the satisfaction of seeing your new and improved documentation!

### 3.4.2.3 Building master branch documentation

When pull requests are merged into the pandas master 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.

### 3.5 Contributing to the code base

**Code Base:**

- Code standards
  - C (cpplint)
  - Python (PEP8)
  - Backwards Compatibility
- Testing With Continuous Integration
- Test-driven development/code writing
  - Writing tests
  - Transitioning to pytest
  - Using pytest
- Running the test suite
3.5.1 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.

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.

Additional standards are outlined on the code style wiki page.

3.5.1.1 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, just 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.

3.5.1.2 Python (PEP8)

*pandas* uses the PEP8 standard. There are several tools to ensure you abide by this standard. Here are some of the more common PEP8 issues:

- we restrict line-length to 79 characters to promote readability
- passing arguments should have spaces after commas, e.g. `foo(arg1, arg2, kw1='bar')`

Continuous Integration will run the flake8 tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself on the diff:

```
git diff master -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 will take longer:

```
git diff master --name-only -- "*.py" | grep "pandas/" | xargs -r flake8
```

Note that on OSX, the `-r` flag is not available, so you have to omit it and run this slightly modified command:

```
git diff master --name-only -- "*.py" | grep "pandas/" | xargs flake8
```

Note that on Windows, these commands are unfortunately not possible because commands like `grep` and `xargs` are not available natively. To imitate the behavior with the commands above, you should run:

```
git diff master --name-only -- "*.py"
```

This will list all of the Python files that have been modified. The only ones that matter during linting are any whose directory filepath begins with “pandas.” For each filepath, copy and paste it after the `flake8` command as shown below:

```
flake8 <python-filepath>
```

Alternatively, you can install the `grep` and `xargs` commands via the MinGW toolchain, and it will allow you to run the commands above.

3.5.1.3 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.

3.5.2 Testing With Continuous Integration

The *pandas* test suite will run automatically on Travis-CI, Appveyor, and Circle CI 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 Travis-CI, Appveyor, and CircleCI.
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.

![Green build example](image)

**Note:** Each time you push to your fork, a new run of the tests will be triggered on the CI. Appveyor will auto-cancel any non-currently-running tests for that same pull-request. You can enable the auto-cancel feature for Travis-CI [here](#) and for CircleCI [here](#).

### 3.5.3 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 3.1.0.

#### 3.5.3.1 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 at these for inspiration. If your test requires working with files or network connectivity, there is more information on the testing page of the wiki.
The `pandas.util.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)
    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)
```

### 3.5.3.2 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(object):
    ....
```

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():
    ....
```

### 3.5.3.3 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.
import pytest
import numpy as np
import pandas as pd
from pandas.util import testing as tm

@ pytest.mark.parametrize('dtype', ['int8', 'int16', 'int32', 'int64'])
def test_dtypes(dtype):
    assert str(np.dtype(dtype)) == dtype

@ 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

```bash
$ (pandas) bash-3.2$ pytest test_cool_feature.py -v
=========================== test session starts ===========================
platform darwin -- Python 3.6.2, pytest-3.2.1, 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_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.

```bash
$ (pandas) bash-3.2$ pytest test_cool_feature.py -v -k int8
```

A test run of this yields

```bash
$ (pandas) bash-3.2$ pytest test_cool_feature.py -v -k int8
=========================== test session starts ===========================
platform darwin -- Python 3.6.2, pytest-3.2.1, 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_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.

```bash
$ (pandas) bash-3.2$ pytest test_cool_feature.py -v -k int8
```
3.5.4 Running the test suite

The tests can then be run directly inside your Git clone (without having to install pandas) by typing:

```
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:

```
pytest pandas/path/to/test.py -k regex_matching_test_name
```

Or with one of the following constructs:

```
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:

```
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:

```
test_fast.sh
```

On Windows, one can type:

```
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.

New in version 0.20.0.

Furthermore one can run

```
pd.test()
```

with an imported pandas to run tests similarly.

3.5.5 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. asv supports both python2 and python3.
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/spacetelescope/asv
```

If you need to run a benchmark, change your directory to `asv_bench/` and run:

```
asv continuous -f 1.1 upstream/master 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/master 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 test suite can take up to one hour and use up to 3GB of RAM. 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 tests from a `pandas/asv_bench/benchmarks/groupby.py` file:

```
asv continuous -f 1.1 upstream/master HEAD -b ^groupby
```

If you want to only run a specific group of tests from a file, you can do it using . as a separator. For example:

```
asv continuous -f 1.1 upstream/master HEAD -b groupby.groupby_agg_builtins
```

will only run the `groupby_agg_builtins` 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:python3.5
```

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.

### 3.5.6 Documenting your code

Changes should be reflected in the release notes located in `doc/source/whatsnew/vx.y.z.txt`. 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 `''GH1234''` where `1234` is the issue/pull request number).

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 above. Further, to let users know when this feature was added, the `versionadded` directive is used. The sphinx syntax for that is:

```
.. versionadded:: 0.21.0
```

#### 3.5. Contributing to the code base

Changes should be reflected in the release notes located in `doc/source/whatsnew/vx.y.z.txt`. 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 `''GH1234''` where `1234` is the issue/pull request number).

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 above. Further, to let users know when this feature was added, the `versionadded` directive is used. The sphinx syntax for that is:
3.6 Contributing your changes to pandas

3.6.1 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:

```bash
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```bash
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
- 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:

```bash
git commit -m
```
3.6.2 Combining commits

If you have multiple commits, you may want to combine them into one commit, often referred to as “squashing” or “rebasing”. This is a common request by package maintainers when submitting a pull request as it maintains a more compact commit history. To rebase your commits:

```
git rebase -i HEAD~#
```

Where # is the number of commits you want to combine. Then you can pick the relevant commit message and discard others.

To squash to the master branch do:

```
git rebase -i master
```

Use the s option on a commit to squash, meaning to keep the commit messages, or f to fixup, meaning to merge the commit messages.

Then you will need to push the branch (see below) forcefully to replace the current commits with the new ones:

```
git push origin shiny-new-feature -f
```

3.6.3 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.

3.6.4 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 master and shiny-new-feature, respectively.

3.6. Contributing your changes to pandas
3.6.5 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 master version. This pull request and its associated changes will eventually be committed to the master 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. If you need to make more changes, you can make them in your branch, push them to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push -f origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the Continuous Integration tests.

3.6.6 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 master into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can just 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:

```
git push origin --delete shiny-new-feature
```
pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

pandas consists of the following elements

- A set of labeled array data structures, the primary of which are Series and DataFrame
- Index objects enabling both simple axis indexing and multi-level / hierarchical axis indexing
- An integrated group by engine for aggregating and transforming data sets
- Date range generation (date_range) and custom date offsets enabling the implementation of customized frequencies
- Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects from the fast and efficient PyTables/HDF5 format.
- Memory-efficient “sparse” versions of the standard data structures for storing data that is mostly missing or mostly constant (some fixed value)
- Moving window statistics (rolling mean, rolling standard deviation, etc.)

### 4.1 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>heterogeneously-typed column</td>
</tr>
</tbody>
</table>

#### 4.1.1 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 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. And 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
```

## 4.2 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, though, we like to favor **immutability** where sensible.

## 4.3 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.

## 4.4 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 Contributing to pandas webpage.

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

## 4.5 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).

## 4.6 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.
4.7 Institutional Partners

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

4.8 License

BSD 3-Clause License

Copyright (c) 2008-2012, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData Development Team
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
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 pandas as pd
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

### 5.1 Object Creation

See the *Data Structure Intro section*

Creating a *Series* by passing a list of values, letting pandas create a default integer index:

```python
In [4]: s = pd.Series([1,3,5,np.nan,6,8])
In [5]: s
Out[5]:
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:

```python
In [6]: dates = pd.date_range('20130101', periods=6)
In [7]: dates
Out[7]:
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 [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
In [9]: df
Out[9]:
          A         B         C         D
```

423
Creating a DataFrame by passing a dict of objects that can be converted to series-like.

```
In [10]: df2 = pd.DataFrame({'A': 1.,
                        ....: '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 [11]: df2
```

<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>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>1.0</td>
<td>2013-01-02</td>
<td>1.0</td>
<td>3</td>
<td>test</td>
<td>foo</td>
</tr>
<tr>
<td>1.0</td>
<td>2013-01-02</td>
<td>1.0</td>
<td>3</td>
<td>train</td>
<td>foo</td>
</tr>
<tr>
<td>1.0</td>
<td>2013-01-02</td>
<td>1.0</td>
<td>3</td>
<td>test</td>
<td>foo</td>
</tr>
<tr>
<td>1.0</td>
<td>2013-01-02</td>
<td>1.0</td>
<td>3</td>
<td>train</td>
<td>foo</td>
</tr>
</tbody>
</table>

Having specific dtypes

```
In [12]: df2.dtypes
```

<table>
<thead>
<tr>
<th>A</th>
<th>float64</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>C</td>
<td>float32</td>
</tr>
<tr>
<td>D</td>
<td>int32</td>
</tr>
<tr>
<td>E</td>
<td>category</td>
</tr>
<tr>
<td>F</td>
<td>object</td>
</tr>
</tbody>
</table>

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:

```
In [13]: df2.<TAB>
```

da2.A             da2.boo
da2.abs           da2.boxplot
da2.add           da2.C
da2.add_prefix    da2.clip
da2.add_suffix    da2.clip_lower
da2.align         da2.clip_upper
da2.all           da2.columns
da2.any           da2.combine
da2.append        da2.combine_first
da2.apply         da2.compound
da2.applymap      da2.consolidate
da2.D

As you can see, the columns A, B, C, and D are automatically tab completed. E is there as well; the rest of the attributes have been truncated for brevity.
5.2 Viewing Data

See the Basics section

See the top & bottom rows of the frame

\begin{verbatim}
In [14]: df.head()
Out[14]:
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
2013-01-03 -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
\end{verbatim}

\begin{verbatim}
In [15]: df.tail(3)
Out[15]:
A    B         C         D
2013-01-04 0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
\end{verbatim}

Display the index, columns, and the underlying numpy data

\begin{verbatim}
In [16]: df.index
Out[16]:
dtype='datetime64[ns]', freq='D')

In [17]: df.columns
Out[17]:
Index(['A', 'B', 'C', 'D'], dtype='object')

In [18]: df.values
Out[18]:
array([[ 0.4691, -0.2829, -1.5091, -1.1356],
       [ 1.2121, -0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949,  1.0718],
       [ 0.7216, -0.7068, -1.0396,  0.2719],
       [-0.425 ,  0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784,  0.525 ]])
\end{verbatim}

Describe shows a quick statistic summary of your data

\begin{verbatim}
In [19]: df.describe()
Out[19]:
          A         B         C         D
count 6.000000 6.000000 6.000000 6.000000
mean  0.073711 -0.431125 -0.687758 -0.233103
std   0.843157  0.922818  0.779887  0.973118
min  -0.861849 -2.104569 -1.509059 -1.135632
25%  -0.611510 -0.600794 -1.368714 -1.076610
50%   0.022070 -0.228039  0.779887  0.973118
75%   0.658444  0.041933 -0.034326  0.461706
max   1.212112  0.567020  0.276232  1.071804
\end{verbatim}
Transposing your data

```python
In [20]: df.T
Out[20]:
A    0.469112  1.212112  -0.861849  0.721555  -0.424972  -0.673690
B   -0.282863 -0.173215  -2.104569  -0.706771  0.567020   0.113648
C   -1.509059  0.119209  -0.494929  -1.039575  0.276232  -1.478427
D   -1.135632 -1.044236   1.071804  0.271860  -1.087401   0.524988
```

Sorting by an axis

```python
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
     D     C     B     A
2013-01-01 -1.135632 -1.509059 -0.282863  0.469112
2013-01-02 -1.044236  0.119209 -0.173215  1.212112
2013-01-03  1.071804 -0.494929 -2.104569  -0.861849
2013-01-04  0.271860  -1.039575  -0.706771  0.721555
2013-01-05  1.087401  0.276232  0.567020 -0.424972
2013-01-06  0.524988 -1.478427  0.113648 -0.673690
```

Sorting by values

```python
In [22]: df.sort_values(by='B')
Out[22]:
     A     B     C     D
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555  -0.706771 -1.039575  0.271860
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

5.3 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, \( .at \), \( .iat \), \( .loc \), \( .iloc \) and \( .ix \).

See the indexing documentation Indexing and Selecting Data and MultiIndex / Advanced Indexing

5.3.1 Getting

Selecting a single column, which yields a Series, equivalent to \( df.A \)

```python
In [23]: df['A']
Out[23]:
   2013-01-01    0.469112
   2013-01-02    1.212112
   2013-01-03   -0.861849
   2013-01-04    0.721555
   2013-01-05   -0.424972
```
Selecting via [], which slices the rows.

In [24]: df[0:3]
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
2013-01-03 -0.861849 -2.104569 -0.494929 1.071804

In [25]: df['20130102':'20130104']
→
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 -0.706771 -1.039575 0.271860

5.3.2 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]:
A   0.469112
B  -0.282863
C  -1.509059
D  -1.135632
Name: 2013-01-01 00:00:00, dtype: float64

Selecting on a multi-axis by label

In [27]: df.loc[:,['A','B']]
Out [27]:
A  B
2013-01-01 0.469112 -0.282863
2013-01-02 1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04 0.721555 -0.706771
2013-01-05 -0.424972 0.567020
2013-01-06 -0.673690 0.113648

Showing label slicing, both endpoints are included

In [28]: df.loc['20130102':'20130104',['A','B']]
Out [28]:
A  B
2013-01-02 1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04 0.721555 -0.706771

Reduction in the dimensions of the returned object

5.3. Selection
In [29]: df.loc['20130102',['A','B']]
Out[29]:
A  1.212112
B -0.173215
Name: 2013-01-02 00:00:00, dtype: float64

For getting a scalar value

In [30]: df.loc[dates[0],'A']
Out[30]: 0.46911229990718628

For getting fast access to a scalar (equiv to the prior method)

In [31]: df.at[dates[0],'A']
Out[31]: 0.46911229990718628

5.3.3 Selection by Position

See more in Selection by Position

Select via the position of the passed integers

In [32]: df.iloc[3]
Out[32]:
A  0.721555
B -0.706771
C -1.039575
D  0.271860
Name: 2013-01-04 00:00:00, dtype: float64

By integer slices, acting similar to numpy/python

In [33]: df.iloc[3:5,0:2]
Out[33]:
           A  B
2013-01-04 0.721555 -0.706771
2013-01-05 -0.424972  0.567020

By lists of integer position locations, similar to the numpy/python style

In [34]: df.iloc[[1,2,4],[0,2]]
Out[34]:
           A   C
2013-01-02 1.212112 0.119209
2013-01-03 -0.861849 -0.494929
2013-01-05 -0.424972  0.276232

For slicing rows explicitly

In [35]: df.iloc[1:3,:]
Out[35]:
           A   B   C   D
2013-01-02 1.212112 -0.173215 0.119209 0.044236
2013-01-03 -0.861849 -2.104569 -0.494929 1.071804

For slicing columns explicitly
In [36]: df.iloc[:,1:3]
Out[36]:

       B       C
2013-01-01 -0.282863 -1.509059
2013-01-02 -0.173215  0.119209
2013-01-03 -2.104569 -0.494929
2013-01-04 -0.706771 -1.039575
2013-01-05  0.567020  0.276232
2013-01-06  0.113648 -1.478427

For getting a value explicitly

In [37]: df.iloc[1,1]
Out[37]: -0.17321464905330858

For getting fast access to a scalar (equiv to the prior method)

In [38]: df.iat[1,1]
Out[38]: -0.17321464905330858

### 5.3.4 Boolean Indexing

Using a single column’s values to select data.

In [39]: df[df.A > 0]
Out[39]:

       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

Selecting values from a DataFrame where a boolean condition is met.

In [40]: df[df > 0]
Out[40]:

       A       B       C       D
2013-01-01  0.469112  NaN    NaN    NaN
2013-01-02  1.212112  NaN  0.119209  NaN
2013-01-03  NaN     NaN    NaN  1.071804
2013-01-04  0.721555  NaN    NaN  0.271860
2013-01-05  NaN  0.567020  0.276232  NaN
2013-01-06  NaN  0.113648  NaN  0.524988

Using the *isin()* method for filtering:

In [41]: df2 = df.copy()

In [42]: df2['E'] = ['one', 'one','two','three','four','three']

In [43]: df2
Out[43]:

       A       B       C       D       E
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632 one
2013-01-02  1.212112 -0.173215  0.119209 -1.044236 one
2013-01-03  0.861849 -2.104569 -0.494929  1.071804 two
2013-01-04  0.721555 -0.706771 -1.039575  0.271860 three

5.3. Selection
In [44]: df2[df2['E'].isin(['two','four'])]

\[
\begin{array}{cccc}
2013-01-03 & -0.861849 & -2.104569 & -0.494929 & 1.071804 \text{ two} \\
2013-01-05 & -0.424972 & 0.567020 & 0.276232 & -1.087401 \text{ four}
\end{array}
\]

5.3.5 Setting

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]:
\[
\begin{array}{cccccc}
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} \\
2013-01-01 & 0.000000 & 0.000000 & -1.509059 & 5 & \text{NaN} \\
2013-01-02 & 1.212112 & -0.173215 & 0.119209 & 5 & 1.0 \\
2013-01-03 & -0.861849 & -2.104569 & -0.494929 & 5 & 2.0 \\
2013-01-04 & 0.721555 & -0.706771 & -1.039575 & 5 & 3.0 \\
2013-01-05 & -0.424972 & 0.567020 & 0.276232 & 5 & 4.0 \\
2013-01-06 & -0.673690 & 0.113648 & -1.478427 & 5 & 5.0
\end{array}
\]

A where operation with setting.

In [52]: df2 = df.copy()
In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]:
         A         B         C         D         F
2013-01-01 0.000000 0.000000 -1.509059   -5  NaN
2013-01-02 -1.212112 -0.173215  -0.119209   -5  -1.0
2013-01-03 -0.861849 -2.104569  -0.494929   -5  -2.0
2013-01-04 -0.721555 -0.706771  -1.039575   -5  -3.0
2013-01-05 -0.424972 -0.567020  -0.276232   -5  -4.0
2013-01-06 -0.673690 -0.113648  -1.478427   -5  -5.0

5.4 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]:
         A         B         C         D         F         E
2013-01-01 0.000000 0.000000 -1.509059   -5  NaN       1.0
2013-01-02 1.212112 -0.173215  -0.119209   -5  1.0       1.0
2013-01-03 -0.861849 -2.104569  -0.494929   -5  2.0       NaN
2013-01-04 -0.721555 -0.706771  -1.039575   -5  3.0       NaN

To drop any rows that have missing data.

In [58]: df1.dropna(how='any')
Out[58]:
         A         B         C         D         F         E
2013-01-02 1.212112 -0.173215  -0.119209   -5  1.0       1.0

Filling missing data

In [59]: df1.fillna(value=5)
Out[59]:
         A         B         C         D         F         E
2013-01-01 0.000000 0.000000 -1.509059   -5   5.0       1.0
2013-01-02 1.212112 -0.173215  -0.119209   -5   1.0       1.0
2013-01-03 -0.861849 -2.104569  -0.494929   -5   2.0       5.0
2013-01-04 -0.721555 -0.706771  -1.039575   -5   3.0       5.0

To get the boolean mask where values are `nan`

In [60]: pd.isna(df1)
Out[60]:
         A         B         C         D         F         E
2013-01-01    False    False    False    False   True    False
2013-01-02    False    False    False    False    True    False

5.4. Missing Data
5.5 Operations

See the *Basic section on Binary Ops*

5.5.1 Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic

```
In [61]: df.mean()
Out[61]:
      A     B     C     D     F
2013-01-01 -0.004474 -0.383981 -0.687758  5.000000  3.000000
Freq: D, dtype: float64
```

Same operation on the other axis

```
In [62]: df.mean(1)
Out[62]:
2013-01-01  0.872735
2013-01-02  1.431621
2013-01-03  0.707731
2013-01-04  1.395042
2013-01-05  1.883656
2013-01-06  1.592306
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
Freq: D, dtype: float64

In [65]: df.sub(s, axis='index')
```

```
→
<table>
<thead>
<tr>
<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>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```
5.5.2 Apply

Applying functions to the data

```python
In [66]: df.apply(np.cumsum)
```

<table>
<thead>
<tr>
<th>Date</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>0.00</td>
<td>0.00</td>
<td>-1.509059</td>
<td>5</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>-1.389850</td>
<td>10</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.350263</td>
<td>-2.277784</td>
<td>-1.884779</td>
<td>15</td>
<td>3.0</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>1.071818</td>
<td>-2.984555</td>
<td>-2.924354</td>
<td>20</td>
<td>6.0</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>0.646846</td>
<td>-2.417535</td>
<td>-2.648122</td>
<td>25</td>
<td>10.0</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-0.026844</td>
<td>-2.303886</td>
<td>-4.126549</td>
<td>30</td>
<td>15.0</td>
</tr>
</tbody>
</table>

```python
In [67]: df.apply(lambda x: x.max() - x.min())
```

<table>
<thead>
<tr>
<th>Series</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2.073961</td>
</tr>
<tr>
<td>B</td>
<td>2.671590</td>
</tr>
<tr>
<td>C</td>
<td>1.785291</td>
</tr>
<tr>
<td>D</td>
<td>0.000000</td>
</tr>
<tr>
<td>F</td>
<td>4.000000</td>
</tr>
</tbody>
</table>

dtype: float64

5.5.3 Histogramming

See more at Histogramming and Discretization

```python
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))
In [69]: s
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

dtype: int64

```python
In [70]: s.value_counts()
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

5.5. Operations
5.5.4 String Methods

Series is equipped with a set of string processing methods in the `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 `str` generally uses regular expressions by default (and in some cases always uses them). See more at Vectorized String Methods.

In [71]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
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

5.6 Merge

5.6.1 Concat

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the Merging section

Concatenating pandas objects together with `concat()`:

In [73]: df = pd.DataFrame(np.random.randn(10, 4))
In [74]: df
Out[74]:
     0          1         2          3
0  -0.548702  1.467327 -1.015962 -0.483075
1   1.637550 -1.217659 -0.291519 -1.745505
2  -0.263952  0.991460 -0.919069  0.266046
3  -0.709661  1.669052  1.037882 -1.705775
4  -0.919854  0.042379  1.247642 -0.009920
5   0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7  -0.932132  1.956030  0.017587 -0.016692
8  -0.575247  0.254161 -1.143704  0.215897
9   1.193555 -0.077118 -0.408530 -0.862495

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]
5.6.2 Join

SQL style merges. See the Database style joining

```
In [77]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
In [78]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [79]: left
Out[79]:
      key lval
0    foo   1
1    foo   2

In [80]: right
Out[80]:
       key rval
0    foo    4
1    foo    5

In [81]: pd.merge(left, right, on='key')
Out[81]:
        key  lval  rval
0    foo    1     4
1    foo    1     5
2    foo    2     4
3    foo    2     5
```

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

\[
\begin{array}{ll}
\text{key} & \text{rval} \\
0 & \text{foo} 4 \\
1 & \text{bar} 5
\end{array}
\]

Out[85]:

\[
\begin{array}{ll}
\text{key} & \text{rval} \\
0 & \text{foo} 4 \\
1 & \text{bar} 5
\end{array}
\]

In [86]: pd.merge(left, right, on='key')

\[
\begin{array}{lll}
\text{key} & \text{lval} & \text{rval} \\
0 & \text{foo} 1 & 4 \\
1 & \text{bar} 2 & 5
\end{array}
\]

5.6.3 Append

Append rows to a dataframe. See the Appending

In [87]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [88]: df

Out[88]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.346061</td>
<td>1.511763</td>
<td>1.627081</td>
<td>-0.990582</td>
</tr>
<tr>
<td>1</td>
<td>-0.441652</td>
<td>1.211526</td>
<td>0.268520</td>
<td>0.024580</td>
</tr>
<tr>
<td>2</td>
<td>-1.577585</td>
<td>0.396823</td>
<td>-0.105381</td>
<td>-0.532532</td>
</tr>
<tr>
<td>3</td>
<td>1.453749</td>
<td>1.208843</td>
<td>-0.080952</td>
<td>-0.264610</td>
</tr>
<tr>
<td>4</td>
<td>-0.727965</td>
<td>-0.589346</td>
<td>0.339969</td>
<td>-0.693205</td>
</tr>
<tr>
<td>5</td>
<td>-0.339355</td>
<td>0.593616</td>
<td>0.884345</td>
<td>1.591431</td>
</tr>
<tr>
<td>6</td>
<td>0.141809</td>
<td>0.220390</td>
<td>0.435589</td>
<td>0.192451</td>
</tr>
<tr>
<td>7</td>
<td>-0.096701</td>
<td>0.803351</td>
<td>1.715071</td>
<td>-0.708758</td>
</tr>
</tbody>
</table>

In [89]: s = df.iloc[3]

In [90]: df.append(s, ignore_index=True)

Out[90]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.346061</td>
<td>1.511763</td>
<td>1.627081</td>
<td>-0.990582</td>
</tr>
<tr>
<td>1</td>
<td>-0.441652</td>
<td>1.211526</td>
<td>0.268520</td>
<td>0.024580</td>
</tr>
<tr>
<td>2</td>
<td>-1.577585</td>
<td>0.396823</td>
<td>-0.105381</td>
<td>-0.532532</td>
</tr>
<tr>
<td>3</td>
<td>1.453749</td>
<td>1.208843</td>
<td>-0.080952</td>
<td>-0.264610</td>
</tr>
<tr>
<td>4</td>
<td>-0.727965</td>
<td>-0.589346</td>
<td>0.339969</td>
<td>-0.693205</td>
</tr>
<tr>
<td>5</td>
<td>-0.339355</td>
<td>0.593616</td>
<td>0.884345</td>
<td>1.591431</td>
</tr>
<tr>
<td>6</td>
<td>0.141809</td>
<td>0.220390</td>
<td>0.435589</td>
<td>0.192451</td>
</tr>
<tr>
<td>7</td>
<td>-0.096701</td>
<td>0.803351</td>
<td>1.715071</td>
<td>-0.708758</td>
</tr>
<tr>
<td>8</td>
<td>1.453749</td>
<td>1.208843</td>
<td>-0.080952</td>
<td>-0.264610</td>
</tr>
</tbody>
</table>

5.7 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

```python
In [91]: 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 [92]: df
Out[92]:
    A  B     C       D
0  foo one -1.202872 -0.055224
1  bar one -1.814470  2.395985
2  foo two  1.018601  1.552825
3  bar three -0.595447  0.166599
4  foo two  1.395433  0.047609
5  bar two -0.392670 -0.136473
6  foo one  0.007207 -0.561757
7  foo three  1.928123 -1.623033
```

Grouping and then applying a function `sum` to the resulting groups.

```python
In [93]: df.groupby('A').sum()
Out[93]:
    C       D
A
bar   -2.802588  2.42611
foo    3.146492 -0.63958
```

Grouping by multiple columns forms a hierarchical index, which we then apply the function.

```python
In [94]: df.groupby(['A', 'B']).sum()
Out[94]:
    C       D
A B
bar one  -1.814470  2.395985
       three  -0.595447  0.166599
       two   -0.392670 -0.136473
foo one  -1.195665 -0.616981
       three  1.928123 -1.623033
       two   2.414034  1.600434
```

5.8 Reshaping

See the sections on Hierarchical Indexing and Reshaping.

5.8.1 Stack

```python
In [95]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                           'foo', 'foo', 'qux', 'qux'],
                          ['one', 'two', 'one', 'two',
                           'one', 'two', 'one', 'two']]))
```
In 

```
index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

```
df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])
```

```
df2 = df[:4]
```

```
df2
```

```
Out[99]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>one</td>
<td>0.029399</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.282696</td>
</tr>
<tr>
<td>baz</td>
<td>one</td>
<td>-1.575170</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.816482</td>
</tr>
</tbody>
</table>

The `stack()` method “compresses” a level in the DataFrame’s columns.

```
stacked = df2.stack()
```

```
stacked
```

```
Out[101]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>one</td>
<td>0.029399</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.282696</td>
</tr>
<tr>
<td>baz</td>
<td>one</td>
<td>-1.575170</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.816482</td>
</tr>
</tbody>
</table>

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:

```
stacked.unstack()
```

```
stacked.unstack(1)
```

```
stacked.unstack(0)
```

"""
### 5.8.2 Pivot Tables

See the section on *Pivot Tables*.

```python
In [105]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
                    'B': ['A', 'B', 'C'] * 4,
                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
                    'D': np.random.randn(12),
                    'E': np.random.randn(12)})

In [106]: df
Out[106]:
    A  B  C       D       E
0  one A  foo  1.418757 -0.179666
1  one B  foo -1.879024  1.291836
2  two C  foo  0.536826 -0.009614
3  three A  bar  1.006160  0.392149
4   one B  bar -0.029716  0.264599
5   one C  bar -1.146178 -0.057409
6  two A  foo  0.100900 -1.425638
7  three B  foo -1.035018  1.024098
8   one C  foo  0.314665 -0.106062
9   one A  bar -0.773723  1.824375
10  two B  bar -1.170653  0.595974
11  three C  bar  0.648740  1.167115
```

We can produce pivot tables from this data very easily:

```python
In [107]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[107]:
    C     bar     foo
A  B
one A  -0.773723  1.418757
     B  -0.029716 -1.879024
     C -1.146178  0.314665
three A  1.006160     NaN
       B  NaN  -1.035018
       C  0.648740     NaN
two A     NaN  0.100900
       B -1.170653     NaN
       C     NaN  0.536826
```

### 5.9 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

In [108]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [109]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [110]: ts.resample('5Min').sum()

Out[110]:
2012-01-01  25083
Freq: 5T, dtype: int64

Time zone representation

In [111]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [112]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [113]: ts

Out[113]:
2012-03-06  0.464000
2012-03-07  0.227371
2012-03-08 -0.496922
2012-03-09  0.306389
2012-03-10 -2.290613
Freq: D, dtype: float64

In [114]: ts_utc = ts.tz_localize('UTC')

In [115]: ts_utc

Out[115]:
2012-03-06 00:00:00+00:00  0.464000
2012-03-07 00:00:00+00:00  0.227371
2012-03-08 00:00:00+00:00 -0.496922
2012-03-09 00:00:00+00:00  0.306389
2012-03-10 00:00:00+00:00 -2.290613
Freq: D, dtype: float64

Convert to another time zone

In [116]: ts_utc.tz_convert('US/Eastern')

Out[116]:
2012-03-05 19:00:00-05:00  0.464000
2012-03-06 19:00:00-05:00  0.227371
2012-03-07 19:00:00-05:00 -0.496922
2012-03-08 19:00:00-05:00  0.306389
2012-03-09 19:00:00-05:00 -2.290613
Freq: D, dtype: float64

Converting between time span representations

In [117]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [118]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [119]: ts

Out[119]:
2012-01-31 -1.134623
2012-02-29 -1.561819
2012-03-31 -0.260838
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 [123]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [124]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [125]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [126]: ts.head()
```

```
1990-03-01 09:00 -0.902937
1990-06-01 09:00 0.068159
1990-09-01 09:00 -0.057873
1990-12-01 09:00 -0.368204
1991-03-01 09:00 -1.144073
Freq: H, dtype: float64
```

### 5.10 Categoricals

Pandas can include categorical data in a DataFrame. For full docs, see the categorical introduction and the API documentation.

```python
In [127]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6], "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
```

Convert the raw grades to a categorical data type.

5.10. Categoricals
In [128]: df["grade"] = df["raw_grade"].astype("category")

In [129]: df["grade"]

Out[129]:
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 inplace!)

In [130]: 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 per default).

In [131]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [132]: df["grade"]

Out[132]:
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.

In [133]: df.sort_values(by="grade")

Out[133]:
   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 shows also empty categories.

In [134]: df.groupby("grade").size()

Out[134]:
groupby("grade").size()
grade  very bad  1
       bad      0
       medium  0
       good    2
       very good  3
dtype: int64
5.11 Plotting

`plot()` docs.

```python
In [135]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [136]: ts = ts.cumsum()

In [137]: ts.plot()
```

On DataFrame, `plot()` is a convenience to plot all of the columns with labels:

```python
In [138]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                 columns=['A', 'B', 'C', 'D'])

In [139]: df = df.cumsum()

In [140]: plt.figure(); df.plot(); plt.legend(loc='best')
```

5.11. Plotting
5.12 Getting Data In/Out

5.12.1 CSV

Writing to a csv file

In [141]: df.to_csv('foo.csv')

Reading from a csv file

In [142]: pd.read_csv('foo.csv')

Out[142]:

<table>
<thead>
<tr>
<th>0</th>
<th>2000-01-01</th>
<th>0.266457</th>
<th>-0.399641</th>
<th>-0.219582</th>
<th>1.186860</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000-01-02</td>
<td>-1.170732</td>
<td>-0.345873</td>
<td>1.653061</td>
<td>-0.282953</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-03</td>
<td>-1.734933</td>
<td>0.530468</td>
<td>2.060811</td>
<td>-0.515536</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-04</td>
<td>-1.555121</td>
<td>1.452620</td>
<td>0.239859</td>
<td>-1.156896</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-05</td>
<td>0.578117</td>
<td>0.511371</td>
<td>0.103552</td>
<td>-2.428202</td>
</tr>
<tr>
<td>5</td>
<td>2000-01-06</td>
<td>0.478344</td>
<td>0.449933</td>
<td>-0.741620</td>
<td>-1.962409</td>
</tr>
<tr>
<td>6</td>
<td>2000-01-07</td>
<td>1.235339</td>
<td>0.491757</td>
<td>-1.543861</td>
<td>-1.084753</td>
</tr>
</tbody>
</table>

...  ...  ...  ...  ...  ...

| 994 | 2002-09-21 | -10.390377| -8.727491 | -6.399645 | 30.914107 |
998 2002-09-25 -10.216020 -9.480682 -3.933802 29.758560  
999 2002-09-26 -11.856774 -10.671012 -3.216025 29.369368  
[1000 rows x 5 columns]

5.12.2 HDF5

Reading and writing to HDFStores

Writing to a HDF5 Store

In [143]: df.to_hdf('foo.h5','df')

Reading from a HDF5 Store

In [144]: pd.read_hdf('foo.h5','df')
Out[144]:
          A         B         C         D
2000-01-01 0.266457 -0.399641 -0.219582 1.186860
2000-01-02 -1.170732 -0.345873 1.653061 -0.282953
2000-01-03 -1.734933  0.530468 2.060811 -0.515536
2000-01-04 -1.555121  1.452620 0.239859 -1.156896
2000-01-05  0.578117  0.511371 0.103552 -2.428202
2000-01-06  0.478344  0.449933 -0.741620 -1.962409
2000-01-07  1.235339 -0.091757 -1.543861 -1.084753
          ...          ...          ...          ...
2002-09-21 -10.390377 -8.727491 -6.399645 30.914107
2002-09-26 -11.856774 -10.671012 -3.216025 29.369368  
[1000 rows x 4 columns]

5.12.3 Excel

Reading and writing to MS Excel

Writing to an excel file

In [145]: df.to_excel('foo.xlsx', sheet_name='Sheet1')

Reading from an excel file

In [146]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
Out[146]:
          A         B         C         D
2000-01-01 0.266457 -0.399641 -0.219582 1.186860
2000-01-02 -1.170732 -0.345873 1.653061 -0.282953
2000-01-03 -1.734933  0.530468 2.060811 -0.515536
2000-01-04 -1.555121  1.452620 0.239859 -1.156896
2000-01-05  0.578117  0.511371 0.103552 -2.428202
2000-01-06  0.478344  0.449933 -0.741620 -1.962409
5.13 Gotchas

If you are trying an operation and you 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.
This is a guide to many pandas tutorials, geared mainly for new users.

### 6.1 Internal Guides

pandas own *10 Minutes to pandas*

More complex recipes are in the *Cookbook*

### 6.2 pandas Cookbook

The goal of this 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 that entails.

Here are links to the v0.1 release. For an up-to-date table of contents, see the pandas-cookbook GitHub repository. To run the examples in this tutorial, you’ll need to clone the GitHub repository and get IPython Notebook running. See How to use this cookbook.

- **A quick tour of the IPython Notebook**: Shows off IPython’s awesome tab completion and magic functions.
- **Chapter 1**: Reading your data into pandas is pretty much the easiest thing. Even when the encoding is wrong!
- **Chapter 2**: It’s not totally obvious how to select data from a pandas dataframe. Here we explain the basics (how to take slices and get columns)
- **Chapter 3**: Here we get into serious slicing and dicing and learn how to filter dataframes in complicated ways, really fast.
- **Chapter 4**: Groupby/aggregate is seriously my favorite thing about pandas and I use it all the time. You should probably read this.
- **Chapter 5**: Here you get to find out if it’s cold in Montreal in the winter (spoiler: yes). Web scraping with pandas is fun! Here we combine dataframes.
- **Chapter 6**: Strings with pandas are great. It has all these vectorized string operations and they’re the best. We will turn a bunch of strings containing “Snow” into vectors of numbers in a trice.
- **Chapter 7**: Cleaning up messy data is never a joy, but with pandas it’s easier.
- **Chapter 8**: Parsing Unix timestamps is confusing at first but it turns out to be really easy.
6.3 Lessons for New pandas Users

For more resources, please visit the main repository.

- 01 - Lesson: - Importing libraries - Creating data sets - Creating data frames - Reading from CSV - Exporting to CSV - Finding maximums - Plotting data
- 02 - Lesson: - Reading from TXT - Exporting to TXT - Selecting top/bottom records - Descriptive statistics - Grouping/sorting data
- 03 - Lesson: - Creating functions - Reading from EXCEL - Exporting to EXCEL - Outliers - Lambda functions - Slice and dice data
- 04 - Lesson: - Adding/deleting columns - Index operations
- 05 - Lesson: - Stack/Unstack/Transpose functions
- 06 - Lesson: - GroupBy function
- 07 - Lesson: - Ways to calculate outliers
- 08 - Lesson: - Read from Microsoft SQL databases
- 09 - Lesson: - Export to CSV/EXCEL/TXT
- 10 - Lesson: - Converting between different kinds of formats
- 11 - Lesson: - Combining data from various sources

6.4 Practical data analysis with Python

This guide is a comprehensive introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as follows:

- Munging Data
- Aggregating Data
- Visualizing Data
- Time Series

6.5 Exercises for New Users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

- 01 - Getting & Knowing Your Data
- 02 - Filtering & Sorting
- 03 - Grouping
- 04 - Apply
- 05 - Merge
- 06 - Stats
- 07 - Visualization
- 08 - Creating Series and DataFrames
6.6 Modern Pandas

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization

6.7 Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

6.8 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 Tutorial, by Mikhail Semeniuk
- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples
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.

These examples are written for python 3.4. Minor tweaks might be necessary for earlier python versions.

### 7.1 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]}); df
Out[1]:
     AAA  BBB  CCC
0     4    10  100
1     5    20   50
2     6    30  -30
3     7    40  -50
```

#### 7.1.1 if-then...

An if-then on one column

```
In [2]: df.loc[df.AAA >= 5,'BBB'] = -1; df
Out[2]:
     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 [3]: df.loc[df.AAA >= 5,['BBB','CCC']] = 555; df
Out[3]:
   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 [4]: df.loc[df.AAA < 5,['BBB','CCC']] = 2000; df
Out[4]:
   AAA  BBB  CCC
0   4  2000  2000
1   5   555  555
2   6   555  555
3   7   555  555

Or use pandas where after you’ve set up a mask
In [6]: df.where(df_mask,-1000)
Out[6]:
   AAA  BBB  CCC
0   4  -1000  2000
1   5   -1000  -1000
2   6  -1000   555
3   7  -1000  -1000

if-then-else using numpy’s where()
In [7]: df = pd.DataFrame(
    ...:     {"AAA" : [4,5,6,7], "BBB" : [10,20,30,40],"CCC" : [100,50,-30,-50]}); df
    ...:
Out[7]:
   AAA  BBB  CCC
0   4    10   100
1   5    20    50
2   6    30   -30
3   7    40   -50

In [8]: df["logic"] = np.where(df["AAA"] > 5,'high','low'); df
   AAA  BBB  CCC   logic
0   4    10   100  low
1   5    20    50  low
2   6    30   -30  high
3   7    40   -50  high

7.1.2 Splitting

Split a frame with a boolean criterion
7.1.3 Building Criteria

Select with multi-column criteria

```
In [12]: df = pd.DataFrame(
    ....:     {'AAA' : [4, 5, 6, 7], 'BBB' : [10, 20, 30, 40], 'CCC' : [100, 50, -30, -50]}); df
    ....:
Out[12]:
    AAA  BBB  CCC
  0   4    10  100
  1   5    20   50
  2   6    30  -30
  3   7    40  -50

In [10]: dflow = df[df.AAA <= 5]; dflow

  AAA  BBB  CCC
0   4    10  100
1   5    20   50

In [11]: dfhigh = df[df.AAA > 5]; dfhigh

  AAA  BBB  CCC
2   6    30  -30
3   7    40  -50
```

...and (without assignment returns a Series)

```
In [13]: newseries = df.loc[(df['BBB'] < 25) & (df['CCC'] >= -40), 'AAA']; newseries
Out[13]:
0   4
1   5
Name: AAA, dtype: int64

In [14]: newseries = df.loc[(df['BBB'] > 25) | (df['CCC'] >= -40), 'AAA']; newseries

    AAA
0   0.1
```

...or (with assignment modifies the DataFrame.)

```
In [15]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= 75), 'AAA'] = 0.1; df
Out[15]:
    AAA  BBB  CCC
  0  0.1    10  100
```

7.1. Idioms

---

pandas: powerful Python data analysis toolkit, Release 0.21.0
Select rows with data closest to certain value using argsort

```python
In [16]: df = pd.DataFrame(
    ...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}); df
...:
Out[16]:
     AAA  BBB  CCC
0     4    10  100
1     5    20   50
2     6    30  -30
3     7    40  -50

In [17]: aValue = 43.0

In [18]: df.loc[(df.CCC-aValue).abs().argsort()]
Out[18]:
     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

```python
In [19]: df = pd.DataFrame(
    ...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}); df
...:
Out[19]:
     AAA  BBB  CCC
0     4    10  100
1     5    20   50
2     6    30  -30
3     7    40  -50

In [20]: Crit1 = df.AAA <= 5.5

In [21]: Crit2 = df.BBB == 10.0

In [22]: Crit3 = df.CCC > -40.0

One could hard code:

```python
In [23]: AllCrit = Crit1 & Crit2 & Crit3

```

...Or it can be done with a list of dynamically built criteria

```python
In [24]: CritList = [Crit1,Crit2,Crit3]

In [25]: AllCrit = functools.reduce(lambda x,y: x & y, CritList)

In [26]: df[AllCrit]
Out[26]:
     AAA  BBB  CCC
0     4    10  100
```

454 Chapter 7. Cookbook
### 7.2 Selection

#### 7.2.1 DataFrames

The *indexing* docs.

Using both row labels and value conditionals

```python
In [27]: df = pd.DataFrame(
    ....:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}); df
    ....:
Out[27]:
     AAA  BBB  CCC
    0  4   10  100
    1  5   20   50
    2  6   30  -30
    3  7   40  -50

In [28]: df[(df.AAA <= 6) & (df.index.isin([0,2,4]))]
   
   AAA  BBB  CCC
    0  4   10  100
    2  6   30  -30

Use loc for label-oriented slicing and iloc positional slicing

```python
In [29]: data = {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}

In [30]: df = pd.DataFrame(data=data,index=['foo','bar','boo','kar']); df
   
   AAA  BBB  CCC
   foo   4   10  100
   bar   5   20   50
   boo   6   30  -30
   kar   7   40  -50

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)

```python
In [31]: df.loc['bar':'kar'] #Label
Out[31]:
     AAA  BBB  CCC
    foo   4   10  100
    bar   5   20   50
    boo   6   30  -30
    kar   7   40  -50

# Generic
In [32]: df.iloc[0:3]
   
   AAA  BBB  CCC
    foo   4   10  100
    bar   5   20   50
```

7.2. Selection 455
Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.

Using inverse operator (~) to take the complement of a mask

7.2.2 Panels

Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions
In [42]: df1, df2, df3 = pd.DataFrame(data, rng, cols), pd.DataFrame(data, rng, cols), pd.DataFrame(data, rng, cols)

In [43]: pf = pd.Panel({'df1':df1,'df2':df2,'df3':df3});pf
Out[43]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 4 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to D

In [44]: pf.loc[:,:,'F'] = pd.DataFrame(data, rng, cols);pf

Out[44]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 5 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to F

Mask a panel by using np.where and then reconstructing the panel with the new masked values

7.2.3 New Columns

Efficiently and dynamically creating new columns using applymap

In [45]: df = pd.DataFrame(
       ....:     {'AAA' : [1,2,1,3], 'BBB' : [1,1,2,2], 'CCC' : [2,1,3,1]}); df
       ....:
Out[45]:
      AAA  BBB  CCC
0    1    1    2
1    2    1    1
2    1    2    3
3    3    2    1

In [46]: source_cols = df.columns # or some subset would work too.

In [47]: new_cols = [str(x) + "_cat" for x in source_cols]

In [48]: categories = {1 : 'Alpha', 2 : 'Beta', 3 : 'Charlie' }

In [49]: df[new_cols] = df[source_cols].applymap(categories.get);df
Out[49]:
      AAA  BBB  CCC   AAA_cat  BBB_cat  CCC_cat
0    1    1    2   Alpha     Alpha   Beta
1    2    1    1   Beta      Alpha   Alpha
2    1    2    3   Alpha     Beta   Charlie
3    3    2    1   Charlie    Beta   Alpha

Keep other columns when using min() with groupby

In [50]: df = pd.DataFrame(
       ....:     {'AAA' : [1,1,1,2,2,2,3,3], 'BBB' : [2,1,3,4,5,1,2,3]}); df
       ....:
Out[50]:

7.2. Selection
Method 1 : idxmin() to get the index of the mins

```
In [51]: df.loc[df.groupby("AAA")["BBB"].idxmin()]
Out[51]:
AAA  BBB
0  1  2
1  1  1
2  1  3
3  2  4
4  2  5
5  2  1
6  3  2
7  3  3
```

Method 2 : sort then take first of each

```
In [52]: df.sort_values(by="BBB").groupby("AAA", as_index=False).first()
Out[52]:
AAA  BBB
0  1  1
1  2  1
2  3  2
```

Notice the same results, with the exception of the index.

### 7.3 MultiIndexing

The *multindexing* docs.

Creating a multi-index from a labeled frame

```
In [53]: 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]}); df
Out[53]:
One_X One_Y Two_X Two_Y row
 0  1.1  1.2  1.11  1.22  0
 1  1.1  1.2  1.11  1.22  1
 2  1.1  1.2  1.11  1.22  2

# As Labelled Index
In [54]: df = df.set_index('row');df
```

```
  One_X One_Y Two_X Two_Y
row
  0  1.1  1.2  1.11  1.22
  1  1.1  1.2  1.11  1.22
```
# With Hierarchical Columns

```
In [55]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split('_')) for c in df.columns]); df
```

```
Out[55]:

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>Two</td>
<td>1.11</td>
<td>1.22</td>
</tr>
</tbody>
</table>
```

# Now stack & Reset

```
In [56]: df = df.stack(0).reset_index(1);df
```

```
Out[56]:

<table>
<thead>
<tr>
<th>level_1</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>1.10</td>
<td>1.20</td>
</tr>
<tr>
<td>Two</td>
<td>1.11</td>
<td>1.22</td>
</tr>
</tbody>
</table>
```

# And fix the labels (Notice the label 'level_1' got added automatically)

```
In [57]: df.columns = ['Sample','All_X','All_Y'];df
```

```
Out[57]:

<table>
<thead>
<tr>
<th></th>
<th>All_X</th>
<th>All_Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>1.10</td>
<td>1.20</td>
</tr>
<tr>
<td>Two</td>
<td>1.11</td>
<td>1.22</td>
</tr>
</tbody>
</table>
```

## 7.3.1 Arithmetic

Performing arithmetic with a multi-index that needs broadcasting

```
In [58]: cols = pd.MultiIndex.from_tuples([(x,y) for x in ['A','B','C'] for y in ['O','I']])
```

```
In [59]: df = pd.DataFrame(np.random.randn(2,6),index=['n','m'],columns=cols); df
```

```
Out[59]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>I</td>
<td>I</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>1.204</td>
<td>0.784</td>
<td>0.931</td>
</tr>
<tr>
<td>m</td>
<td>1.9209</td>
<td>-0.3882</td>
<td>2.314394</td>
</tr>
</tbody>
</table>
```

```
In [60]: df = df.div(df['C'],level=1); df
```

```
Out[60]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>I</td>
<td>I</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>1.204</td>
<td>0.784</td>
<td>0.931</td>
</tr>
<tr>
<td>m</td>
<td>1.9209</td>
<td>-0.3882</td>
<td>2.314394</td>
</tr>
</tbody>
</table>
```

7.3. Multilevel Indexing
7.3.2 Slicing

Slicing a multi-index with `xs`

```
In [61]: coords = [('AA','one'), ('AA','six'), ('BB','one'), ('BB','two'), ('BB','six')]
In [62]: index = pd.MultiIndex.from_tuples(coords)
In [63]: df = pd.DataFrame([11,22,33,44,55],index,['MyData']); df
Out[63]:
    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 [64]: df.xs('BB',level=0,axis=0)  #Note : level and axis are optional, and default to zero
Out[64]:
    MyData   
one     33
two     44
six     55

...and now the 2nd level of the 1st axis.
```
```
In [65]: df.xs('six',level=1,axis=0)
Out[65]:
    MyData   
AA  22
BB  55

Slicing a multi-index with `xs`, method #2

```
In [66]: index = list(itertools.product(['Ada','Quinn','Violet'],=['Comp','Math','Sci']))
In [67]: headr = list(itertools.product(['Exams','Labs'],=['I','II']))
In [68]: indx = pd.MultiIndex.from_tuples(index,names=['Student','Course'])
In [69]: cols = pd.MultiIndex.from_tuples(headr)  #Notice these are un-named
In [70]: data = [[70+x+y+(x*y)%3 for x in range(4)] for y in range(9)]
In [71]: df = pd.DataFrame(data,indx,cols); df
Out[71]:
   Exams  Labs
   460  Chapter 7. Cookbook
In [72]: All = slice(None)

In [73]: df.loc['Violet']
Out[73]:

<table>
<thead>
<tr>
<th>Exams</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td></td>
</tr>
<tr>
<td>Sci</td>
<td></td>
</tr>
</tbody>
</table>

    Exams | Labs |
    I   | I   |

<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>76</td>
<td>77</td>
</tr>
<tr>
<td>Ada</td>
<td>Math</td>
<td>77</td>
<td>79</td>
</tr>
<tr>
<td>Ada</td>
<td>Sci</td>
<td>78</td>
<td>81</td>
</tr>
<tr>
<td>Quinn</td>
<td>Comp</td>
<td>73</td>
<td>74</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>77</td>
<td>79</td>
</tr>
<tr>
<td>Quinn</td>
<td>Sci</td>
<td>78</td>
<td>81</td>
</tr>
<tr>
<td>Violet</td>
<td>Comp</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>Violet</td>
<td>Math</td>
<td>77</td>
<td>79</td>
</tr>
<tr>
<td>Violet</td>
<td>Sci</td>
<td>78</td>
<td>81</td>
</tr>
</tbody>
</table>

In [74]: df.loc[(All,'Math'),All]

In [75]: df.loc[(slice('Ada','Quinn'),'Math'),All]

In [76]: df.loc[(All,'Math'),(All,'II')]

In [77]: df.loc[(All,'Math'),(All,'II')]

---

7.3. MultiIndexing
Setting portions of a multi-index with `xs`

### 7.3.3 Sorting

Sort by specific column or an ordered list of columns, with a multi-index

```python
In [78]: df.sort_values(by=('Labs', 'II'), ascending=False)
Out[78]:
   Exams  Labs
   I  II  I  II
Student Course
Violet  Sci  78  81  81  81
       Math  77  79  81  80
       Comp  76  77  78  79
Quinn  Sci  75  78  78  78
       Math  74  76  78  77
       Comp  73  74  75  76
Ada    Sci  72  75  75  75
       Math  71  73  75  74
       Comp  70  71  72  73
```

Partial Selection, the need for sortedness;

### 7.3.4 Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

### 7.3.5 panelnd

The `panelnd` docs.

Construct a 5D `panelnd`

### 7.4 Missing Data

The `missing data` docs.

Fill forward a reversed timeseries

```python
In [79]: df = pd.DataFrame(np.random.randn(6,1), index=pd.date_range('2013-08-01', periods=6, freq='B'), columns=list('A'))
In [80]: df.loc[df.index[3], 'A'] = np.nan
In [81]: df
Out[81]:
   A
2013-08-01 -1.054874
```
In [82]: df.reindex(df.index[::-1]).ffill()

Out[82]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-08-08</td>
<td>0.104050</td>
</tr>
<tr>
<td>2013-08-07</td>
<td>1.906684</td>
</tr>
<tr>
<td>2013-08-06</td>
<td>1.906684</td>
</tr>
<tr>
<td>2013-08-05</td>
<td>0.639589</td>
</tr>
<tr>
<td>2013-08-02</td>
<td>-0.179642</td>
</tr>
<tr>
<td>2013-08-01</td>
<td>-1.054874</td>
</tr>
</tbody>
</table>

cumsum reset at NaN values

### 7.4.1 Replace

Using replace with backrefs

### 7.5 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 [83]:

df = pd.DataFrame({'animal': 'cat dog cat fish dog cat cat'.split(),
                   'size': list('SSMMMLL'),
                   'weight': [8, 10, 11, 1, 20, 12, 12],
                   'adult' : [False] * 5 + [True] * 2}); df

Out[83]:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adult</td>
<td>animal</td>
<td>size</td>
<td>weight</td>
</tr>
<tr>
<td>False</td>
<td>cat</td>
<td>S</td>
<td>8</td>
</tr>
<tr>
<td>False</td>
<td>dog</td>
<td>S</td>
<td>10</td>
</tr>
<tr>
<td>False</td>
<td>cat</td>
<td>M</td>
<td>11</td>
</tr>
<tr>
<td>False</td>
<td>fish</td>
<td>M</td>
<td>1</td>
</tr>
<tr>
<td>False</td>
<td>dog</td>
<td>M</td>
<td>20</td>
</tr>
<tr>
<td>True</td>
<td>cat</td>
<td>L</td>
<td>12</td>
</tr>
<tr>
<td>True</td>
<td>cat</td>
<td>L</td>
<td>12</td>
</tr>
</tbody>
</table>

#List the size of the animals with the highest weight.

In [84]: df.groupby('animal').apply(lambda subf: subf['size'][subf['weight'].idxmax()])

animal

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
</tr>
<tr>
<td>dog</td>
</tr>
</tbody>
</table>
Using get_group

```python
In [85]: gb = df.groupby(['animal'])
In [86]: gb.get_group('cat')
Out[86]:
    animal size  weight
0       cat   S    8
2       cat   M   11
5       cat   L   12
6       cat   L   12
```

Apply to different items in a group

```python
In [87]: 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 [88]: expected_df = gb.apply(GrowUp)
In [89]: expected_df
Out[89]:
          size  weight  adult
animal  animal
cat      L   12.4375   True
dog      L   20.0000   True
fish     L    1.2500   True
```

Expanding Apply

```python
In [90]: S = pd.Series([i / 100.0 for i in range(1,11)])
In [91]: def CumRet(x,y):
   ....:     return x * (1 + y)
   ....:
In [92]: def Red(x):
   ....:     return functools.reduce(CumRet,x,1.0)
   ....:
In [93]: S.expanding().apply(Red)
Out[93]:
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
```
Replacing some values with mean of the rest of a group

```python
In [94]: df = pd.DataFrame({'A' : [1, 1, 2, 2], 'B' : [1, -1, 1, 2]})
In [95]: gb = df.groupby('A')
In [96]:
def replace(g):
    mask = g < 0
    g.loc[mask] = g[~mask].mean()
    return g

In [97]: gb.transform(replace)
Out[97]:
   B
0  1.0
1  1.0
2  1.0
3  2.0
```

Sort groups by aggregated data

```python
In [98]: 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 [99]: code_groups = df.groupby('code')
In [100]: agg_n_sort_order = code_groups[['data']].transform(sum).sort_values(by='data')
In [101]: sorted_df = df.loc[agg_n_sort_order.index]
In [102]: sorted_df
Out[102]:
   code  data  flag
0   foo  0.16 False
2   baz  0.33 False
3   foo  0.45 True
4   bar -0.59 False
5   baz  0.62 True
```

Create multiple aggregated columns

```python
In [103]: rng = pd.date_range(start="2014-10-07", periods=10, freq='2min')
In [104]: ts = pd.Series(data = list(range(10)), index = rng)
In [105]:
def MyCust(x):
    if len(x) > 2:
        return x[1] * 1.234
    return pd.NaT
```

7.5. Grouping
In [106]: mhc = {'Mean' : np.mean, 'Max' : np.max, 'Custom' : MyCust}

In [107]: ts.resample("5min").apply(mhc)

Out[107]:

<table>
<thead>
<tr>
<th></th>
<th>2014-10-07 00:00:00</th>
<th>1.234</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2014-10-07 00:05:00</td>
<td>NaT</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:10:00</td>
<td>7.404</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:15:00</td>
<td>NaT</td>
</tr>
<tr>
<td>Custom</td>
<td>2014-10-07 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:05:00</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:10:00</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:15:00</td>
<td>8.5</td>
</tr>
</tbody>
</table>

**dtype**: object

In [108]: ts

Out[108]:

<table>
<thead>
<tr>
<th></th>
<th>2014-10-07 00:00:00</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2014-10-07 00:02:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:04:00</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:06:00</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:08:00</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:10:00</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:12:00</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:14:00</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:16:00</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2014-10-07 00:18:00</td>
<td>9</td>
</tr>
</tbody>
</table>

**Freq**: 2T, **dtype**: int64

Create a value counts column and reassign back to the DataFrame

In [109]: df = pd.DataFrame({"Color": 'Red Red Red Blue'.split(),
                      
                      'Value': [100, 150, 50, 50]})

In [110]: df['Counts'] = df.groupby(['Color']).transform(len)

In [111]: df

Out[111]:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Red</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>Red</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>Blue</td>
<td>50</td>
</tr>
</tbody>
</table>

Shift groups of the values in a column based on the index
In [112]: df = pd.DataFrame(
......:
  u'line_race': [10, 10, 8, 10, 10, 8],
  u'beyer': [99, 102, 103, 103, 88, 100],
......:
  index=[u'Last Gunfighter', u'Last Gunfighter', u'Last Gunfighter',
......:
    u'Paynter', u'Paynter', u'Paynter']), df
......:
Out[112]:
    beyer  line_race
Last Gunfighter  99  10
Last Gunfighter  102  10
Last Gunfighter  103  8
Paynter          103  10
Paynter          88  10
Paynter          100  8

In [113]: df['beyer_shifted'] = df.groupby(level=0)['beyer'].shift(1)
In [114]: df
Out[114]:
    beyer  line_race  beyer_shifted
Last Gunfighter  99  10           NaN
Last Gunfighter 102  10          99.0
Last Gunfighter 103  8        102.0
Paynter          103  10           NaN
Paynter          88  10        103.0
Paynter          100  8         88.0

Select row with maximum value from each group

In [115]: 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 [116]: mask = df.groupby(level=0).agg('idxmax')
In [117]: df_count = df.loc[mask['no']].reset_index()
In [118]: df_count
Out[118]:
      host  service  no
0     other     web  2
1      that    mail  1
2      this     mail  2

Grouping like Python's itertools.groupby

In [119]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=['A'])
In [120]: df.A.groupby((df.A != df.A.shift()).cumsum()).groups
Out[120]:
{1: Int64Index([0], dtype='int64'),
 2: Int64Index([1], dtype='int64'),
 3: Int64Index([2], dtype='int64'),
 4: Int64Index([3, 4, 5], dtype='int64'),
 5: Int64Index([6], dtype='int64'),
 6: Int64Index([7, 8], dtype='int64')}
### 7.5.1 Expanding Data

Alignment and to-date
Rolling Computation window based on values instead of counts
Rolling Mean by Time Interval

### 7.5.2 Splitting

Splitting a frame
Create a list of dataframes, split using a delineation based on logic included in rows.
7.5.3 Pivot

The *Pivot* docs.

Partial sums and subtotals

```python
In [127]: df = pd.DataFrame(data={'Province': ['ON','QC','BC','AL','AL','MN','ON'],
                               'City': ['Toronto','Montreal','Vancouver','Calgary','Edmonton','Winnipeg','Windsor'],
                               'Sales': [13,6,8,4,3,1]})

In [128]: table = pd.pivot_table(df,values=['Sales'],index=['Province'],columns=['City'],aggfunc=np.sum,margins=True)

In [129]: table.stack('City')
Out[129]:

<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>MN</td>
<td>All</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>Winnipeg</td>
<td>3.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</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></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

```python
In [130]: grades = [48,99,75,80,42,80,72,68,36,78]

In [131]: df = pd.DataFrame( {'ID': ["x%d" % r for r in range(10)],
                           'Gender': ['F', 'M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'M'],
                           'Class': ['algebra', 'stats', 'bio', 'algebra', 'algebra', 'algebra', 'stats', 'stats', 'algebra', 'algebra'],
                           'Participated': ['yes','yes','yes','yes','no','yes','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,False],
                           'Grade': grades})
```

7.5. Grouping
In [132]: 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[132]:

<table>
<thead>
<tr>
<th>ExamYear</th>
<th>Participated</th>
<th>Passed</th>
<th>Employed</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>74.000000</td>
</tr>
<tr>
<td>2008</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>68.500000</td>
</tr>
<tr>
<td>2009</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>60.666667</td>
</tr>
</tbody>
</table>

7.5.4 Apply

Rolling Apply to Organize - Turning embedded lists into a multi-index frame

In [135]: df = pd.DataFrame({'A': [[2, 4, 8, 16], [100, 200], [10, 20, 30]], 'B': [['a →', 'b', 'c'], ['jj', 'kk'], ['ccc']]), index=['I', 'II', 'III'])

In [136]: def SeriesFromSubList(aList):
       return pd.Series(aList)

In [137]: df_orgz = pd.concat(dict((ind, row.apply(SeriesFromSubList)) for ind, row in df.iterrows()))
Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

```python
In [138]: df = pd.DataFrame(data=np.random.randn(2000,2)/10000,
                   index=pd.date_range('2001-01-01',periods=2000),
                   columns=['A','B']); df
Out[138]:
     A   B
2001-01-01 0.000032 -0.000004
2001-01-02 -0.000001  0.000207
2001-01-03  0.000120 -0.000220
2001-01-04 -0.000083 -0.000165
2001-01-05 -0.000047  0.000156
2001-01-06  0.000027  0.000104
2001-01-07  0.000041 -0.000101
...        ...    ...
2006-06-17 -0.000034  0.000034
2006-06-18  0.000002  0.000166
2006-06-19 -0.000023 -0.000081
2006-06-20 -0.000061  0.000012
2006-06-21  0.000111  0.000027
2006-06-22 -0.000061 -0.000009
2006-06-23 -0.000074  0.000034
[2000 rows x 2 columns]
```

```python
In [139]: def gm(aDF,Const):
   ....:     v = ((((aDF.A+aDF.B)+1).cumprod())-1)*Const
   ....:     return (aDF.index[0],v.iloc[-1])
   ....:
In [140]: S = pd.Series(dict([ gm(df.iloc[i:min(i+51,len(df)-1)],5)
                        for i in range(len(df)-50) ])); S
Out[140]:
2001-01-01  -0.001373
2001-01-02  -0.001705
2001-01-03  -0.002885
2001-01-04  -0.002987
2001-01-05  -0.002384
2001-01-06  -0.004700
2001-01-07  -0.005550
...        ...
2006-04-28  -0.002682
2006-04-29  -0.002436
2006-04-30  -0.002602
2006-05-01  -0.001785
2006-05-02  -0.001799
2006-05-03  -0.000605
2006-05-04  -0.000541
Length: 1950, dtype: float64
```

Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)

```python
In [141]: rng = pd.date_range(start = '2014-01-01',periods = 100)
In [142]: df = pd.DataFrame({'Open' : np.random.randn(len(rng)),
                     'Close' : np.random.randn(len(rng))},
                     index=rng)
```

7.5. Grouping
```python
.....:
   'Volume' : np.random.randint(100,2000,len(rng)),
→index=rng); df
.....:

Out[142]:

<table>
<thead>
<tr>
<th></th>
<th>Close</th>
<th>Open</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.653039</td>
<td>0.011174</td>
<td>1581</td>
</tr>
<tr>
<td>1</td>
<td>1.314205</td>
<td>0.214258</td>
<td>1707</td>
</tr>
<tr>
<td>2</td>
<td>-0.341915</td>
<td>-1.046922</td>
<td>1768</td>
</tr>
<tr>
<td>3</td>
<td>-1.303586</td>
<td>-0.752902</td>
<td>836</td>
</tr>
<tr>
<td>4</td>
<td>0.396288</td>
<td>-0.410793</td>
<td>694</td>
</tr>
<tr>
<td>5</td>
<td>-0.548006</td>
<td>0.648401</td>
<td>796</td>
</tr>
<tr>
<td>6</td>
<td>0.481380</td>
<td>0.737320</td>
<td>265</td>
</tr>
<tr>
<td>7</td>
<td>-2.548128</td>
<td>0.120378</td>
<td>564</td>
</tr>
<tr>
<td>8</td>
<td>0.223346</td>
<td>0.231661</td>
<td>1908</td>
</tr>
<tr>
<td>9</td>
<td>1.228841</td>
<td>0.952664</td>
<td>1090</td>
</tr>
<tr>
<td>10</td>
<td>0.552784</td>
<td>-0.176090</td>
<td>1813</td>
</tr>
<tr>
<td>11</td>
<td>-0.795389</td>
<td>1.781318</td>
<td>1103</td>
</tr>
<tr>
<td>12</td>
<td>-0.018815</td>
<td>-0.753493</td>
<td>1456</td>
</tr>
<tr>
<td>13</td>
<td>1.138197</td>
<td>-1.047997</td>
<td>1193</td>
</tr>
</tbody>
</table>

[100 rows x 3 columns]

In [143]: def vwap(bars):

In [144]: window = 5

In [145]: s = pd.concat([pd.Series(vwap(df.iloc[i:i+window]), index=df.
→index[i+window]) for i in range(len(df)-window)]);

In [146]: s.round(2)

Out[146]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.03</td>
</tr>
<tr>
<td>1</td>
<td>0.07</td>
</tr>
<tr>
<td>2</td>
<td>-0.40</td>
</tr>
<tr>
<td>3</td>
<td>-0.81</td>
</tr>
<tr>
<td>4</td>
<td>-0.63</td>
</tr>
<tr>
<td>5</td>
<td>-0.86</td>
</tr>
<tr>
<td>6</td>
<td>-0.36</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-1.27</td>
</tr>
<tr>
<td>9</td>
<td>-1.36</td>
</tr>
<tr>
<td>10</td>
<td>-0.73</td>
</tr>
<tr>
<td>11</td>
<td>0.04</td>
</tr>
<tr>
<td>12</td>
<td>0.21</td>
</tr>
<tr>
<td>13</td>
<td>0.07</td>
</tr>
<tr>
<td>14</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Length: 95, dtype: float64
```

### 7.6 Timeseries

Between times

Using indexer between time

Constructing a datetime range that excludes weekends and includes only certain times

---

Chapter 7. Cookbook
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 [147]: dates = pd.date_range('2000-01-01', periods=5)
In [148]: dates.to_period(freq='M').to_timestamp()
Out[148]:
              '2000-01-01'],
dtype='datetime64[ns]', freq=None)
```

### 7.6.1 Resampling


Using Grouper instead of TimeGrouper for time grouping of values

Time grouping with some missing values

Valid frequency arguments to Grouper

Grouping using a MultiIndex

Using TimeGrouper and another grouping to create subgroups, then apply a custom function

Resampling with custom periods

Resample intraday frame without adding new days

Resample minute data

Resample with groupby

### 7.7 Merge


Append two dataframes with overlapping index (emulate R `rbind`)

```
In [149]: rng = pd.date_range('2000-01-01', periods=6)
In [150]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=['A', 'B', 'C'])
In [151]: df2 = df1.copy()
```

Depending on df construction, `ignore_index` may be needed

```
In [152]: df = df1.append(df2,ignore_index=True); df
Out[152]:
    A      B      C
0 -0.480676 -1.305282 -0.212846
```

473
Self Join of a DataFrame

```python
In [153]: 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)});df
Out[153]:
Area Bins Data Test_0
0 A 110 -0.378914 0
1 A 110 -1.032527 1
2 A 160 -1.402816 0
3 A 160 0.715333 1
4 A 160 -0.091438 2
5 C 40 1.608418 0
6 C 40 0.753207 1

In [154]: df['Test_1'] = df['Test_0'] - 1

In [155]: pd.merge(df, df, left_on=['Bins', 'Area','Test_0'], right_on=['Bins', 'Area →', 'Test_1'],suffixes=('L','R'))
Out[155]:
Area Bins Data_L Test_0_L Test_1_L Data_R Test_0_R Test_1_R
0 A 110 -0.378914 0 -1 -1.032527 1 0
1 A 160 -1.402816 0 -1 0.715333 1 0
2 A 160 0.715333 1 0 -0.091438 2 1
3 C 40 1.608418 0 -1 0.753207 1 0
```

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

### 7.8 Plotting

The Plotting docs.

Make Matplotlib look like R

Setting x-axis major and minor labels

Plotting multiple charts in an ipython 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 [156]: df = pd.DataFrame(
   ....:     {u'stratifying_var': np.random.uniform(0, 100, 20),
   ....:      u'price': np.random.normal(100, 5, 20)})
   ....:
In [157]: df[u'quartiles'] = pd.qcut(
   ....:     df[u'stratifying_var'],
   ....:     4,
   ....:     labels=[u'0-25%', u'25-50%', u'50-75%', u'75-100%'])
   ....:
In [158]: df.boxplot(column=u'price', by=u'quartiles')
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0x121bc60f0>
```

![Boxplot grouped by quartiles](image-url)
7.9 Data In/Out

Performance comparison of SQL vs HDF5

7.9.1 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

Dealing with bad lines II

Reading CSV with Unix timestamps and converting to local timezone

Write a multi-row index CSV without writing duplicates

7.9.1.1 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():

```python
In [159]: for i in range(3):
    ...:     data = pd.DataFrame(np.random.randn(10, 4))
    ...:     data.to_csv('file{}.csv'.format(i))
    ...:
In [160]: files = ['file_0.csv', 'file_1.csv', 'file_2.csv']
In [161]: 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:

```python
In [162]: import glob
In [163]: files = glob.glob('file*.csv')
In [164]: 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.
7.9.1.2 Parsing date components in multi-columns

Parsing date components in multi-columns is faster with a format

```python
In [30]: i = pd.date_range('20000101',periods=10000)

In [31]: df = pd.DataFrame(dict(year = i.year, month = i.month, day = i.day))

In [32]: df.head()
Out[32]:
   day   month  year
0    1      1  2000
1    2      1  2000
2    3      1  2000
3    4      1  2000
4    5      1  2000

In [33]: %timeit pd.to_datetime(df.year*10000+df.month*100+df.day,format='%Y%m%d')
100 loops, best of 3: 7.08 ms per loop

# simulate combining into a string, then parsing
In [34]: ds = df.apply(lambda x: "%04d%02d%02d" % (x['year'],x['month'],x['day']), axis=1)

In [35]: ds.head()
Out[35]:
0  20000101
1  20000102
2  20000103
3  20000104
4  20000105

In [36]: %timeit pd.to_datetime(ds)
1 loops, best of 3: 488 ms per loop
```

7.9.1.3 Skip row between header and data

```python
In [165]: data = """;
......: 
......: 
......: 
......: 
......: date;Param1;Param2;Param4;Param5
......: ;°C;m;m
......: 
......: 01.01.1990 00:00;1;1;2;3
......: 01.01.1990 01:00;5;3;4;5
......: 01.01.1990 02:00;9;5;6;7
......: 01.01.1990 03:00;13;7;8;9
......: 01.01.1990 04:00;17;9;10;11
......: 01.01.1990 05:00;21;11;12;13
```

7.9. Data In/Out
Option 1: pass rows explicitly to skiprows

```python
In [166]: pd.read_csv(StringIO(data), sep=';', skiprows=[11,12],
index_col=0, parse_dates=True, header=10)
```

```
Out[166]:
          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 [167]: pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
Out[167]:
Index(['date', 'Param1', 'Param2', 'Param4', 'Param5'], dtype='object')
In [168]: columns = pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
In [169]: pd.read_csv(StringIO(data), sep=';', index_col=0,
header=12, parse_dates=True, names=columns)
```

```
Out[169]:
          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
```

### 7.9.2 SQL

The [SQL docs](#) Reading from databases with SQL

### 7.9.3 Excel

The [Excel docs](#) Reading from a filelike handle Modifying formatting in XlsxWriter output
7.9.4 HTML

Reading HTML tables from a server that cannot handle the default request header

7.9.5 HDFStore

The HDFStores docs
Simple Queries with a Timestamp Index
Managing heterogeneous data using a linked multiple table hierarchy
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 [170]: df = pd.DataFrame(np.random.randn(8,3))
In [171]: store = pd.HDFStore('test.h5')
In [172]: store.put('df',df)
# you can store an arbitrary python object via pickle
In [173]: store.get_storer('df').attrs.my_attribute = dict(A = 10)
In [174]: store.get_storer('df').attrs.my_attribute
Out[174]: {'A': 10}
```

7.9.6 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
from __future__ import division
import os, sys
import numpy as np
import pandas as pd

# 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 msgpack, both of which are supported by pandas’ IO facilities.

### 7.10 Computation

Numerical integration (sample-based) of a time series
7.11 Timedeltas

The Timedeltas docs.

Using timedeltas

In [175]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [176]: s - s.max()
Out[176]:
0    -2 days
1    -1 days
2     0 days
dtype: timedelta64[ns]

In [177]: s.max() - s

Out[177]:
0    2 days
1    1 days
2     0 days
dtype: timedelta64[ns]

In [178]: s - datetime.datetime(2011,1,1,3,5)

Out[178]:
0 364 days 20:55:00
1 365 days 20:55:00
2 366 days 20:55:00
dtype: timedelta64[ns]

In [179]: s + datetime.timedelta(minutes=5)

Out[179]:
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 [180]: datetime.datetime(2011,1,1,3,5) - s

Out[180]:
0 -365 days +03:05:00
1 -366 days +03:05:00
2 -367 days +03:05:00
dtype: timedelta64[ns]

In [181]: datetime.timedelta(minutes=5) + s

Out[181]:
0 2012-01-01 00:05:00
1 2012-01-02 00:05:00
2 2012-01-03 00:05:00
dtype: datetime64[ns]

Adding and subtracting deltas and dates

In [182]: deltas = pd.Series([datetime.timedelta(days=i) for i in range(3)])
Another example

Values can be set to NaT using np.nan, similar to datetime

7.12 Aliasing Axis Names

To globally provide aliases for axis names, one can define these 2 functions:
In [191]: set_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

In [192]: df2 = pd.DataFrame(np.random.randn(3,2), columns=['c1', 'c2'], index=['i1', 'i2', 'i3'])

In [193]: df2.sum(axis='myaxis2')
Out[193]:
    i1  0.745167
    i2 -0.176251
    i3  0.014354
dtype: float64

In [194]: clear_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

### 7.13 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:

```
In [195]: def expand_grid(data_dict):
    ...:     rows = itertools.product(*data_dict.values())
    ...:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
    ...:

In [196]: df = expand_grid({'height': [60, 70],
                      'weight': [100, 140, 180],
                      'sex': ['Male', 'Female']})

In [197]: df
Out[197]:
     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
```
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.

## 8.1 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:

```
>>> 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]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [4]: s
Out[4]:
a    0.4691
b   -0.2829
c   -1.5091
```

```
In [5]: s.index

Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))

Out[6]:
0   -0.1732
1    0.1192
2   -1.0442
3   -0.8618
4   -2.1046
Name: 0, 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

If data is a dict, if index is passed the values in data corresponding to the labels in the index will be pulled out. Otherwise, an index will be constructed from the sorted keys of the dict, if possible.

In [7]: d = {'a': 0., 'b': 1., 'c': 2.}

In [8]: pd.Series(d)

Out[8]:
  a    0.0
  b    1.0
  c    2.0
dtype: float64

In [9]: pd.Series(d, index=['b', 'c', 'd', 'a'])

Out[9]:
  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 [10]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])

Out[10]:
  a    5.0
  b    5.0
  c    5.0
8.1.1 Series is ndarray-like

Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions. However, things like slicing also slice the index.

```
In [11]: s[0]
Out[11]: 0.46911229990718628

In [12]: s[:3]
Out[12]:
    a  0.4691
    b -0.2829
    c -1.5091

In [13]: s[s > s.median()]
Out[13]:
        a  0.4691
        e 1.2121

In [14]: s[[4, 3, 1]]
Out[14]:
        e  1.2121
        d -1.1356
        b -0.2829

In [15]: np.exp(s)
Out[15]:
        a  1.5986
        b  0.7536
        c  0.2211
        d  0.3212
        e  3.3606
```

We will address array-based indexing in a separate section.

8.1.2 Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [16]: s['a']
Out[16]: 0.46911229990718628

In [17]: s['e'] = 12.
```

8.1. Series
In [18]: s
Out[18]:
a  0.4691  
b -0.2829  
c -1.5091  
d -1.1356  
e  12.0000  
dtype: float64

In [19]: 'e' in s
Out[19]: True

In [20]: 'f' in s
Out[20]: False

If a label is not contained, an exception is raised:

```python
>>> s['f']
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```python
In [21]: s.get('f')
In [22]: s.get('f', np.nan)
Out[22]: nan
```

See also the section on attribute access.

### 8.1.3 Vectorized operations and label alignment with Series

When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usually not necessary. Series can also be passed into most NumPy methods expecting an ndarray.

```python
In [23]: s + s
Out[23]:
a  0.9382  
b -0.5657  
c -3.0181  
d -2.2713  
e  24.0000  
dtype: float64

In [24]: s * 2
Out[24]:
a  0.9382  
b -0.5657  
c -3.0181  
d -2.2713  
e  24.0000  
dtype: float64

In [25]: np.exp(s)
```
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.

```
In [26]: s[1:] + s[:-1]
Out[26]:
  a   NaN
  b -0.5657
  c  3.0181
  d -2.2713
  e   NaN
dtype: float64
```

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.

### 8.1.4 Name attribute

Series can also have a **name** attribute:

```
In [27]: s = pd.Series(np.random.randn(5), name='something')
In [28]: s
Out[28]:
0   -0.4949
1    1.0718
2    0.7216
3   -0.7068
4   -1.0396
Name: something, dtype: float64
```

```
In [29]: s.name
Out[29]: 'something'
```

The Series **name** will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.
New in version 0.18.0.

You can rename a Series with the `pandas.Series.rename()` method.

```python
In [30]: s2 = s.rename("different")
In [31]: s2.name
Out[31]: 'different'
```

Note that `s` and `s2` refer to different objects.

## 8.2 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.

### 8.2.1 From dict of Series or dicts

The result `index` will be the union of the indexes of the various Series. If there are any nested dicts, these will be first converted to Series. If no columns are passed, the columns will be the sorted list of dict keys.

```python
In [32]: d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
    ....:     'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
    ....:
In [33]: df = pd.DataFrame(d)
In [34]: df
Out[34]:
     one  two
    a   1.0  1.0
    b   2.0  2.0
    c   3.0  3.0
    d   NaN  4.0

In [35]: pd.DataFrame(d, index=['d', 'b', 'a'])
```

```
Out[35]:
     one  two
    d   NaN  4.0
    b   2.0  2.0
    a   1.0  1.0
```
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.

8.2.2 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.

8.2.3 From structured or record array

This case is handled identically to a dict of arrays.

8.2. DataFrame
In [44]: pd.DataFrame(data)
Out[44]:
    A   B   C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [45]: pd.DataFrame(data, index=['first', 'second'])
    A   B   C
first 1  2.0  b'Hello'
second 2  3.0  b'World'

In [46]: pd.DataFrame(data, columns=['C', 'A', 'B'])
    C   A   B
0  b'Hello' 1  2.0
1  b'World' 2  3.0

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

8.2.4 From a list of dicts

In [47]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [48]: pd.DataFrame(data2)
Out[48]:
     a   b   c
0  1.0  2.0  NaN
1  5.0 10.0 20.0

In [49]: pd.DataFrame(data2, index=['first', 'second'])
    a   b   c
first 1  2.0  NaN
second 5 10.0 20.0

In [50]: pd.DataFrame(data2, columns=['a', 'b'])
   →
    a   b
0  1  2.0
1  5 10.0

8.2.5 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary

In [51]: 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},}
8.2.6 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).

Missing Data

Much more will be said on this topic in the Missing data section. To construct a DataFrame with missing data, use np.nan for those values which are missing. Alternatively, you may pass a numpy.MaskedArray as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

8.2.7 Alternate Constructors

**DataFrame.from_dict**

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 orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

 DataFrame.from_records

DataFrame.from_records takes a list of tuples or an ndarray with structured dtype. Works analogously to the normal DataFrame constructor, except that index maybe be a specific field of the structured dtype to use as the index. For example:

```
In [52]: data
Out[52]: array([(1, 2., b'Hello'), (2, 3., b'World')],
               dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

```
In [53]: pd.DataFrame.from_records(data, index='C')
     →
     A  B
C b'Hello' 1  2.0
b'World' 2  3.0
```

**DataFrame.from_items**

DataFrame.from_items works analogously to the form of the dict constructor that takes a sequence of (key, value) pairs, where the keys are column (or row, in the case of orient='index') names, and the value are the column values (or row values). This can be useful for constructing a DataFrame with the columns in a particular order without having to pass an explicit list of columns:

```
In [54]: pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
Out[54]:
     A  B
   →
C  1  2  3
D  4  5  6
```
If you pass `orient='index'`, the keys will be the row labels. But in this case you must also pass the desired column names:

```python
In [55]: pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
                             orient='index', columns=['one', 'two', 'three'])
Out[55]:
     one  two  three
A   1.0  2.0  3.0
B   4.0  5.0  6.0
```

### 8.2.8 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:

```python
In [56]: df['one']
Out[56]:
     a    b   c   NaN
Name: one, dtype: float64

In [57]: df['three'] = df['one'] * df['two']
In [58]: df['flag'] = df['one'] > 2
In [59]: df
Out[59]:
     one  two  three  flag
     a    b   c    NaN
    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 a dict:

```python
In [60]: del df['two']
In [61]: three = df.pop('three')
In [62]: df
Out[62]:
     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:
When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:

```python
In [65]: df['one_trunc'] = df['one'][:2]
In [66]: df
Out[66]:
   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:

```python
In [67]: df.insert(1, 'bar', df['one'])
In [68]: df
Out[68]:
   one  bar  flag  foo one_trunc
  a  1.0  1.0  False  bar  1.0
  b  2.0  2.0  False  bar  2.0
  c  3.0  3.0   True  bar  NaN
  d  NaN  NaN  False  bar  NaN
```

### 8.2.9 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.

```python
In [69]: iris = pd.read_csv('data/iris.data')
In [70]: iris.head()
Out[70]:
   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 [71]: (iris.assign(sepal_ratio = iris['SepalWidth'] / iris['SepalLength'])
......:     .head())
```
Above was an example of inserting a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```python
In [72]: iris.assign(sepal_ratio = lambda x: x['SepalWidth'] / x['SepalLength']).head()
```

<table>
<thead>
<tr>
<th>SepalLength</th>
<th>SepalWidth</th>
<th>PetalLength</th>
<th>PetalWidth</th>
<th>Name</th>
<th>sepal_ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>0.6863</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>0.6122</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>0.6809</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>0.6739</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>0.7200</td>
</tr>
</tbody>
</table>

The `assign` method 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 chains 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 [73]: (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[73]: <matplotlib.axes._subplots.AxesSubplot at 0x122d7fba8>
```
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.

**Warning:** Since the function signature of `assign` is `**kwargs`, a dictionary, the order of the new columns in the resulting DataFrame cannot be guaranteed to match the order you pass in. To make things predictable, items are inserted alphabetically (by key) at the end of the DataFrame.

All expressions are computed first, and then assigned. So you can’t refer to another column being assigned in the same call to `assign`. For example:

```python
In [74]: # Don't do this, bad reference to 'C'
   ...: df.assign(C = lambda x: x['A'] + x['B'],
   ...:             D = lambda x: x['A'] + x['C'])
In [2]: # Instead, break it into two assigns
   ...: (df.assign(C = lambda x: x['A'] + x['B'])
   ...:     .assign(D = lambda x: x['A'] + x['C']))
```

### 8.2.10 Indexing / Selection

The basics of indexing are as follows:
8.2.11 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.

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 the special case of working with time series data, and the DataFrame index also contains dates, the broadcasting will be column-wise:

```
In [81]: index = pd.date_range('1/1/2000', periods=8)
In [82]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=list('ABC'))
In [83]: df
Out[83]:
   A       B       C
2000-01-01 -1.2268  0.7698  1.2812
2000-01-02 -0.7277 -0.1213 -0.0979
2000-01-03  0.6958  0.3417  0.9597
2000-01-04 -1.1103 -0.6200  0.1497
2000-01-05 -0.7323  0.6877  0.1764
2000-01-06  0.4033 -0.1550  0.3016
2000-01-07 -2.1799 -1.3698 -0.9542
2000-01-08  1.4627 -1.7432 -0.8266
```

```
In [84]: type(df['A'])
Out[84]: pandas.core.series.Series
```

```
In [85]: df - df['A']
```

```
The resulting DataFrame is:
```

```
2000-01-01
2000-01-02
2000-01-03
2000-01-04
2000-01-05
2000-01-06
2000-01-07
2000-01-08
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Warning:

df - df['A']

is now deprecated and will be removed in a future release. The preferred way to replicate this behavior is
df.sub(df['A'], axis=0)

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 [86]: df * 5 + 2
Out[86]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-4.1341</td>
<td>5.8490</td>
<td>-4.4062</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-1.6385</td>
<td>1.3935</td>
<td>1.5106</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>5.4789</td>
<td>3.7087</td>
<td>6.7986</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-3.5517</td>
<td>-1.0999</td>
<td>2.7487</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.6617</td>
<td>5.4387</td>
<td>2.8822</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>4.0165</td>
<td>1.2252</td>
<td>3.5081</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-8.8993</td>
<td>-4.8492</td>
<td>-2.7710</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>9.3135</td>
<td>-6.7158</td>
<td>-2.1330</td>
</tr>
</tbody>
</table>

In [87]: 1 / df

In [88]: df ** 4

Boolean operators work as well:
In [89]: df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1]}, dtype=bool)

In [90]: df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0]}, dtype=bool)

In [91]: df1 & df2
Out[91]:
      a  b
0  False  False
1  False   True
2   True  False

In [92]: df1 | df2
Out[92]:
      a  b
0   True   True
1   True   True
2   True   True

In [93]: df1 ^ df2
     a  b
0   True   True
1   True  False
2  False   True

In [94]: -df1
     a  b
0  False   True
1   True  False
2  False  False

8.2.12 Transposing
To transpose, access the T attribute (also the transpose function), similar to an ndarray:

# only show the first 5 rows
In [95]: df[:5].T
Out[95]:
A    -1.2268    -0.7277     0.6958    -1.1103    -0.7323
B    0.7698    -0.1213     0.3417    -0.6200     0.6877
C   -1.2812    -0.0979     0.9597     0.1497     0.1764

8.2.13 DataFrame interoperability with NumPy functions
Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on DataFrame, assuming the data within are numeric:

In [96]: np.exp(df)
Out[96]:
   A    B    C
0  2.719  2.719  2.719
1  2.718  2.718  2.718
2  2.719  2.719  2.719

8.2. DataFrame
In [97]: np.asarray(df)

Out[97]:
\[\begin{array}{ccc}
-1.2268 & 0.7698 & -1.2812 \\
-0.7277 & -0.1213 & -0.0979 \\
0.6958 & 0.3417 & 0.9597 \\
-1.1103 & -0.62 & 0.1497 \\
-0.7323 & 0.6877 & 0.1764 \\
0.4033 & -0.155 & 0.3016 \\
-2.1799 & -1.3698 & -0.9542 \\
1.4627 & -1.7432 & -0.8266 \\
\end{array}\]

The dot method on DataFrame implements matrix multiplication:

In [98]: df.T.dot(df)

Out[98]:
\begin{array}{ccc}
A & B & C \\
\hline
A & 11.3419 & -0.0598 & 3.0080 \\
B & -0.0598 & 6.5206 & 2.0833 \\
C & 3.0080 & 2.0833 & 4.3105 \\
\end{array}

Similarly, the dot method on Series implements dot product:

In [99]: s1 = pd.Series(np.arange(5,10))

In [100]: s1.dot(s1)

Out[100]: 255

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics are quite different in places from a matrix.

### 8.2.14 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 [101]: baseball = pd.read_csv('data/baseball.csv')

In [102]: print(baseball)

<table>
<thead>
<tr>
<th>id</th>
<th>player</th>
<th>year</th>
<th>stint</th>
<th>...</th>
<th>hbp</th>
<th>sh</th>
<th>sf</th>
<th>gidp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>womacto01</td>
<td>2006</td>
<td>2</td>
<td>...</td>
<td>0.0</td>
<td>3.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>schilcu01</td>
<td>2006</td>
<td>1</td>
<td>...</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>98</td>
<td>alouno01</td>
<td>2007</td>
<td>1</td>
<td>...</td>
<td>2.0</td>
<td>0.0</td>
<td>3.0</td>
<td>13.0</td>
</tr>
<tr>
<td>99</td>
<td>alomasa02</td>
<td>2007</td>
<td>1</td>
<td>...</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

[100 rows x 23 columns]
In [103]: baseball.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
   id    100 non-null int64
   player 100 non-null object
   year  100 non-null int64
   stint 100 non-null int64
   team   100 non-null object
   lg     100 non-null object
   g      100 non-null int64
   ab     100 non-null int64
   r      100 non-null int64
   h      100 non-null int64
   X2b    100 non-null int64
   X3b    100 non-null int64
   hr     100 non-null int64
   rbi    100 non-null float64
   sb     100 non-null float64
   cs     100 non-null float64
   bb     100 non-null int64
   so     100 non-null float64
   ibb    100 non-null float64
   hbp    100 non-null float64
   sh     100 non-null float64
   sf     100 non-null float64
   gidp   100 non-null float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.0+ KB
```

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 [104]: print(baseball.iloc[-20:, :12].to_string())

```
  id  player  year  stint  team  lg  g  ab  r  h  X2b  X3b
--- --- ------- ------ ------ --- --- --- --- --- --- ---
80  89474 finlest01 2007 1 COL NL  43  94 9 17  3  0
81  89480 embreal01 2007 1 OAK AL  4  0 0  0  0  0
82  89481 edmonji01 2007 1 SLN NL 117 365 92 15  2
83  89482 easleda01 2007 1 NYN NL  76 193 24  6
84  89489 delgaca01 2007 1 NYN NL 139 538 71  3
85  89493 cormirh01 2007 1 CIN NL  6  0 0  0
86  89494 coninje01 2007 1 NYN NL  41  2  8  2
87  89495 coninje01 2007 1 CIN NL 215  3 57 11
88  89497 clemero02 2007 1 NYA AL  2  0  0
89  89498 claytro01 2007 1 BOS AL  8  6  0
90  89499 claytro01 2007 1 TOR AL  9  8  4
91  89501 cirilje01 2007 2 ARI NL  8  6  0
92  89502 cirilje01 2007 1 MIN AL  5  3  2
93  89521 bondsba01 2007 1 SFN NL 126  7  5  4
94  89523 biggicr01 2007 1 HOU NL 141  7 31
95  89525 benitar01 2007 2 FLO NL  3  0
96  89526 benitar01 2007 1 SFN NL  2  0
97  89530 ausmubr01 2007 1 HOU NL 349  38
98  89533 aloumo01 2007 1 NYN NL 87  3  1
99  89534alomasa02 2007 1 NYN NL  8  2  0
```
Wide DataFrames will be printed across multiple rows by default:

```python
In [105]: pd.DataFrame(np.random.randn(3, 12))
Out[105]:
   0    1    2    3    4    5    6
0 -0.345352 1.314232 0.690579 0.995761 2.396780 0.014871 3.357427
1 -2.182937 0.380396 0.084844 0.432390 1.519970 -0.493662 0.600178
2  0.206053 -0.251905 -2.213588 1.063327 1.266143 0.299368 -0.863838
   7    8    9   10   11
0 -0.317441 -1.236269 0.896171 -0.487602 -0.082240
1  0.274230 0.132885 -0.023688 2.410179 1.450520
2  0.408204 -1.048089 -0.025747 -0.988387 0.094055
```

You can change how much to print on a single row by setting the `display.width` option:

```python
In [106]: pd.set_option('display.width', 40)  # default is 80
In [107]: pd.DataFrame(np.random.randn(3, 12))
Out[107]:
   0   1   2
0 1.262731 1.289997 0.082423
1 1.126203 -0.977349 1.474071
2 0.758527 1.729689 -0.964980
   3   4   5
0 -0.055758 0.536580 -0.489682
1 -0.064034 -1.282782 0.781836
2 -0.845696 -1.340896 1.846883
   6   7   8
0 0.369374 -0.034571 -2.484478
1 -1.071357 0.441153 2.353925
2 -1.328865 1.682706 -1.717693
   9  10  11
0 -0.281461 0.030711 0.109121
1 0.583787 0.221471 -0.744471
2 0.888782 0.228440 0.901805
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`:

```python
In [108]: datafile={'filename': ['filename_01','filename_02'],'path': ['media/user_name/storage/folder_01/filename_01','media/user_name/storage/folder_02/filename_02']}
In [109]: pd.set_option('display.max_colwidth',30)
In [110]: pd.DataFrame(datafile)
Out[110]:
   filename
0  filename_01
1  filename_02
   path
0  media/user_name/storage/fo...
1  media/user_name/storage/fo...
```

Chapter 8. Intro to Data Structures
In [111]: pd.set_option('display.max_colwidth',100)

In [112]: pd.DataFrame(datafile)
Out[112]:
   filename  
0  filename_01
1  filename_02

   path
0  media/user_name/storage/folder_01/filename_01
1  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.

### 8.2.15 DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like attributes:

```
In [113]: df = pd.DataFrame({'foo1' : np.random.randn(5),
                      'foo2' : np.random.randn(5)})

In [114]: df
Out[114]:
   foo1   foo2
0  1.1712  -0.8584
1  0.5203  0.3070
2 -1.1971  -0.0287
3 -1.0670  0.3843
4 -0.3034  1.5742

In [115]: df.foo1
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.fo<TAB>
df.foo1  df.foo2
```

### 8.3 Panel

**Warning**: In 0.20.0, `Panel` is deprecated and will be removed in a future version. See the section *Deprecate Panel*.
Panel is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

- **items**: axis 0, each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1, it is the index (rows) of each of the DataFrames
- **minor_axis**: axis 2, it is the columns of each of the DataFrames

Construction of Panels works about like you would expect:

### 8.3.1 From 3D ndarray with optional axis labels

```python
In [116]: 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 [117]: wp
Out[117]:
<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
```

### 8.3.2 From dict of DataFrame objects

```python
In [118]: data = {'Item1': pd.DataFrame(np.random.randn(4, 3)),
      'Item2': pd.DataFrame(np.random.randn(4, 2))}

In [119]: pd.Panel(data)
Out[119]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

Note that the values in the dict need only be convertible to DataFrame. Thus, they can be any of the other valid inputs to DataFrame as per above.

One helpful factory method is `Panel.from_dict`, which takes a dictionary of DataFrames as above, and the following named parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersect</td>
<td>False</td>
<td>drops elements whose indices do not align</td>
</tr>
<tr>
<td>orient</td>
<td>items</td>
<td>use minor to use DataFrames’ columns as panel items</td>
</tr>
</tbody>
</table>

For example, compare to the construction above:
Orient is especially useful for mixed-type DataFrames. If you pass a dict of DataFrame objects with mixed-type columns, all of the data will get upcasted to \texttt{dtype=object} unless you pass \texttt{orient='minor'}:

```python
In [121]: df = pd.DataFrame({'a': ['foo', 'bar', 'baz'],

.....:       'b': np.random.randn(3))
.....:

In [122]: df
Out[122]:
  a   b
0 foo -0.308853
1 bar -0.681087
2 baz  0.377953

In [123]: data = {'item1': df, 'item2': df}

In [124]: panel = pd.Panel.from_dict(data, orient='minor')

In [125]: panel['a']
Out[125]:
    item1  item2
0   foo      foo
1   bar      bar
2   baz      baz

In [126]: panel['b']
Out[126]:
    item1  item2
0 -0.308853 -0.308853
1 -0.681087 -0.681087
2  0.377953  0.377953

In [127]: panel['b'].dtypes
Out[127]:
    item1    item2
dtype: object
```

Note: Panel, being less commonly used than Series and DataFrame, has been slightly neglected feature-wise. A number of methods and options available in DataFrame are not available in Panel.

**8.3.3 From DataFrame using \texttt{to\_panel} method**

to\_panel converts a DataFrame with a two-level index to a Panel.
8.3.4 Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

```
In [131]: wp['Item1']
Out[131]:
   A     B     C     D
2000-01-01 1.588931 0.476720 0.473424 -0.242861
2000-01-02 -0.014805 -0.284319 0.650776 -1.461665
2000-01-03 -1.137707 -0.891060 -0.693921 1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838
```

```
In [132]: wp['Item3'] = wp['Item1'] / wp['Item2']
```

The API for insertion and deletion is the same as for DataFrame. And as with DataFrame, if the item is a valid python identifier, you can access it as an attribute and tab-complete it in IPython.

8.3.5 Transposing

A Panel can be rearranged using its `transpose` method (which does not make a copy by default unless the data are heterogeneous):

```
In [133]: wp.transpose(2, 0, 1)
Out[133]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 5 (minor_axis)
Items axis: A to D
Major_axis axis: Item1 to Item3
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

8.3.6 Indexing / Selection

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select item</td>
<td><code>wp[item]</code></td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at major_axis label</td>
<td><code>wp.major_xs(val)</code></td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at minor_axis label</td>
<td><code>wp.minor_xs(val)</code></td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

For example, using the earlier example data, we could do:
8.3.7 Squeezing

Another way to change the dimensionality of an object is to squeeze a 1-len object, similar to `wp['Item1']`

```
In [134]: wp['Item1']
Out[134]:
                     A      B      C      D
2000-01-01  1.588893  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707  0.891060  0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819  0.260838
```

```
In [135]: wp.major_xs(wp.major_axis[2])

         Item1  Item2  Item3
A -1.137707  0.800193 -1.421791
B -0.891060  0.782098 -1.139320
C -0.693921 -1.069094  0.649074
D  1.613616 -1.099248 -1.467927
```

```
In [136]: wp.minor_axis

Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
In [137]: wp.minor_xs('C')

         Item1  Item2  Item3
2000-01-01  0.473424 -0.902937 -0.524316
2000-01-02  0.650776 -1.144073 -0.568824
2000-01-03 -0.693921 -1.069094  0.649074
2000-01-04 -0.496922  0.661084 -0.751678
2000-01-05  1.561819 -1.056652  1.478083
```

```
In [138]: wp.reindex(items=['Item1']).squeeze()
Out[138]:
                     A      B      C      D
2000-01-01  1.588893  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707  0.891060  0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819  0.260838
```

```
In [139]: wp.reindex(items=['Item1'], minor=['B']).squeeze()
```

```
Freq: D, Name: B, dtype: float64
```
8.3.8 Conversion to DataFrame

A Panel can be represented in 2D form as a hierarchically indexed DataFrame. See the section hierarchical indexing for more on this. To convert a Panel to a DataFrame, use the to_frame method:

```
In [140]: panel = pd.Panel(np.random.randn(3, 5, 4),
                       items=['one', 'two', 'three'],
                       major_axis=pd.date_range('1/1/2000', periods=5),
                       minor_axis=['a', 'b', 'c', 'd'])

In [141]: panel.to_frame()
```

```
Out[141]:
     one   two   three
major minor
2000-01-01 a  0.493672  1.219492 -1.290493
      b -2.461467  0.062297  0.787872
      c -1.553902 -0.110388  1.515707
      d  2.015523 -1.184357 -0.276487
2000-01-02 a -1.833722 -0.558081 -0.223762
      b  1.771740  0.077849  1.397431
      c -0.670027  0.629498  1.503874
      d  0.049307 -1.035260 -0.478905
2000-01-03 a -0.521493 -0.438229 -0.135950
      b -3.201750  0.503703 -0.730327
      c  0.792716  0.413086 -0.033277
      d  0.146111 -1.139050  0.281151
2000-01-04 a  1.903247  0.660342 -1.298915
      b -0.747169  0.464794 -2.819487
      c -0.309038 -0.309337 -0.851985
      d  0.393876 -0.649593 -1.106952
2000-01-05 a  1.861468  0.683758 -0.937731
      b  0.936527 -0.643834 -1.537770
      c  1.255746  0.421287  0.557595
      d -2.655452  1.032814 -2.277282
```

8.4 Deprecate Panel

Over the last few years, pandas has increased in both breadth and depth, with new features, datatype support, and manipulation routines. As a result, supporting efficient indexing and functional routines for `Series`, `DataFrame` and `Panel` has contributed to an increasingly fragmented and difficult-to-understand codebase.

The 3-D structure of a `Panel` is much less common for many types of data analysis, than the 1-D of the `Series` or the 2-D of the `DataFrame`. Going forward it makes sense for pandas to focus on these areas exclusively.

Oftentimes, one can simply use a MultiIndex `DataFrame` for easily working with higher dimensional data.

In addition, the `xarray` package was built from the ground up, specifically in order to support the multi-dimensional analysis that is one of `Panel`'s main usecases. Here is a link to the `xarray` panel-transition documentation.

```
In [142]: p = tm.makePanel()

In [143]: p
Out[143]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 30 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
```
Convert to a MultiIndex DataFrame

<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>-0.390201</td>
<td>-1.624062</td>
<td>-0.605044</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.562443</td>
<td>0.483103</td>
<td>0.583129</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.085663</td>
<td>0.768159</td>
<td>-0.273458</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.136235</td>
<td>-0.021763</td>
<td>-0.700648</td>
</tr>
<tr>
<td>2000-01-04 A</td>
<td>1.207122</td>
<td>-0.758514</td>
<td>0.878404</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.763264</td>
<td>0.061495</td>
<td>-0.876690</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.114738</td>
<td>0.225441</td>
<td>-0.335117</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.886313</td>
<td>-0.047152</td>
<td>-1.166607</td>
</tr>
<tr>
<td>2000-01-05 A</td>
<td>0.178690</td>
<td>-0.560859</td>
<td>-0.921485</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.162027</td>
<td>0.240767</td>
<td>-1.919354</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.058216</td>
<td>0.543294</td>
<td>-0.476268</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-1.350722</td>
<td>0.088472</td>
<td>-0.367236</td>
</tr>
<tr>
<td>2000-01-06 A</td>
<td>-1.004168</td>
<td>-0.589005</td>
<td>-0.200312</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.902704</td>
<td>0.782413</td>
<td>-0.572707</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.486768</td>
<td>0.771931</td>
<td>-1.765602</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.886348</td>
<td>-0.857435</td>
<td>1.296674</td>
</tr>
<tr>
<td>2000-01-07 A</td>
<td>-1.377627</td>
<td>-1.070678</td>
<td>0.522423</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.106010</td>
<td>0.628462</td>
<td>-1.736484</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1.685148</td>
<td>-0.968145</td>
<td>0.578223</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-1.013316</td>
<td>-2.503786</td>
<td>0.641385</td>
</tr>
<tr>
<td>2000-01-10 A</td>
<td>0.499281</td>
<td>-1.681101</td>
<td>0.722511</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.199234</td>
<td>-0.880627</td>
<td>-1.335113</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.112572</td>
<td>-1.176383</td>
<td>0.242697</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>1.920906</td>
<td>-1.058041</td>
<td>-0.779432</td>
</tr>
<tr>
<td>2000-01-11 A</td>
<td>-1.405256</td>
<td>0.403776</td>
<td>-1.702486</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.458265</td>
<td>0.777575</td>
<td>-1.244471</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.495309</td>
<td>-3.192716</td>
<td>0.208129</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.388231</td>
<td>-0.657981</td>
<td>0.602456</td>
</tr>
<tr>
<td>2000-01-12 A</td>
<td>0.162565</td>
<td>0.609862</td>
<td>-0.709535</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.491048</td>
<td>-0.779367</td>
<td>0.347339</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000-02-02 C</td>
<td>-0.303961</td>
<td>-0.463752</td>
<td>-0.288962</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.104050</td>
<td>1.116086</td>
<td>0.506445</td>
</tr>
<tr>
<td>2000-02-03 A</td>
<td>-2.338595</td>
<td>-0.581967</td>
<td>-0.801820</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.557697</td>
<td>-0.033731</td>
<td>-0.176382</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.625555</td>
<td>-0.055289</td>
<td>0.875359</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.174068</td>
<td>-0.443915</td>
<td>1.626369</td>
</tr>
<tr>
<td>2000-02-04 A</td>
<td>-0.374279</td>
<td>-1.233862</td>
<td>-0.915751</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.381353</td>
<td>-1.108761</td>
<td>-1.970108</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.059268</td>
<td>-0.360853</td>
<td>-0.614618</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.439461</td>
<td>-0.200491</td>
<td>0.429518</td>
</tr>
<tr>
<td>2000-02-07 A</td>
<td>-2.359958</td>
<td>-3.520876</td>
<td>-0.288156</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.337122</td>
<td>-0.314399</td>
<td>-1.044208</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.249698</td>
<td>0.728197</td>
<td>0.565375</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.741343</td>
<td>1.092633</td>
<td>0.013910</td>
</tr>
<tr>
<td>2000-02-08 A</td>
<td>-1.157886</td>
<td>0.516870</td>
<td>-1.199945</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-1.531095</td>
<td>-0.860626</td>
<td>-0.821179</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1.103949</td>
<td>1.326768</td>
<td>0.068184</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.079673</td>
<td>-1.675194</td>
<td>-0.458272</td>
</tr>
</tbody>
</table>
Alternatively, one can convert to an xarray DataArray.

```python
In [145]: p.to_xarray()
Out[145]:
<xarray.DataArray (items: 3, major_axis: 30, minor_axis: 4)>
array([[-0.390201, 1.562443, -1.085663, 0.136235],
       [ 1.207122, 0.763264, -1.114738, 0.886313],
       [-1.526739, -0.571329, 1.998044, 0.303638],
       [ 1.559318, -0.026671, -0.244548, -0.917368],
       [ 1.2137827, -1.761442, 0.292058, 0.388254],
       [ 0.452429, -0.899454, -2.01961 , 0.47963 ],
       [ 1.505044 , 0.583129, -0.273458, -0.700648],
       [ 0.878404 , 0.87669 , -0.335117, -1.166607],
       [-1.82874, -0.826439, -0.280343, -0.500569],
       [-1.716981, 0.124808, 0.931536, 0.87069 ]]),
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'
```

You can see the full-documentation for the xarray package.

### 8.5 Panel4D and PanelND (Deprecated)

**Warning:** In 0.19.0 Panel4D and PanelND 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.

See the docs of a previous version for documentation on these objects.
Here we discuss a lot of the essential functionality common to the pandas data structures. Here’s how to create some of the objects used in the examples from the previous 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'])
In [4]: 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'])
```

### 9.1 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 [5]: long_series = pd.Series(np.random.randn(1000))
In [6]: long_series.head()
Out[6]:
0   0.229453
1   0.304418
2   0.736135
3  -0.859631
4  -0.424100
dtype: float64
In [7]: long_series.tail(3)
Out[7]:
997  -0.351587
998   1.136249
999  -0.448789
dtype: float64
```
9.2 Attributes and the raw ndarray(s)

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`
  - **Panel**: `items`, `major_axis`, and `minor_axis`

Note, these attributes can be safely assigned to!

```python
In [8]: df[:2]
Out[8]:
   A    B    C
0  2000-01-01  0.048869 -1.360687 -0.479010
1  2000-01-02  -0.859661 -0.231595 -0.527750

In [9]: df.columns = [x.lower() for x in df.columns]
In [10]: df
Out[10]:
   a    b    c
0  2000-01-01  0.048869 -1.360687 -0.479010
1  2000-01-02  -0.859661 -0.231595 -0.527750
2  2000-01-03  -1.296337  0.150680  0.123836
3  2000-01-04   0.571764  1.555563 -0.823761
4  2000-01-05   0.535420 -1.032853  1.469725
5  2000-01-06   1.304124  1.449735  0.203109
6  2000-01-07  -1.032011  0.969818  -0.962723
7  2000-01-08   1.382083  -0.938794  0.669142
```

To get the actual data inside a data structure, one need only access the **values** property:

```python
In [11]: s.values
Out[11]: array([-1.9339, 0.3773, 0.7341, 2.1416, -0.0112])

In [12]: df.values
Out[12]:

   0.0489 -1.3607 -0.479
   -0.8597 -0.2316 -0.5278
   -1.2963  0.1507  0.1238
   0.5718  1.5556 -0.8238
   0.5354 -1.0329  1.4697
   1.3041  1.4497  0.2031
   -1.032  0.9698 -0.9627
   1.3821 -0.9388  0.6691

In [13]: wp.values
array([[-0.4336, -0.2736, 0.6804, -0.3084],
       [-0.2761, -1.8212, -1.9936, -1.9274],
       [-2.0279, 1.625 , 0.5511, 3.0593],
       [ 0.4553, -0.0307, 0.9357, 1.0612],
       [-2.1079, 0.1999, 0.3236, -0.6416]], dtype=float64)
```
If a DataFrame or Panel 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.

### 9.3 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:

New in version 0.20.0.

```python
pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)
```

### 9.4 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.
9.4.1 Matching / broadcasting behavior

DataFrame has the methods \texttt{add()}, \texttt{sub()}, \texttt{mul()}, \texttt{div()} and related functions \texttt{radd()}, \texttt{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 \texttt{index} or \texttt{columns} via the \texttt{axis} keyword:

```python
In [14]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c', 'd']),
                        'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
                        'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd']))

In [15]: df
Out[15]:
        one   three   two
   a -1.101558  NaN  1.124472
   b -0.177289 -0.634293  2.487104
   c  0.462215  1.931194 -0.486066
d  NaN -1.222918 -0.456288

In [16]: row = df.iloc[1]
In [17]: column = df['two']

In [18]: df.sub(row, axis='columns')
Out[18]:
        one   three   two
   a -0.924269  NaN -1.362632
   b  0.000000  0.000000  0.000000
   c  0.639504  2.565487 -2.973170
d  NaN -0.588625 -2.943392

In [19]: df.sub(column, axis='index')

In [20]: df.sub(column, axis=0)

In [21]: df.sub(column, axis=0)
```

---

Chapter 9. Essential Basic Functionality
Furthermore you can align a level of a multi-indexed DataFrame with a Series.

```python
In [22]: dfmi = df.copy()
In [23]: dfmi.index = pd.MultiIndex.from_tuples([(1,'a'),(1,'b'),(1,'c'),(2,'a')], names=['first','second'])
In [24]: dfmi.sub(column, axis=0, level='second')
Out[24]:
   one  three  two
first second
1   a  -2.226031  NaN  0.0000
   b  -2.664393 -3.121397  0.0000
   c   0.948280  2.417260  0.0000
2   a    NaN  -2.347391 -1.58076
```

With Panel, describing the matching behavior is a bit more difficult, so the arithmetic methods instead (and perhaps confusingly?) give you the option to specify the broadcast axis. For example, suppose we wished to demean the data over a particular axis. This can be accomplished by taking the mean over an axis and broadcasting over the same axis:

```python
In [25]: major_mean = wp.mean(axis='major')
In [26]: major_mean
Out[26]:
   Item1  Item2
A -0.878036 -0.092218
B -0.060128  0.529811
C  0.099453 -0.715139
D  0.248599 -0.186535
In [27]: wp.sub(major_mean, axis='major')
```

And similarly for axis="items" and axis="minor".

**Note:** I could be convinced to make the axis argument in the DataFrame methods match the broadcasting behavior of Panel. Though it would require a transition period so users can change their code...

Series and Index also support the `divmod()` built-in. 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:

```python
In [28]: s = pd.Series(np.arange(10))
In [29]: s
Out[29]:
0     0
```

## 9.4. Flexible binary operations
In [30]: div, rem = divmod(s, 3)

In [31]: div
Out[31]:
0 0
1 0
2 0
3 1
4 1
5 1
6 2
7 2
8 2
9 3
dtype: int64

In [32]: rem

In [33]: idx = pd.Index(np.arange(10))

In [34]: idx
Out[34]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [35]: div, rem = divmod(idx, 3)

In [36]: div
Out[36]: Int64Index([0, 0, 1, 1, 2, 2, 2, 3], dtype='int64')

In [37]: rem

We can also do elementwise divmod():

Chapter 9. Essential Basic Functionality
9.4.2 Missing data / operations with fill values

In Series and DataFrame (though not yet in Panel), 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).
9.4.3 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 [45]: df.gt(df2)
Out[45]:
  one three two
a  False False False
b  False False False
c  False False False
d  False False False
```

```
In [46]: df2.ne(df)
Out[46]:
  one three two
a  False True False
b  False False False
c  False False False
d  True False False
```

These operations produce a pandas object the same type as the left-hand-side input that if of dtype `bool`. These boolean objects can be used in indexing operations, see [here](#).

9.4.4 Boolean Reductions

You can apply the reductions: `empty, any(), all()`, and `bool()` to provide a way to summarize a boolean result.

```
In [47]: (df > 0).all()
Out[47]:
  one    False
three   False
two     False
dtype:   bool
```

```
In [48]: (df > 0).any()
Out[48]:
  one
three
two
```
You can reduce to a final boolean value.

```python
In [49]: (df > 0).any().any()
Out[49]: True
```

You can test if a pandas object is empty, via the `empty` property.

```python
In [50]: df.empty
Out[50]: False
```

```python
In [51]: pd.DataFrame(columns=list('ABC')).empty
Out[51]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```python
In [52]: pd.Series([True]).bool()
Out[52]: True
```

```python
In [53]: pd.Series([False]).bool()
Out[53]: False
```

```python
In [54]: pd.DataFrame([[True]]).bool()
Out[54]: True
```

```python
In [55]: pd.DataFrame([[False]]).bool()
Out[55]: False
```

**Warning:** You might be tempted to do the following:

```python
>>> if df:
...     ...
```

Or

```python
>>> df and df2
```

These both will raise 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.

### 9.4.5 Comparing if objects are equivalent

Often you may find there is more than one way to compute the same result. As a simple example, consider `df+df` and `df*2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df+df == df*2).all()`. But in fact, this expression is False:
In [56]: df+df == df*2
Out[56]:
    one  three  two
   a    True   False   True
   b    True    True   True
   c    True    True   True
   d   False    True   True

In [57]: (df+df == df*2).all()
   →
    one   False
    three  False
    two    True
dtype: bool

Notice that the boolean DataFrame df+df == df*2 contains some False values! That is because NaNs do not compare as equals:

In [58]: np.nan == np.nan
Out[58]: False

So, NDFrames (such as Series, DataFrames, and Panels) have an equals() method for testing equality, with NaNs in corresponding locations treated as equal.

In [59]: (df+df).equals(df*2)
Out[59]: True

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

In [60]: df1 = pd.DataFrame({'col':['foo', 0, np.nan]})
In [61]: df2 = pd.DataFrame({'col':[np.nan, 0, 'foo']}, index=[2,1,0])
In [62]: df1.equals(df2)
Out[62]: False
In [63]: df1.equals(df2.sort_index())
Out[63]: True

9.4.6 Comparing array-like objects

You can conveniently do element-wise comparisons when comparing a pandas data structure with a scalar value:

In [64]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
Out[64]:
0   True
1  False
2  False
dtype: bool

In [65]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
Out[65]: array([True, False, False], dtype=bool)

Pandas also handles element-wise comparisons between different array-like objects of the same length:
trying to compare index or series objects of different lengths will raise a valueerror:

```python
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare
```

```python
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:

```python
In [68]: np.array([1, 2, 3]) == np.array([2])
Out[68]: array([False, True, False], dtype=bool)
```

or it can return False if broadcasting cannot be done:

```python
In [69]: np.array([1, 2, 3]) == np.array([1, 2])
Out[69]: False
```

### 9.4.7 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:

```python
In [70]: df1 = pd.DataFrame({'A': [1., np.nan, 3., 5., np.nan],
                     'B': [np.nan, 2., 3., np.nan, 6.]})

In [71]: df2 = pd.DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
                     'B': [np.nan, np.nan, 3., 4., 6., 8.]})

In [72]: df1
Out[72]:
    A     B
   --- ---
0   1.0  NaN
1  NaN  2.0
2   3.0   3.0
3   5.0  NaN
```
9.4.8 General DataFrame Combine

The `combine_first()` method above calls the more general DataFrame method `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 [75]: combiner = lambda x, y: np.where(pd.isna(x), y, x)
In [76]: df1.combine(df2, combiner)
Out[76]:
   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
```

9.5 Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on `Series`, `DataFrame`, and `Panel`. 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)
- **Panel**: “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

```
In [77]: df
Out[77]:
    one    three    two
a -1.101558  NaN   1.124472
b -0.177289 -0.634293  2.487104
c  0.462215  1.931194 -0.486066
d   NaN      -1.222918 -0.456288
```

```
In [78]: df.mean(0)
Out[78]:
    one    three
a -0.272211  0.024661
b  0.558507 -0.486066
c  0.667306 -0.456288
d  NaN       NaN
```

```
In [79]: df.mean(1)
Out[79]:
    a     b     c     d
one -0.011457  0.558507  3.635781 -0.839603
two  0.022914  1.675522  1.907343 -1.679206
d   NaN       NaN        NaN     NaN
```

All such methods have a `skipna` option signaling whether to exclude missing data (True by default):

```
In [80]: df.sum(0, skipna=False)
Out[80]:
    one    three
a   NaN  NaN
b   NaN  NaN
two 2.669223
```

```
In [81]: df.sum(axis=1, skipna=True)
Out[81]:
    a     b     c     d
one -0.022914  1.675522  1.907343 -1.679206
two  0.022914  1.675522  1.907343 -1.679206
d   NaN       NaN       NaN     NaN
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [82]: ts_stand = (df - df.mean()) / df.std()
```

```
In [83]: ts_stand.std()
Out[83]:
    one    three
a  1.0   1.0
b  1.0   1.0
two 1.0   1.0
```

9.5. Descriptive statistics
```python
dtype: float64

In [84]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [85]: xs_stand.std(1)
Out[85]:
   a   b   c   d
0  1.0  1.0  1.0  1.0
```

Note that methods like `cumsum()` and `cumprod()` preserve the location of NaN values. This is somewhat different from `expanding()` and `rolling()`. For more details please see this note.

```python
In [86]: df.cumsum()
Out[86]:
   one  three  two
  a    -1.101558  NaN    1.124472
  b    -1.278848 -0.634293  3.611576
  c    -0.816633  1.296901  3.125511
  d     NaN    0.073983  2.669223
```

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 [87]: np.mean(df['one'])
Out[87]: -0.27221094480450114

In [88]: np.mean(df['one'].values)
```

\```
Series also has a method `nunique()` which will return the number of unique non-NA values:

```python
In [89]: series = pd.Series(np.random.randn(500))
In [90]: series[20:500] = np.nan
In [91]: series[10:20] = 5
In [92]: series.nunique()
```
```
Out[92]: 11
```

### 9.5.1 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 [93]: series = pd.Series(np.random.randn(1000))
In [94]: series[::2] = np.nan
In [95]: series.describe()
```
```
Out[95]:

<table>
<thead>
<tr>
<th></th>
<th>500.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>500.000000</td>
</tr>
<tr>
<td>mean</td>
<td>-0.032127</td>
</tr>
<tr>
<td>std</td>
<td>1.067484</td>
</tr>
<tr>
<td>min</td>
<td>-3.463789</td>
</tr>
<tr>
<td>25%</td>
<td>-0.725523</td>
</tr>
<tr>
<td>50%</td>
<td>-0.053230</td>
</tr>
<tr>
<td>75%</td>
<td>0.679790</td>
</tr>
<tr>
<td>max</td>
<td>3.120271</td>
</tr>
</tbody>
</table>

```

```python
In [96]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [97]: frame.iloc[::2] = np.nan
In [98]: frame.describe()
```
```
Out[98]:

<table>
<thead>
<tr>
<th></th>
<th>500.000000</th>
<th>500.000000</th>
<th>500.000000</th>
<th>500.000000</th>
<th>500.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
</tr>
<tr>
<td>mean</td>
<td>-0.045109</td>
<td>-0.052045</td>
<td>0.024520</td>
<td>0.006117</td>
<td>0.001141</td>
</tr>
<tr>
<td>std</td>
<td>1.029268</td>
<td>1.002320</td>
<td>1.042793</td>
<td>1.040134</td>
<td>1.005207</td>
</tr>
<tr>
<td>min</td>
<td>-2.915767</td>
<td>-3.294023</td>
<td>-3.610499</td>
<td>-2.907036</td>
<td>-3.010899</td>
</tr>
<tr>
<td>25%</td>
<td>-0.763783</td>
<td>-0.720389</td>
<td>-0.609600</td>
<td>-0.665896</td>
<td>-0.682900</td>
</tr>
<tr>
<td>50%</td>
<td>-0.086033</td>
<td>-0.048843</td>
<td>0.006093</td>
<td>0.043191</td>
<td>-0.001651</td>
</tr>
<tr>
<td>75%</td>
<td>0.663399</td>
<td>0.620980</td>
<td>0.728382</td>
<td>0.735973</td>
<td>0.656439</td>
</tr>
<tr>
<td>max</td>
<td>3.400646</td>
<td>2.925597</td>
<td>3.416896</td>
<td>3.331522</td>
<td>3.007143</td>
</tr>
</tbody>
</table>
```

You can select specific percentiles to include in the output:

```python
In [99]: series.describe(percentiles=[.05, .25, .75, .95])
```
```
Out[99]:

<table>
<thead>
<tr>
<th></th>
<th>500.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>500.000000</td>
</tr>
<tr>
<td>mean</td>
<td>-0.032127</td>
</tr>
<tr>
<td>std</td>
<td>1.067484</td>
</tr>
<tr>
<td>min</td>
<td>-3.463789</td>
</tr>
</tbody>
</table>
```

### 9.5. Descriptive statistics
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:

```python
In [100]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])
In [101]: s.describe()
```
```
Out[101]:
    count 9
    unique 4
    top  a
    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:

```python
In [102]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})
In [103]: frame.describe()
```
```
Out[103]:
    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 behaviour can be controlled by providing a list of types as `include/exclude` arguments. The special value `all` can also be used:

```python
In [104]: frame.describe(include=['object'])
Out[104]:
    a
   count 4
   unique 2
   top  Yes
   freq  2

In [105]: frame.describe(include=['number'])
```
```
Out[105]:
    b
   count 4.000000
   mean 1.500000
   std  1.290994
   min  0.000000
```
That feature relies on `select_dtypes`. Refer to there for details about accepted inputs.

### 9.5.2 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 [107]: s1 = pd.Series(np.random.randn(5))

In [108]: s1
Out[108]:
0  -1.649461
1   0.169660
2   1.246181
3   0.131682
4  -2.001988
dtype: float64

In [109]: s1.idxmin(), s1.idxmax()
Out[109]: (4, 2)

In [110]: df1 = pd.DataFrame(np.random.randn(5,3), columns=['A','B','C'])

In [111]: df1
Out[111]:
   A         B         C
0  0.248389  0.158838 -0.340113
1  0.608160  1.102219 -0.177877
2 -0.083854  1.352369 -0.285804
3  0.048924 -0.454786  0.649357
4 -0.173340 -0.838154 -0.466048

In [112]: df1.idxmin(axis=0)
```
When there are multiple rows (or columns) matching the minimum or maximum value, \texttt{idxmin()} and \texttt{idxmax()} return the first matching index:

```python
In [114]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))
In [115]: df3
Out[115]:
   A
a NaN
b  3.0
c  1.0
d  1.0
e  2.0

In [116]: df3['A'].idxmin()
Out[116]: 'd'
```

**Note:** \texttt{idxmin} and \texttt{idxmax} are called \texttt{argmin} and \texttt{argmax} in NumPy.

### 9.5.3 Value counts (histogramming) / Mode

The \texttt{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 [117]: data = np.random.randint(0, 7, size=50)

In [118]: data
Out[118]:
array([3, 3, 0, 2, 1, 0, 5, 3, 6, 1, 5, 6, 2, 0, 0, 6, 3, 3, 5, 0, 4, 3,
      3, 3, 0, 6, 1, 3, 5, 5, 0, 4, 0, 6, 3, 6, 5, 4, 3, 2, 1, 5, 0, 1, 1,
      6, 4, 1, 4])

In [119]: s = pd.Series(data)

In [120]: s.value_counts()
Out[120]:
3    11
0     9
5     8
```
Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```python
In [122]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])

In [123]: s5.mode()
Out[123]:
0 3
1 7
dtype: int64

In [124]: df5 = pd.DataFrame({'A': np.random.randint(0, 7, size=50),
                        'B': np.random.randint(-10, 15, size=50)})

In [125]: df5.mode()
Out[125]:
   A  B
0 2 -5
```

### 9.5.4 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 [126]: arr = np.random.randn(20)

In [127]: factor = pd.cut(arr, 4)

In [128]: factor
Out[128]:
[(-2.611, -1.58], (0.473, 1.499], (-2.611, -1.58], (-1.58, -0.554], (-0.554, 0.473], ...
          ... , (0.473, 1.499], (0.473, 1.499], (-0.554, 0.473], (-0.554, 0.473]]
Length: 20
Categories (4, interval[float64]): [(-2.611, -1.58] < (-1.58, -0.554] < (-0.554, 0.
          ... , (0.473, 1.499])
```

9.5. Descriptive statistics
qcut() computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [131]: arr = np.random.randn(30)
In [132]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])
In [133]: factor
Out[133]: [(0.544, 1.976], (0.544, 1.976], (-1.255, -0.375], (0.544, 1.976], (-0.103, 0.544], ..
            ..
            ..
            ..
            , (-0.103, 0.544], (0.544, 1.976], (-0.103, 0.544], (-1.255, -0.375], (-0.375, -0.103])
Length: 30
Categories (4, interval[float64]): [(-1.255, -0.375) < (-0.375, -0.103) < (-0.103, 0.544) <
                                                  (0.544, 1.976)]
```

We can also pass infinite values to define the bins:

```
In [135]: arr = np.random.randn(20)
In [136]: factor = pd.cut(arr, [-np.inf, 0, np.inf])
In [137]: factor
Out[137]: [(0.0, inf], (0.0, inf], (0.0, inf], (0.0, inf], (0.0, inf], (0.0, inf], ...
           , (-inf, 0.0], (0.0, inf], (-inf, 0.0], (0.0, inf], (0.0, inf]
Length: 20
Categories (2, interval[float64]): [(-inf, 0.0) < (0.0, inf]]
```

### 9.6 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()`

### 9.6.1 Tablewise Function Application

DataFrames and Series can of course just be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method. Compare the following

```python
# f, g, and h are functions taking and returning ''DataFrames''
>>> f(g(h(df), arg1=1), arg2=2, arg3=3)
```

with the equivalent

```python
>>> (df.pipe(h)
     .pipe(g, arg1=1)
     .pipe(f, arg2=2, arg3=3)
)
```

Pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another library’s functions in method chains, alongside pandas’ methods.

In the example above, the functions `f`, `g`, and `h` each expected the 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.poisson, 'data')` to `pipe`:

```python
In [138]: import statsmodels.formula.api as sm
In [139]: bb = pd.read_csv('data/baseball.csv', index_col='id')
In [140]: (bb.query('h > 0')
     ....: .assign(ln_h = lambda df: np.log(df.h))
     ....: .pipe(sm.poisson, 'data'), 'hr ~ ln_h + year + g + C(lg)')
     ....: .fit()
     ....: .summary()
     ....: )
Optimization terminated successfully.
Current function value: 2.116284
Iterations 24
```

```python
Out[140]:
<class 'statsmodels.iolib.summary.Summary'>
```

```python
Poisson Regression Results
==============================================================================
Dep. Variable:                     hr   No. Observations:               68
Model:                   Poisson   Df Residuals:                  63
Method:                     MLE   Df Model:                     4
Date:            Fri, 27 Oct 2017   Pseudo R-squ.:           0.6878
Time:                        10:34:06   Log-Likelihood:       -143.91
converged:                    True   LL-Null:            -460.91
                           LLR p-value:    6.774e-136
==============================================================================
```

9.6. **Function application**
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 (pd.DataFrame.pipe?? in IPython).

### 9.6.2 Row or Column-wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the `apply()` method, which, like the descriptive statistics methods, take an optional `axis` argument:
.apply() will also dispatch on a string method name.

```python
In [146]: df.apply('mean')
Out[146]:
one -0.272211
three 0.024661
two  0.667306
dtype: float64
```

```python
In [147]: df.apply('mean', axis=1)
Out[147]:
a  0.011457
b  0.558507
c  0.635781
d -0.839603
dtype: float64
```

Depending on the return type of the function passed to `apply()`, the result will either be of lower dimension or the same dimension.

`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:

```python
In [148]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'], index=pd.date_range('1/1/2000', periods=1000))
In [149]: tsdf.apply(lambda x: x.idxmax())
Out[149]:
A  2001-04-25
B  2002-05-31
C  2002-09-25
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:

```python
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:

```python
In [150]: tsdf
Out[150]:
          A         B         C
2000-01-01 -0.720299  0.546303 -0.082042
2000-01-02  0.200295 -0.577554 -0.908402
2000-01-03  0.102533  1.653614  0.303319
```

9.6. Function application
Finally, \texttt{apply()} takes an argument \texttt{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.

### 9.6.3 Aggregation API

New in version 0.20.0.

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see \textit{groupby API}, the \textit{window functions API}, and the \textit{resample API}. The entry point for aggregation is the method \texttt{aggregate()}, or the alias \texttt{agg()}.

We will use a similar starting frame from above:

```
In [152]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
....:                     index=pd.date_range('1/1/2000', periods=10))
....:
```

```
In [153]: tsdf.iloc[3:7] = np.nan
```

```
In [154]: tsdf
Out[154]:
```

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.170247</td>
<td>-0.916844</td>
<td>0.835024</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.259919</td>
<td>0.801111</td>
<td>0.445614</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>1.453046</td>
<td>2.430373</td>
<td>0.653093</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>-1.874526</td>
<td>0.569822</td>
<td>-0.609644</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>0.812462</td>
<td>0.565894</td>
<td>-1.461363</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-0.985475</td>
<td>1.388154</td>
<td>-0.078747</td>
</tr>
</tbody>
</table>
```
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 [155]: tsdf.agg(np.sum)
Out[155]:
    A   B   C
dtype: float64
   0.835673  4.838510 -0.216025

In [156]: tsdf.agg('sum')
```

# these are equivalent to a `sum()` because we are aggregating on a single function

```python
In [157]: tsdf.sum()
```

Single aggregations on a Series this will result in a scalar value:

```python
In [158]: tsdf.A.agg('sum')
Out[158]:
    0.83567297915820504
```

### 9.6.3.1 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 resultant DataFrame. These are naturally named from the aggregation function.

```python
In [159]: tsdf.agg(['sum'])
Out[159]:
   A   B   C
sum  0.835673  4.838510 -0.216025
```

Multiple functions yield multiple rows:

```python
In [160]: tsdf.agg(['sum', 'mean'])
Out[160]:
   A   B   C
sum  0.835673  4.838510 -0.216025
  mean  0.139279  0.806418 -0.036004
```

On a Series, multiple functions return a Series, indexed by the function names:

```python
In [161]: tsdf.A.agg(['sum', 'mean'])
Out[161]:
   sum  0.835673
  mean  0.139279
Name: A, dtype: float64
```
Passing a lambda function will yield a <lambda> named row:

```
In [162]: tsdf.A.agg(['sum', lambda x: x.mean()])
Out[162]:
   sum     <lambda>
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [163]: def mymean(x):
   .....:     return x.mean()
   .....:

In [164]: tsdf.A.agg(['sum', mymean])
Out[164]:
   sum  0.835673
   mymean  0.139279
Name: A, dtype: float64
```

### 9.6.3.2 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.

```
In [165]: tsdf.agg({'A': 'mean', 'B': 'sum'})
Out[165]:
   A  0.139279
   B  4.838510
dtype: float64
```

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:

```
In [166]: tsdf.agg({'A': ['mean', 'min'], 'B': 'sum'})
Out[166]:
       A  B
mean 0.139279  NaN
min -1.874526  NaN
sum  NaN  4.83851
```

### 9.6.3.3 Mixed Dtypes

When presented with mixed dtypes that cannot aggregate, .agg will only take the valid aggregations. This is similar to how groupby .agg works.

```
In [167]: mdf = pd.DataFrame({'A': [1, 2, 3],
   .....:                        'B': [1., 2., 3.],
   .....:                        'C': ['foo', 'bar', 'baz'],
   .....:                        'D': pd.date_range('20130101', periods=3))
   .....:

In [168]: mdf.dtypes
Out[168]:
```
9.6.3.4 Custom describe

With `.agg()` is it possible to easily create a custom describe function, similar to the built in `describe` function.

```python
In [170]: from functools import partial
In [171]: q_25 = partial(pd.Series.quantile, q=0.25)
In [172]: q_25.__name__ = '25%'
In [173]: q_75 = partial(pd.Series.quantile, q=0.75)
In [174]: q_75.__name__ = '75%'
In [175]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
```

9.6.4 Transform API

New in version 0.20.0.

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.

Use a similar frame to the above sections.

```python
In [176]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                       index=pd.date_range('1/1/2000', periods=10))
In [177]: tsdf.iloc[3:7] = np.nan
In [178]: tsdf
```
Transform the entire frame. `.transform()` allows input functions as: a numpy function, a string function name or a user defined function.

In [179]: tsdf.transform(np.abs)

Out[179]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.578465</td>
<td>0.503335</td>
<td>0.987140</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.767147</td>
<td>0.266046</td>
<td>1.083797</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.195348</td>
<td>0.722247</td>
<td>0.894537</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>0.556397</td>
<td>0.542165</td>
<td>0.308675</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>1.010924</td>
<td>0.672504</td>
<td>1.139222</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>0.354653</td>
<td>0.563622</td>
<td>0.365106</td>
</tr>
</tbody>
</table>

In [180]: tsdf.transform('abs')

Out[180]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.578465</td>
<td>0.503335</td>
<td>0.987140</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.767147</td>
<td>0.266046</td>
<td>1.083797</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.195348</td>
<td>0.722247</td>
<td>0.894537</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>0.556397</td>
<td>0.542165</td>
<td>0.308675</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>1.010924</td>
<td>0.672504</td>
<td>1.139222</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>0.354653</td>
<td>0.563622</td>
<td>0.365106</td>
</tr>
</tbody>
</table>

In [181]: tsdf.transform(lambda x: x.abs())

Out[181]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.578465</td>
<td>0.503335</td>
<td>0.987140</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.767147</td>
<td>0.266046</td>
<td>1.083797</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.195348</td>
<td>0.722247</td>
<td>0.894537</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>0.556397</td>
<td>0.542165</td>
<td>0.308675</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>1.010924</td>
<td>0.672504</td>
<td>1.139222</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>0.354653</td>
<td>0.563622</td>
<td>0.365106</td>
</tr>
</tbody>
</table>
Here .transform() received a single function; this is equivalent to a ufunc application

```python
In [182]: np.abs(tsdf)
Out[182]:
     A        B        C
2000-01-01  0.578465  0.503335  0.987140
2000-01-02  0.767147  0.266046  1.083797
2000-01-03  0.195348  0.722247  0.894537
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  0.556397  0.542165  0.308675
2000-01-09  1.010924  0.672504  1.139222
2000-01-10  0.354653  0.563622  0.365106
```

Passing a single function to .transform() with a Series will yield a single Series in return.

```python
In [183]: tsdf.A.transform(np.abs)
Out[183]:
     2000-01-01  0.578465
     2000-01-02  0.767147
     2000-01-03  0.195348
     2000-01-04   NaN
     2000-01-05   NaN
     2000-01-06   NaN
     2000-01-07   NaN
     2000-01-08  0.556397
     2000-01-09  1.010924
     2000-01-10  0.354653
Freq: D, Name: A, dtype: float64
```

### 9.6.4.1 Transform with multiple functions

Passing multiple functions will yield a column multi-indexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```python
In [184]: tsdf.transform([np.abs, lambda x: x+1])
Out[184]:
    A       <lambda>  B       <lambda>  C       <lambda>
2000-01-01  0.578465  0.421535  0.503335  0.496665  0.987140  0.012860
2000-01-02  0.767147  0.232853  0.266046  0.733954  1.083797  2.083797
2000-01-03  0.195348  1.195348  0.722247  1.722247  0.894537  0.105463
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  0.556397  0.443603  0.542165  1.542165  0.308675  0.691325
2000-01-09  1.010924 -0.010924  0.672504  0.327496  1.139222 -0.139222
2000-01-10  0.354653  1.354653  0.563622  1.563622  0.365106  0.634894
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```python
In [185]: tsdf.A.transform([np.abs, lambda x: x+1])
 Out[185]:
```

### 9.6. Function application
9.6.4.2 Transforming with a dict

Passing a dict of functions will will allow selective transforming per column.

```
In [186]: tsdf.transform({'A': np.abs, 'B': lambda x: x+1})
Out[186]:
      A       B
2000-01-01  0.578465  0.496665
2000-01-02  0.767147  0.733954
2000-01-03  0.195348  1.722247
2000-01-04  NaN        NaN
2000-01-05  NaN        NaN
2000-01-06  NaN        NaN
2000-01-07  NaN        NaN
2000-01-08  0.556397  1.542165
2000-01-09  1.010924 -0.010924
2000-01-10  0.354653  1.354653
```

Passing a dict of lists will generate a multi-indexed DataFrame with these selective transforms.

```
In [187]: tsdf.transform({'A': np.abs, 'B': [lambda x: x+1, 'sqrt']})
Out[187]:
      absolute       <lambda>      sqrt
2000-01-01  0.578465   0.496665        NaN
2000-01-02  0.767147   0.733954        NaN
2000-01-03  0.195348  1.722247   0.849851
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  0.556397  1.542165        NaN
2000-01-09  1.010924 -0.010924        NaN
2000-01-10  0.354653  1.354653        NaN
```

9.6.5 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:
pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>one</th>
<th>three</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.101558</td>
<td>NaN</td>
</tr>
<tr>
<td>b</td>
<td>-0.177289</td>
<td>-0.634293</td>
</tr>
<tr>
<td>c</td>
<td>0.462215</td>
<td>1.931194</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
<td>-1.222918</td>
</tr>
</tbody>
</table>

In [189]: f = lambda x: len(str(x))

In [190]: df4['one'].map(f)
Out[190]:
a 19
b 20
c 18
d 3
Name: one, dtype: int64

In [191]: df4.applymap(f)

Series.map() has an additional feature which is that it can be used to easily “link” or “map” values defined by a secondary series. This is closely related to merging/joining functionality:

In [192]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'], index=['a', 'b', 'c', 'd', 'e'])

In [193]: t = pd.Series({'six': 6., 'seven': 7.})

In [194]: s
Out[194]:
a  six
b  seven
c  six
d  seven
e  six
dtype: object

In [195]: s.map(t)

9.6.6 Applying with a Panel

Applying with a Panel will pass a Series to the applied function. If the applied function returns a Series, the result of the application will be a Panel. If the applied function reduces to a scalar, the result of the application will
be a DataFrame.

```
In [196]: import pandas.util.testing as tm
In [197]: panel = tm.makePanel(5)
In [198]: panel
Out[198]:
<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 [199]: panel['ItemA']
            A     B     C     D
2000-01-03  1.092702  0.604244 -2.927808  0.339642
2000-01-04  -1.481449 -0.487265  0.082065  1.499953
2000-01-05   1.781190  1.990533  0.456554  -0.317818
2000-01-06  -0.031543  0.327007  -1.757911  0.447371
2000-01-07   0.480993  1.053639   0.982407 -1.315799
```

A transformational apply.

```
In [200]: result = panel.apply(lambda x: x*2, axis='items')
In [201]: result
Out[201]:
<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 [202]: result['ItemA']
            A     B     C     D
2000-01-03  2.185405  1.208489 -5.855616  0.679285
2000-01-04  -2.962899 -0.974530  0.164130  2.999905
2000-01-05   3.562379  3.981066   0.913107 -0.635635
2000-01-06  -0.063086  0.654013  -3.515821  0.894742
2000-01-07   0.961986  2.107278   1.964815 -2.631598
```

A reduction operation.

```
In [203]: panel.apply(lambda x: x.dtype, axis='items')
Out[203]:
          A     B     C     D
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
In [204]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[204]:
<table>
<thead>
<tr>
<th></th>
<th>ItemA</th>
<th>ItemB</th>
<th>ItemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.841893</td>
<td>0.918017</td>
<td>-1.160547</td>
</tr>
<tr>
<td>B</td>
<td>3.488158</td>
<td>-2.629773</td>
<td>0.603397</td>
</tr>
<tr>
<td>C</td>
<td>-3.164692</td>
<td>0.805970</td>
<td>0.806501</td>
</tr>
<tr>
<td>D</td>
<td>0.653349</td>
<td>-0.152299</td>
<td>0.252577</td>
</tr>
</tbody>
</table>

This last reduction is equivalent to

In [205]: panel.sum('major_axis')
Out[205]:
<table>
<thead>
<tr>
<th></th>
<th>ItemA</th>
<th>ItemB</th>
<th>ItemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.841893</td>
<td>0.918017</td>
<td>-1.160547</td>
</tr>
<tr>
<td>B</td>
<td>3.488158</td>
<td>-2.629773</td>
<td>0.603397</td>
</tr>
<tr>
<td>C</td>
<td>-3.164692</td>
<td>0.805970</td>
<td>0.806501</td>
</tr>
<tr>
<td>D</td>
<td>0.653349</td>
<td>-0.152299</td>
<td>0.252577</td>
</tr>
</tbody>
</table>

A transformation operation that returns a Panel, but is computing the z-score across the major_axis.

In [206]: result = panel.apply(lambda x: (x-x.mean())/x.std(), axis='major_axis')

In [207]: result
Out[207]:
<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 [208]: result['ItemA']

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>0.585813</td>
<td>-0.102070</td>
<td>-1.394063</td>
<td>0.201263</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.496089</td>
<td>-1.295066</td>
<td>0.434343</td>
<td>1.318766</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>1.142642</td>
<td>1.413112</td>
<td>0.661833</td>
<td>-0.431942</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.323445</td>
<td>-0.405085</td>
<td>-0.683386</td>
<td>0.305017</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.091079</td>
<td>0.389108</td>
<td>0.981273</td>
<td>-1.393105</td>
</tr>
</tbody>
</table>

Apply can also accept multiple axes in the axis argument. This will pass a DataFrame of the cross-section to the applied function.

In [209]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T

In [210]: result = panel.apply(f, axis=['items','major_axis'])

In [211]: result
Out[211]:
<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

9.6. Function application  545
This is equivalent to the following

```python
In [213]: result = pd.Panel(dict((ax, f(panel.loc[:,:,ax])) for ax in panel.minor_axis ))
```
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:

```python
In [219]: df
Out[219]:
   one  three  two
a -1.101558  NaN  1.124472
b -0.177289  0.634293  2.487104
c  0.462215  1.931194  0.486066
d  NaN  1.222918  0.456288

In [220]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[220]:
     three  two  one
   c  1.931194  0.486066  0.462215
   f  NaN      NaN      NaN
   b -0.634293  2.487104  0.177289
```

You may also use reindex with an axis keyword:

```python
In [221]: df.reindex(['c', 'f', 'b'], axis='index')
Out[221]:
   one  three  two
a -1.101558  NaN  1.124472
b -0.177289 -0.634293  2.487104
c  0.462215  1.931194  0.486066
d  NaN      NaN      NaN
```

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:

```python
In [222]: rs = s.reindex(df.index)

In [223]: rs
Out[223]:
   a  -0.454087
   b  -0.360309
   c  -0.951631
   d  -0.535459
dtype: float64

In [224]: rs.index is df.index
Out[224]: True
```

This means that the reindexed Series’s index is the same Python object as the DataFrame’s index.
New in version 0.21.0.

`DataFrame.reindex()` also supports an “axis-style” calling convention, where you specify a single `labels` argument and the `axis` it applies to.

```python
In [225]: df.reindex(['c', 'f', 'b'], axis='index')
Out[225]:
    one three two
  c  0.462215 1.931194 -0.486066
  f  NaN      NaN        NaN
  b -0.177289 -0.634293  2.487104

In [226]: df.reindex(['three', 'two', 'one'], axis='columns')
    three    two    one
  a  NaN   1.124472 -1.101558
  b -0.634293  2.487104  0.177289
  c  1.931194 -0.486066  0.462215
  d -1.222918 -0.456288   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.

### 9.7.1 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:

```python
In [227]: df2
Out[227]:
    one    two
  a -1.101558  1.124472
  b -0.177289  2.487104
  c  0.462215 -0.486066

In [228]: df3
    one    two
  a -0.829347  0.082635
  b  0.094922  1.445267
  c  0.734426 -1.527903

In [229]: df.reindex_like(df2)
    one    two
  a -1.101558  1.124472
```
### 9.7.2 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 [230]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [231]: s1 = s[:4]
In [232]: s2 = s[1:]
In [233]: s1.align(s2)
Out [233]:
a 0.505453  
b 1.788110  
c -0.405908 
d -0.801912 
       a NaN   
       b 1.788110  
c -0.405908  
d -0.801912  
e 0.768460   
dtype: float64,
```
```
In [234]: s1.align(s2, join='inner')
Out [234]:
   b 1.788110  
c -0.405908  
d -0.801912  
e 0.768460
   b 1.788110  
c -0.405908  
d -0.801912  
e 0.768460
   dtype: float64)
```
```
In [235]: s1.align(s2, join='left')
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
For DataFrames, the join method will be applied to both the index and the columns by default:

```python
In [236]: df.align(df2, join='inner')
Out[236]:
  one   two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066,
   one   two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066)
```

You can also pass an `axis` option to only align on the specified axis:

```python
In [237]: df.align(df2, join='inner', axis=0)
Out[237]:
  one   three   two
a -1.101558   NaN  1.124472
b -0.177289 -0.634293  2.487104
c  0.462215  1.931194 -0.486066,
  one   two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066)
```

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:

```python
In [238]: df.align(df2.iloc[0], axis=1)
Out[238]:
  one   three   two
a -1.101558   NaN  1.124472
b -0.177289 -0.634293  2.487104
c  0.462215  1.931194 -0.486066
d   NaN -1.222918 -0.456288,
  one  -1.101558
three  NaN   NaN
two   1.124472
Name: a, dtype: float64)
```

### 9.7.3 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:

```python
In [239]: rng = pd.date_range('1/3/2000', periods=8)
In [240]: ts = pd.Series(np.random.randn(8), index=rng)
```
In [241]: ts2 = ts[[0, 3, 6]]

In [242]: ts
Out[242]:
2000-01-03  0.466284
2000-01-04  -0.457411
2000-01-05  -0.364060
2000-01-06   0.785367
2000-01-07  -1.463093
2000-01-08   1.187315
2000-01-09  -0.493153
2000-01-10  -1.323445
Freq: D, dtype: float64

In [243]: ts2
Out[243]:
2000-01-03  0.466284
2000-01-06   0.785367
2000-01-09  -0.493153
dtype: float64

In [244]: ts2.reindex(ts.index)
Out[244]:
2000-01-03  0.466284
2000-01-04   NaN
2000-01-05   NaN
2000-01-06   0.785367
2000-01-07   NaN
2000-01-08   NaN
2000-01-09  -0.493153
2000-01-10   NaN
Freq: D, dtype: float64

In [245]: ts2.reindex(ts.index, method='ffill')
Out[245]:
2000-01-03  0.466284
2000-01-04  0.466284
2000-01-05  0.466284
2000-01-06  0.785367
2000-01-07  0.785367
2000-01-08  0.785367
2000-01-09  -0.493153
2000-01-10  -0.493153
Freq: D, dtype: float64

In [246]: ts2.reindex(ts.index, method='bfill')
Out[246]:
2000-01-03  0.466284
2000-01-04  0.785367
2000-01-05  0.785367
2000-01-06  0.785367
2000-01-07  0.785367
2000-01-08  -0.493153
2000-01-09  -0.493153
2000-01-10  -0.493153

9.7. Reindexing and altering labels
2000-01-09  -0.493153
2000-01-10   NaN
Freq: D, dtype: float64

In [247]: ts2.reindex(ts.index, method='nearest')

...  
Out[247]:

2000-01-03  0.466284
2000-01-04  0.466284
2000-01-05  0.785367
2000-01-06  0.785367
2000-01-07  0.785367
2000-01-08  -0.493153
2000-01-09  -0.493153
2000-01-10  -0.493153
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 [248]: ts2.reindex(ts.index).fillna(method='ffill')

Out[248]:

2000-01-03  0.466284
2000-01-04  0.466284
2000-01-05  0.466284
2000-01-06  0.785367
2000-01-07  0.785367
2000-01-08  0.785367
2000-01-09  -0.493153
2000-01-10  -0.493153
Freq: D, dtype: float64

reindex() will raise a ValueError if the index is not monotonic increasing or decreasing. fillna() and interpolate() will not make any checks on the order of the index.

9.7.4 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 [249]: ts2.reindex(ts.index, method='ffill', limit=1)

Out[249]:

2000-01-03  0.466284
2000-01-04  0.466284
2000-01-05   NaN
2000-01-06  0.785367
2000-01-07  0.785367
2000-01-08   NaN
2000-01-09  -0.493153
2000-01-10  -0.493153
Freq: D, dtype: float64

In contrast, tolerance specifies the maximum distance between the index and indexer values:

In [250]: ts2.reindex(ts.index, method='ffill', tolerance='1 day')

Out[250]:

...
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.

### 9.7.5 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 [251]: df
Out[251]:
    one     three     two
   a -1.101558  NaN  1.124472
   b -0.177289 -0.634293  2.487104
   c  0.462215  1.931194 -0.486066
   d  NaN -1.222918 -0.456288

In [252]: df.drop(['a', 'd'], axis=0)
\--
   one     three     two
   b -0.177289 -0.634293  2.487104
   c  0.462215  1.931194 -0.486066

In [253]: df.drop(['one'], axis=1)
\--
   three     two
   a  NaN  1.124472
   b -0.634293  2.487104
   c  1.931194 -0.486066
   d -1.222918 -0.456288
```

Note that the following also works, but is a bit less obvious / clean:

```
In [254]: df.reindex(df.index.difference(['a', 'd']))
Out[254]:
   one     three     two
   b -0.177289 -0.634293  2.487104
   c  0.462215  1.931194 -0.486066
```

### 9.7.6 Renaming / mapping labels

The rename() method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [255]: s
Out[255]:
```

### 9.7. Reindexing and altering labels
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 [257]: df.rename(columns={'one': 'foo', 'two': 'bar'},
                   index={'a': 'apple', 'b': 'banana', 'd': 'durian'})
Out[257]:
          foo  three  bar
apple -1.101558   NaN  1.124472
banana -0.177289 -0.634293  2.487104
c       0.462215  1.931194 -0.486066
durian   NaN  -1.222918  -0.456288
```

If the mapping doesn’t include a column/index label, it isn’t renamed. Also extra labels in the mapping don’t throw an error.

New in version 0.21.0.

`DataFrame.rename()` also supports an “axis-style” calling convention, where you specify a single `mapper` and the `axis` to apply that mapping to.

```python
In [258]: df.rename({'one': 'foo', 'two': 'bar'}, axis='columns')
```

```
File "<ipython-input-258-d5b8afcaa5ce>", line 1
    df.rename({'one': 'foo', 'two': 'bar'}, axis='columns'))
                       ^
SyntaxError: invalid syntax
```

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.

New in version 0.18.0.

Finally, `rename()` also accepts a scalar or list-like for altering the `Series.name` attribute.
The Panel class has a related `rename_axis()` class which can rename any of its three axes.

## 9.8 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. Other data structures, like DataFrame and Panel, 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
- **Panel**: item labels

Thus, for example, iterating over a DataFrame gives you the column names:

```python
In [261]: df = pd.DataFrame({'col1' : np.random.randn(3), 'col2' : np.random.randn(3)})
   ....:    index=['a', 'b', 'c'])
   ....:
In [262]: for col in df:
   ....:    print(col)
   ....:
   col1
   col2
```

Pandas objects also have the dict-like `iteritems()` 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 by 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*. 

---

**9.8. Iteration** 555
If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop using e.g. 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:

```python
In [263]: df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})
In [264]: for index, row in df.iterrows():
       ....:     row['a'] = 10
       ....:
In [265]: df
Out[265]:
     a b
0  1 a
1  2 b
2  3 c
```

### 9.8.1 `iteritems`

Consistent with the dict-like interface, `iteritems()` iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs
- **Panel**: (item, DataFrame) pairs

For example:

```python
In [266]: for item, frame in wp.iteritems():
       ....:     print(item)
       ....:     print(frame)
       ....:
Item1
     A     B     C     D
2000-01-01 -0.433567 -0.273610 0.680433 -0.308450
2000-01-02 -0.276099 -1.821168 -1.993606 -1.927385
2000-01-03 -2.027924 1.624972 0.551135 3.059267
2000-01-04  0.455264  0.030740 0.935716  1.061192
2000-01-05  0.095031  0.307409 0.323586  0.641630
Item2
     A     B     C     D
2000-01-01  0.587514  0.053897 0.194889  0.381994
2000-01-02  0.318587  2.089075  0.328293  0.090255
2000-01-03  0.748199  1.318931  1.929766  0.792652
2000-01-04  0.461007  0.542749  0.305384  0.479195
2000-01-05  0.950310  0.307409  0.787140  0.773882
```
9.8.2 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 [267]: for row_index, row in df.iterrows():
    ......:       print('%s

%s' % (row_index, row))
    ......:
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 [268]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
In [269]: df_orig.dtypes
Out[269]:
int   int64
float float64
dtype: object
In [270]: row = next(df_orig.iterrows())[1]
In [271]: row
Out[271]:
int  1
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`:

```python
In [272]: row['int'].dtype
Out[272]: dtype('float64')
In [273]: df_orig['int'].dtype
Out[273]: 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 as `iterrows`.

For instance, a contrived way to transpose the DataFrame would be:

```python
In [274]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})
In [275]: print(df2)
```

9.8. Iteration
9.8.3 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,

```
In [279]: 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 but just 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.

9.9 .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.

```
# datetime
In [280]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [281]: s
```

```
Out[281]:
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 [282]: s.dt.hour

→
0  9
1  9
2  9
3  9
dtype: int64

In [283]: s.dt.second

→
0 12
1 12
2 12
3 12
dtype: int64

In [284]: s.dt.day

→
0 1
1 2
2 3
3 4
dtype: int64

This enables nice expressions like this:

In [285]: s[s.dt.day==2]
Out[285]:
1  2013-01-02 09:10:12

dtype: datetime64[ns]

You can easily produces tz aware transformations:

In [286]: stz = s.dt.tz_localize('US/Eastern')

In [287]: stz
Out[287]:
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]

In [288]: stz.dt.tz

→<DstTzInfo 'US/Eastern' LMT-> 1 day, 19:04:00 STD>

You can also chain these types of operations:

In [289]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[289]:
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
You can also format datetime values as strings with `Series.dt.strftime()` which supports the same format as the standard `strftime()`.

```python
# DatetimeIndex
In [290]: s = pd.Series(pd.date_range('20130101', periods=4))

In [291]: s
Out[291]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: datetime64[ns]

In [292]: s.dt.strftime('%Y/%m/%d')
   0  2013/01/01
   1  2013/01/02
   2  2013/01/03
   3  2013/01/04
dtype: object
```

```python
# PeriodIndex
In [293]: s = pd.Series(pd.period_range('20130101', periods=4))

In [294]: s
Out[294]:
0 2013-01-01
1 2013-01-02
2 2013-01-03
3 2013-01-04
dtype: object

In [295]: s.dt.strftime('%Y/%m/%d')
   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.

```python
# period
In [296]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [297]: s
Out[297]:
0 2013-01-01
1 2013-01-02
2 2013-01-03
3 2013-01-04
```
In [298]: s.dt.year
   Out[298]:
   0  2013
   1  2013
   2  2013
   3  2013
dtype: int64

In [299]: s.dt.day
   Out[299]:
   0  1
   1  2
   2  3
   3  4
dtype: int64

# timedelta
In [300]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [301]: s
Out[301]:
   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 [302]: s.dt.days
   Out[302]:
   0  1
   1  1
   2  1
   3  1
dtype: int64

In [303]: s.dt.seconds
   Out[303]:
   0  5
   1  6
   2  7
   3  8
dtype: int64

In [304]: s.dt.components
   Out[304]:
   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

9.9. dt accessor
9.10 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:

```python
In [305]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog →', 'cat'])

In [306]: s.str.lower()
Out[306]:
   0    a
   1    b
   2    c
   3   aaba
   4    baca
   5   NaN
   6    caba
   7     dog
   8    cat
```

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).

Please see Vectorized String Methods for a complete description.

9.11 Sorting

**Warning**: The sorting API is substantially changed in 0.17.0, see here for these changes. In particular, all sorting methods now return a new object by default, and **DO NOT** operate in-place (except by passing `inplace=True`).

There are two obvious kinds of sorting that you may be interested in: sorting by label and sorting by actual values.

9.11.1 By Index

The primary method for sorting axis labels (indexes) are the `Series.sort_index()` and the `DataFrame.sort_index()` methods.

```python
In [307]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
                           columns=['three', 'two', 'one'])

# DataFrame
In [308]: unsorted_df.sort_index()
Out[308]:
   three  two  one  
   ---  ---  ---
      a    b    c
```

Note: `Series.dt` will raise a `TypeError` if you access with a non-datetimelike values
# Series

9.11.2 By Values

The `Series.sort_values()` and `DataFrame.sort_values()` are the entry points for value sorting (that is the values in a column or row). `DataFrame.sort_values()` can accept an optional `by` argument for `axis=0` which will use an arbitrary vector or a column name of the DataFrame to determine the sort order:

```python
In [312]: df1 = pd.DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})

In [313]: df1.sort_values(by='two')
Out[313]:
      one  three  two
0     2     5     1
1     1     4     3
2     1     3     2
3     1     2     4
```

The `by` argument can take a list of column names, e.g.:

```python
In [314]: df1[['one', 'two', 'three']].sort_values(by=['one','two'])
Out[314]:
   one  two  three
2     1     2     3
1     1     3     4
```
These methods have special treatment of NA values via the `na_position` argument:

```python
In [315]: s[2] = np.nan

In [316]: s.sort_values()
Out[316]:
0   A
1   B
2  NaN
3  Aaba
4  Baca
6  CABA
7   dog
2  NaN
5  NaN
dtype: object

In [317]: s.sort_values(na_position='first')
Out[317]:
     0  1  2  3  4  5
  NaN  NaN  A  B  Baca  CABA
dtype: object
```

### 9.11.3 searchsorted

Series has the `searchsorted()` method, which works similar to `numpy.ndarray.searchsorted()`.

```python
In [318]: ser = pd.Series([1, 2, 3])

In [319]: ser.searchsorted([0, 3])
Out[319]: array([0, 2])

In [320]: ser.searchsorted([0, 4])
Out[320]: array([0, 3])

In [321]: ser.searchsorted([1, 3], side='right')
Out[321]: array([1, 3])

In [322]: ser.searchsorted([1, 3], side='left')
Out[322]: array([0, 2])

In [323]: ser = pd.Series([3, 1, 2])

In [324]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
```

---

564  Chapter 9. Essential Basic Functionality
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.

```python
In [325]: s = pd.Series(np.random.permutation(10))

In [326]: s
Out[326]:
   0  3
   1  1
   2  9
   3  6
   4  0
   5  8
   6  5
   7  2
   8  7
   9  4
dtype: int64

In [327]: s.sort_values()
Out[327]:
   4  0
   1  1
   7  2
   0  3
   9  4
   6  5
   3  6
   8  7
   5  8
   2  9
dtype: int64

In [328]: s.nsmallest(3)
Out[328]:
   4  0
   1  1
   7  2
 dtype: int64

In [329]: s.nlargest(3)
Out[329]:
   2  9
   5  8
   8  7
 dtype: int64
```

New in version 0.17.0.

DataFrame also has the `nlargest` and `nsmallest` methods.
In [330]: 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 [331]: df.nlargest(3, 'a')
Out[331]:
      a  b  c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN

In [332]: df.nlargest(5, ['a', 'c'])
Out[332]:
      a  b  c
6  -1  f  4.0
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN
2   1  d  4.0

In [333]: df.nsmallest(3, 'a')
Out[333]:
       a  b  c
0  -2  a  1.0
1  -1  b  2.0
6  -1  f  4.0

In [334]: df.nsmallest(5, ['a', 'c'])
Out[334]:
       a  b  c
0  -2  a  1.0
2   1  d  4.0
4   8  e  NaN
1  -1  b  2.0
6  -1  f  4.0

9.11.5 Sorting by a multi-index column

You must be explicit about sorting when the column is a multi-index, and fully specify all levels to by.

In [335]: df1.columns = pd.MultiIndex.from_tuples([('a','one'),('a','two'),('b','three')])

In [336]: df1.sort_values(by=('a','two'))
Out[336]:
       a  b
one  two  three
3  1  2  4
2  1  3  2
1  1  4  3
0  2  5  1
9.12 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 methods have the side effect of modifying your data; almost all methods return new objects, leaving the original object untouched. If data is modified, it is because you did so explicitly.

9.13 dtypes

The main types stored in pandas objects are `float`, `int`, `bool`, `datetime64[ns]` and `datetime64[ns, tz]` (in >= 0.17.0), `timedelta[ns]`, `category` and `object`. In addition these dtypes have item sizes, e.g. `int64` and `int32`. See `Series with TZ` for more detail on `datetime64[ns, tz]` dtypes.

A convenient `dtypes` attribute for DataFrames returns a Series with the data type of each column.

```
In [337]: dft = pd.DataFrame(dict(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 [338]: dft
Out[338]:
          A      B       C     D             E                 F       G
0  0.534749  1   foo 2001-01-02  1.0   False
1  0.688452  1   foo 2001-01-02  1.0   False
2  0.777842  1   foo 2001-01-02  1.0   False

In [339]: dft.dtypes
Out[339]:
A    float64
B      int64
C    object
D   datetime64[ns]
E      float32
F       bool
G       int8
dtype: object
```

On a Series use the `dtype` attribute.

```
In [340]: dft['A'].dtype
Out[340]: dtype('float64')
```
If a pandas object contains data 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 [341]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[341]:
0  1.0
1  2.0
2  3.0
3  4.0
4  5.0
5  6.0
dtype: float64

# string data forces an 'object' dtype
In [342]: pd.Series([1, 2, 3, 6., 'foo'])
Out[342]:
   0  1
   1  2
   2  3
   3  6
   4  foo
dtype: object
```

The method `get_dtype_counts()` will return the number of columns of each type in a DataFrame:

```python
In [343]: dft.get_dtype_counts()
Out[343]:
bool     1
datetime64[ns]     1
float32    1
float64    1
int64     1
int8      1
object    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 [344]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')
In [345]: df1
Out[345]:
     A
0 -2.038777
1  1.121731
2  0.586626
3 -0.282532
4  0.410238
5 -0.540166
6  1.400679
7 -0.255975
```

```python
In [346]: df1.dtypes
```

---

568 Chapter 9. Essential Basic Functionality
A    float32
dtype: object

In [347]: df2 = pd.DataFrame(dict( 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 [348]: df2
Out[348]:
       A         B         C
0  -0.624512  -1.397492   0
1   0.022354   1.338115   0
2  -0.433594   0.781169  255
3  -0.405762  -0.791687  255
4  -0.149658  -0.764810  255
5   0.644531  -2.000933   0
6  -1.260742  -0.345662   0
7   0.365967   0.393915   0

In [349]: df2.dtypes
Out[349]:
       A         B         C
dtype: object

A float16
B float64
C uint8

dtype: object

9.13.1 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 [350]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[350]:
a    int64
dtype: object

In [351]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[351]:
a    int64
dtype: object

In [352]: pd.DataFrame({'a': 1 }, index=list(range(2))).dtypes
Out[352]:
a    int64
dtype: object

Numpy, however will choose platform-dependent types when creating arrays. The following WILL result in int32 on 32-bit platform.

In [353]: frame = pd.DataFrame(np.array([1, 2]))

9.13. dtypes
9.13.2 upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (say int to float)

```
In [354]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [355]: df3
Out[355]:
   A      B      C
0 -2.663288 -1.397492  0.0
1  1.144085  1.338115  0.0
2  0.153032  0.781169 255.0
3 -0.688294 -0.791687  0.0
4  0.260580 -0.764810 255.0
5  0.104365 -2.000933  0.0
6  0.139937 -0.345662  0.0
7  0.109992  0.393915  0.0
```

```
In [356]: df3.dtypes

Type: object
```

The `values` attribute on a DataFrame return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped numpy array. This can force some upcasting.

```
In [357]: df3.values.dtype
Out[357]: dtype('float64')
```

9.13.3 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 [358]: df3
Out[358]:
   A      B      C
0 -2.663288 -1.397492  0.0
1  1.144085  1.338115  0.0
2  0.153032  0.781169 255.0
3 -0.688294 -0.791687  0.0
4  0.260580 -0.764810 255.0
5  0.104365 -2.000933  0.0
6  0.139937 -0.345662  0.0
7  0.109992  0.393915  0.0
```

```
In [359]: df3.dtypes

Type: object
```
Convert a subset of columns to a specified type using `astype()`

```python
In [361]: dft = pd.DataFrame({'a': [1,2,3], 'b': [4,5,6], 'c': [7, 8, 9]})
In [362]: dft[['a','b']] = dft[['a','b']].astype(np.uint8)
In [363]: dft
Out[363]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9
In [364]: dft.dtypes
Out[364]:
   a  uint8
   b  uint8
   c  int64
dtype: object
```

New in version 0.19.0.

Convert certain columns to a specific dtype by passing a dict to `astype()`

```python
In [365]: dft1 = pd.DataFrame({'a': [1,0,1], 'b': [4,5,6], 'c': [7, 8, 9]})
In [366]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})
In [367]: dft1
Out[367]:
    a   b   c
0   True  4.0
1  False  5.0
2   True  6.0
In [368]: dft1.dtypes
Out[368]:
   a  bool
   b  int64
   c  float64
dtype: object
```
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 [369]: dft = pd.DataFrame({'a': [1,2,3], 'b': [4,5,6], 'c': [7, 8, 9]})

In [370]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out[370]:
a uint8
b uint8
dtype: object

In [371]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [372]: dft.dtypes
Out[372]:
a int64
b int64
c int64
dtype: object
```

### 9.13.4 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 [373]: import datetime

In [374]: df = pd.DataFrame([[[1, 2], ['a', 'b'], [datetime.datetime(2016, 3, 2), datetime.datetime(2016, 3, 2)])])

In [375]: df = df.T

In [376]: df
Out[376]:
   0  1  2  3  4  5  6  7  8  9 10 11
0 a  b  
1 c  d  e

In [377]: df.dtypes
Out[377]:
0 object
1 object
2 object
dtype: object
```

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)

```
In [379]: m = ['1.1', 2, 3]
In [380]: pd.to_numeric(m)
Out[380]: array([1.1, 2. , 3. ])
```

- `to_datetime()` (conversion to datetime objects)

```
In [381]: import datetime
In [382]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]
In [383]: pd.to_datetime(m)
Out[383]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]', freq=None)
```

- `to_timedelta()` (conversion to timedelta objects)

```
In [384]: m = ['5us', pd.Timedelta('1day')]
In [385]: pd.to_timedelta(m)
Out[385]: 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:

```
In [386]: import datetime
In [387]: m = ['apple', datetime.datetime(2016, 3, 2)]
In [388]: pd.to_datetime(m, errors='coerce')
Out[388]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)
In [389]: m = ['apple', 2, 3]
In [390]: pd.to_numeric(m, errors='coerce')
Out[390]: array([nan, 2., 3.])
In [391]: m = ['apple', pd.Timedelta('1day')]
In [392]: pd.to_timedelta(m, errors='coerce')
Out[392]: 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 [393]: import datetime
In [394]: m = ['apple', datetime.datetime(2016, 3, 2)]
In [395]: pd.to_datetime(m, errors='ignore')
Out[395]: array(['apple', datetime.datetime(2016, 3, 2, 0, 0)], dtype=object)
In [396]: m = ['apple', 2, 3]
In [397]: pd.to_numeric(m, errors='ignore')
Out[397]: array(['apple', 2, 3], dtype=object)
In [398]: m = ['apple', pd.Timedelta('1day')]
In [399]: pd.to_timedelta(m, errors='ignore')
Out[399]: 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 [400]: m = ['1', 2, 3]
In [401]: pd.to_numeric(m, downcast='integer')  # smallest signed int dtype
Out[401]: array([1, 2, 3], dtype=int8)
In [402]: pd.to_numeric(m, downcast='signed')  # same as 'integer'
   
Out[402]: array([1, 2, 3], dtype=int8)
In [403]: pd.to_numeric(m, downcast='unsigned')  # smallest unsigned int dtype
   
Out[403]: array([1, 2, 3], dtype=uint8)
In [404]: pd.to_numeric(m, downcast='float')  # smallest float dtype
   
Out[404]: array([1., 2., 3.], dtype=float32)
```

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:

```python
In [405]: import datetime
In [406]: df = pd.DataFrame([['2016-07-09', datetime.datetime(2016, 3, 2)] * 2, 'O')
In [407]: df
Out[407]:
   0 1
0 2016-07-09 2016-03-02 00:00:00
1 2016-07-09 2016-03-02 00:00:00
In [408]: df.apply(pd.to_datetime)
```

---

Chapter 9. Essential Basic Functionality
9.13.5 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
A    int32
B    int32
C    int32
E    int64
dtype: object

In [419]: casted = dfi[dfi>0]

In [420]: casted
Out[420]:
   A   B    C   E
0  NaN  NaN  NaN  1
1  1.0  1.0  NaN  1
2  NaN  NaN  255.0  1
3  NaN  NaN  NaN  1
4  NaN  NaN  255.0  1
5  NaN  NaN  NaN  1
6  NaN  NaN  NaN  1
7  NaN  NaN  NaN  1

In [421]: casted.dtypes

While float dtypes are unchanged.

In [422]: dfa = df3.copy()

In [423]: dfa['A'] = dfa['A'].astype('float32')

In [424]: dfa.dtypes
Out[424]:
     A    B    C   E
dtype: object

In [425]: casted = dfa[df2>0]

In [426]: casted
Out[426]:
   A     B     C
0  NaN  NaN  NaN
1  1.144085  1.338115  NaN
2  NaN   0.781169  255.0
3  NaN   NaN   NaN
4   0.104365  NaN   NaN
5   0.109992  0.393915  NaN

In [427]: casted.dtypes

...
9.14 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:

```python
In [428]: 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).values,
                      'category': pd.Series(list('ABC')).astype('category')})
```

```python
In [429]: df['tdeltas'] = df.dates.diff()
In [430]: df['uint64'] = np.arange(3, 6).astype('u8')
In [431]: df['other_dates'] = pd.date_range('20130101', periods=3).values
In [432]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')
```

```python
In [433]: df
Out[433]:
bool1  bool2  category   dates  float64  int64  string  \
0   True   False      A 2017-10-27 10:34:07.934179  4.0     1     a
1   False   True      B 2017-10-28 10:34:07.934179  5.0     2     b
2   True   False      C 2017-10-29 10:34:07.934179  6.0     3     c
```

And the dtypes:

```python
In [434]: df.dtypes
Out[434]:
bool1  bool
bool2  bool
category category
dates    datetime64[ns]
float64  float64
int64    int64
string   object
uint8    uint8
tdeltas  timedelta64[ns]
uint64   uint64
other_dates    datetime64[ns]
```
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

```python
In [435]: df.select_dtypes(include=[bool])
Out[435]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the numpy dtype hierarchy:

```python
In [436]: df.select_dtypes(include=['bool'])
Out[436]:
   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

```python
In [437]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[437]:
   bool1  bool2  float64  int64  tdeltas
0   True  False  4.0     1   NaT
1  False   True  5.0     2  1 days
2   True  False  6.0     3  1 days
```

To select string columns you must use the object dtype:

```python
In [438]: df.select_dtypes(include=['object'])
Out[438]:
   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:

```python
In [439]: def subtypes(dtype):
       ....:   subs = dtype.__subclasses__()
       ....:   if not subs:
       ....:     return dtype
       ....:   return [dtype, [subtypes(dt) for dt in subs]]
```

All numpy dtypes are subclasses of numpy.generic:

```python
In [440]: subtypes(np.generic)
Out[440]:
```
Note: Pandas also defines the types category, and datetime64[ns, tz], which are not integrated into the normal numpy hierarchy and won't show up with the above function.
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 [1]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [2]: s.str.lower()
Out[2]:
0    a
1    b
2    c
3   aaba
4   baca
5  NaN
6    caba
7     dog
8     cat
dtype: object

In [3]: s.str.upper()

Out[3]:
0    A
1    B
2    C
3   AABA
4   BACA
5  NaN
6    CABA
7     DOG
8     CAT
dtype: object

In [4]: s.str.len()

Out[4]:
0    1.0
1    1.0
2    1.0
3    4.0
4    4.0
5   NaN
6    4.0
7    3.0
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:

```python
In [9]: df = pd.DataFrame(randn(3, 2), columns=[' Column A ', ' Column B '], index=range(3))

In [10]: df
Out[10]:
   Column A    Column B
0  1.425575  -1.336299
1  0.740933   1.032121
2 -1.585660   0.913812
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```python
In [11]: df.columns.str.strip()
Out[11]: Index(['Column A', 'Column B'], dtype='object')

In [12]: df.columns.str.lower()
Out[12]: 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, lowercasing all names, and replacing any remaining whitespaces with underscores:

```python
In [13]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [14]: df
Out[14]:
   column_a    column_b
0  -1.425575  -1.336299
1   0.740933   1.032121
2  -1.585660   0.913812
```

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 of 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.

10.1 Splitting and Replacing Strings

Methods like split return a Series of lists:

```python
In [15]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])
In [16]: s2.str.split('_')
Out[16]:
   0    [a, b, c]
   1    [c, d, e]
   2      NaN
   3    [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using get or [] notation:

```python
In [17]: s2.str.split('_').str.get(1)
Out[17]:
   0    b
   1    d
   2  NaN
   3    g
dtype: object
```

```python
In [18]: s2.str.split('_').str[1]
```

Easy to expand this to return a DataFrame using expand.

```python
In [19]: s2.str.split('_', expand=True)
Out[19]:
   0     1     2
0    a    b    c
1    c    d    e
2  NaN   None  None
3    f    g    h
```

It is also possible to limit the number of splits:

```python
In [20]: s2.str.split('_', expand=True, n=1)
Out[20]:
   0  1
0    a  b\_c
1    c  d\_e
2  NaN   None
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:

```python
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
0  |
0  a_b  c
1  c_d  e
2  NaN  None
3  f_g  h
```

Methods like replace and.findall take regular expressions, too:

```python
In [22]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', '', np.nan, 'CABA', 'dog', 'cat'])
In [23]: s3
Out[23]:
0  A
1  B
2  C
3  Aaba
4  Baca
5  NaN
6  CABA
7  dog
8  cat
dtype: object
In [24]: s3.str.replace('^\w|dog', 'XX-XX ', case=False)
```

Some caution must be taken to keep regular expressions in mind! For example, the following code will cause trouble because of the regular expression meaning of $:

```python
# Consider the following badly formatted financial data
In [25]: dollars = pd.Series(['12', '-$10', '$10,000'])
# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
```

```python
0  12
1  -10
2  10,000
```
The `replace` method can also take a callable as replacement. It is called on every pattern using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

New in version 0.20.0.

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.

New in version 0.20.0.
Including a flags argument when calling replace with a compiled regular expression object will raise a `ValueError`.

```python
In [38]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

### 10.2 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.

```python
In [39]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, '
.....:       'CABA', 'dog', 'cat'])
.....:

In [40]: s.str[0]
Out[40]:
0   A
1   B
2   C
3   A
4   B
5  NaN
6   C
7   d
8   c
dtype: object

In [41]: s.str[1]
```

```bash
→
0  NaN
1  NaN
2  NaN
3   a
4   a
5  NaN
6   A
7   o
8   a
dtype: object
```
10.3 Extracting Substrings

10.3.1 Extract first match in each subject (extract)

**Warning:** In version 0.18.0, `extract` gained the `expand` argument. When `expand=False` it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0). When `expand=True` it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user.

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.

```
In [42]: pd.Series(['a1', 'b2', 'c3']).str.extract('([ab])\((\d)\)', expand=False)
Out[42]:
   0 1
0  a 1
1  b 2
2  NaN NaN
```

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 [43]: pd.Series(['a1', 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)', expand=False)
Out[43]:
   letter  digit
0      a      1
1      b      2
2  NaN  NaN
```

and optional groups like

```
In [44]: pd.Series(['a1', 'b2', '3']).str.extract('([ab])?(\d)', expand=False)
Out[44]:
   0  1
0  a  1
1  b  2
2  NaN  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 [45]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab]\((\d)\)', expand=True)
Out[45]:
   0
0  1
1  2
2  NaN
```
It returns a Series if expand=False.

```
In [46]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=False)
Out[46]:
   0  1
   1  2
   2 NaN
dtype: object
```

Calling on an Index with a regex with exactly one capture group returns a DataFrame with one column if expand=True.

```
In [47]: s = pd.Series(['a1', 'b2', 'c3'], ['A11', 'B22', 'C33'])
In [48]: s
Out[48]:
A11 a1
B22 b2
C33 c3
dtype: object
In [49]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
```

It returns an Index if expand=False.

```
In [50]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[50]: Index(['A', 'B', 'C'], dtype='object', name='letter')
```

Calling on an Index with a regex with more than one capture group returns a DataFrame if expand=True.

```
In [51]: s.index.str.extract("(?P<letter>[a-zA-Z])([-0-9]+)", expand=True)
Out[51]:
   letter  1
    0      A  11
    1      B  22
    2      C  33
```

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>Input</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>

### 10.3.2 Extract all matches in each subject (extractall)

New in version 0.18.0.
Unlike `extract` (which returns only the first match),

```python
In [52]: s = pd.Series(['a1a2', 'b1', 'c1'], index=['A', 'B', 'C'])
In [53]: s
Out[53]:
      A   B   C
0  a1a2  b1  c1

In [54]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'
In [55]: s.str.extract(two_groups, expand=True)
Out[55]:
       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.

```python
In [56]: s.str.extractall(two_groups)
Out[56]:
         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,

```python
In [57]: s = pd.Series(['a3', 'b3', 'c2'])
In [58]: s
Out[58]:
     0    1    2
  a3 b3 c2
```

then `extractall(pat).xs(0, level='match')` gives the same result as `extract(pat)`.

```python
In [59]: extract_result = s.str.extract(two_groups, expand=True)
In [60]: extract_result
Out[60]:
         letter digit
    0     a     3
    1     b     3
    2     c     2
```

```python
In [61]: extractall_result = s.str.extractall(two_groups)
In [62]: extractall_result
```

10.3. Extracting Substrings
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).

New in version 0.19.0.

### 10.4 Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

```python
In [66]: pattern = r'\d[a-z]'

In [67]: pd.Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
Out[67]:
0   False
1   False
2    True
3    True
4    True
dtype: bool
```
The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match`, while `contains` relies on `re.search`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```python
In [69]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
In [70]: s4.str.contains('A', na=False)
Out[70]:
0   True
1  False
2  False
3   True
4  False
5  False
6   True
7  False
8  False
dtype: bool
```

10.5 Creating Indicator Variables

You can extract dummy variables from string columns. For example if they are separated by a `|`:

```python
In [71]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])
In [72]: s.str.get_dummies(sep='|')
Out[72]:
     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.

New in version 0.18.1.

```python
In [73]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])
In [74]: idx.str.get_dummies(sep='|')
Out[74]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
lables=[[1, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 1]],
names=['a', 'b', 'c'])
```
See also `get_dummies()`.

## 10.6 Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat()</code></td>
<td>Concatenate strings</td>
</tr>
<tr>
<td><code>split()</code></td>
<td>Split strings on delimiter</td>
</tr>
<tr>
<td><code>rsplit()</code></td>
<td>Split strings on delimiter working from the end of the string</td>
</tr>
<tr>
<td><code>get()</code></td>
<td>Index into each element (retrieve i-th element)</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>Join strings in each element of the Series with passed separator</td>
</tr>
<tr>
<td><code>get_dummies()</code></td>
<td>Split strings on the delimiter returning DataFrame of dummy variables</td>
</tr>
<tr>
<td><code>contains()</code></td>
<td>Return boolean array if each string contains pattern/regex</td>
</tr>
<tr>
<td><code>replace()</code></td>
<td>Replace occurrences of pattern/regex with some other string or the return value of a callable given the occurrence</td>
</tr>
<tr>
<td><code>repeat()</code></td>
<td>Duplicate values (s.str.repeat(3) equivalent to x * 3)</td>
</tr>
<tr>
<td><code>pad()</code></td>
<td>Add whitespace to left, right, or both sides of strings</td>
</tr>
<tr>
<td><code>center()</code></td>
<td>Equivalent to <code>str.center</code></td>
</tr>
<tr>
<td><code>ljust()</code></td>
<td>Equivalent to <code>str.ljust</code></td>
</tr>
<tr>
<td><code>rjust()</code></td>
<td>Equivalent to <code>str.rjust</code></td>
</tr>
<tr>
<td><code>zfill()</code></td>
<td>Equivalent to <code>str.zfill</code></td>
</tr>
<tr>
<td><code>wrap()</code></td>
<td>Split long strings into lines with length less than a given width</td>
</tr>
<tr>
<td><code>slice()</code></td>
<td>Slice each string in the Series</td>
</tr>
<tr>
<td><code>slice_replace()</code></td>
<td>Replace slice in each string with passed value</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Count occurrences of pattern</td>
</tr>
<tr>
<td><code>startswith()</code></td>
<td>Equivalent to <code>str.startswith(pat)</code> for each element</td>
</tr>
<tr>
<td><code>endswith()</code></td>
<td>Equivalent to <code>str.endswith(pat)</code> for each element</td>
</tr>
<tr>
<td><code>findall()</code></td>
<td>Compute list of all occurrences of pattern/regex for each string</td>
</tr>
<tr>
<td><code>match()</code></td>
<td>Call <code>re.match</code> on each element, returning matched groups as list</td>
</tr>
<tr>
<td><code>extract()</code></td>
<td>Call <code>re.search</code> on each element, returning DataFrame with one row for each element and one column for each match and one column for each regex capture group</td>
</tr>
<tr>
<td><code>extractall()</code></td>
<td>Call <code>re.findall</code> on each element, returning DataFrame with one row for each match and one column for each regex capture group</td>
</tr>
<tr>
<td><code>len()</code></td>
<td>Compute string lengths</td>
</tr>
<tr>
<td><code>strip()</code></td>
<td>Equivalent to <code>str.strip</code></td>
</tr>
<tr>
<td><code>rstrip()</code></td>
<td>Equivalent to <code>str.rstrip</code></td>
</tr>
<tr>
<td><code>lstrip()</code></td>
<td>Equivalent to <code>str.lstrip</code></td>
</tr>
<tr>
<td><code>partition()</code></td>
<td>Equivalent to <code>str.partition</code></td>
</tr>
<tr>
<td><code>rpartition()</code></td>
<td>Equivalent to <code>str.rpartition</code></td>
</tr>
<tr>
<td><code>lower()</code></td>
<td>Equivalent to <code>str.lower</code></td>
</tr>
<tr>
<td><code>upper()</code></td>
<td>Equivalent to <code>str.upper</code></td>
</tr>
<tr>
<td><code>find()</code></td>
<td>Equivalent to <code>str.find</code></td>
</tr>
<tr>
<td><code>rfind()</code></td>
<td>Equivalent to <code>str.rfind</code></td>
</tr>
<tr>
<td><code>index()</code></td>
<td>Equivalent to <code>str.index</code></td>
</tr>
<tr>
<td><code>rindex()</code></td>
<td>Equivalent to <code>str.rindex</code></td>
</tr>
<tr>
<td><code>capitalize()</code></td>
<td>Equivalent to <code>str.capitalize</code></td>
</tr>
<tr>
<td><code>swapcase()</code></td>
<td>Equivalent to <code>str.swapcase</code></td>
</tr>
<tr>
<td><code>normalize()</code></td>
<td>Return Unicode normal form. Equivalent to <code>unicodedata.normalize</code></td>
</tr>
<tr>
<td><code>translate()</code></td>
<td>Equivalent to <code>str.translate</code></td>
</tr>
<tr>
<td><code>isalnum()</code></td>
<td>Equivalent to <code>str.isalnum</code></td>
</tr>
<tr>
<td><code>isalpha()</code></td>
<td>Equivalent to <code>str.isalpha</code></td>
</tr>
<tr>
<td><code>isdigit()</code></td>
<td>Equivalent to <code>str.isdigit</code></td>
</tr>
<tr>
<td><code>isspace()</code></td>
<td>Equivalent to <code>str.isspace</code></td>
</tr>
</tbody>
</table>
Table 10.1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>islower()</code></td>
<td>Equivalent to <code>str.islower</code></td>
</tr>
<tr>
<td><code>isupper()</code></td>
<td>Equivalent to <code>str.isupper</code></td>
</tr>
<tr>
<td><code>istitle()</code></td>
<td>Equivalent to <code>str.istitle</code></td>
</tr>
<tr>
<td><code>isnumeric()</code></td>
<td>Equivalent to <code>str.isnumeric</code></td>
</tr>
<tr>
<td><code>isdecimal()</code></td>
<td>Equivalent to <code>str.isdecimal</code></td>
</tr>
</tbody>
</table>
11.1 Overview

pandas has an options system that lets you customize some aspects of its behaviour, display-related options being those the user is most likely to adjust.

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]: 15
In [3]: pd.options.display.max_rows = 999
In [4]: pd.options.display.max_rows
Out[4]: 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.py for more info.

All of the functions above accept a regexp pattern (re.search style) as an argument, so passing in a substring will work as long as it is unambiguous:

```python
In [5]: pd.get_option("display.max_rows")
Out[5]: 999
In [6]: pd.set_option("display.max_rows",101)
In [7]: pd.get_option("display.max_rows")
Out[7]: 101
In [8]: pd.set_option("max_r",102)
In [9]: pd.get_option("display.max_rows")
Out[9]: 102
```
The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```python
In [10]: try:
    ....:     pd.get_option("column")
    ....:     except KeyError as e:
    ....:         print(e)
    ....:
'Pattern matched multiple keys'
```

**Note:** Using this form of shorthand may cause your code to break if new options with similar names are added in future versions.

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.

## 11.2 Getting and Setting Options

As described above, `get_option()` and `set_option()` are available from the pandas namespace. To change an option, call `set_option('option regex', new_value)`

```python
In [11]: pd.get_option('mode.sim_interactive')
Out[11]: False

In [12]: pd.set_option('mode.sim_interactive', True)

In [13]: pd.get_option('mode.sim_interactive')
Out[13]: True
```

**Note:** that the option ‘mode.sim_interactive’ is mostly used for debugging purposes.

All options also have a default value, and you can use `reset_option` to do just that:

```python
In [14]: pd.get_option("display.max_rows")
Out[14]: 60

In [15]: pd.set_option("display.max_rows", 999)

In [16]: pd.get_option("display.max_rows")
Out[16]: 999

In [17]: pd.reset_option("display.max_rows")

In [18]: pd.get_option("display.max_rows")
Out[18]: 60
```

It's also possible to reset multiple options at once (using a regex):

```python
In [19]: pd.reset_option("^display")
```

`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:

```python
In [20]: with pd.option_context("display.max_rows",10,"display.max_columns", 5):
    ....:         print(pd.get_option("display.max_rows"))
    ....:         print(pd.get_option("display.max_columns"))
    ....:         print(pd.get_option("display.max_colwidth"))
```

---

**pandas:** powerful Python data analysis toolkit, Release 0.21.0
In 

5

In [21]: print(pd.get_option("display.max_rows"))
   \\\n   \60

In [22]: print(pd.get_option("display.max_columns"))
   \\\\\20

11.3 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:

$IPYTHONDIR/profile_default/startup

More information can be found in the ipython documentation. An example startup script for pandas is displayed below:

```
import pandas as pd
pd.set_option('display.max_rows', 999)
pd.set_option('precision', 5)
```

11.4 Frequently Used Options

The following is a walkthrough of 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 [23]: df = pd.DataFrame(np.random.randn(7,2))
In [24]: pd.set_option('max_rows', 7)
In [25]: df
Out[25]:
   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 [26]: pd.set_option('max_rows', 5)
In [27]: df
Out[27]:
   0   1
0  0.469112 -0.282863
1 -1.509059 -1.135632

.. pandas: powerful Python data analysis toolkit, Release 0.21.0

.. |disp| image:: unknown
   :width: 100%

.. figure:: unknown
   :width: 100%

display\_expand\_frame\_repr allows for the the representation of dataframes to stretch across pages, wrapped over the full column vs row-wise.

```python
In [29]: df = pd.DataFrame(np.random.randn(5,10))
In [30]: pd.set_option('expand_frame_repr', True)
In [31]: df
Out[31]:
     0    1    2    3    4    5    6    7    8    9
0 -1.039575 0.271860 -0.424972 0.567020 0.276232 -1.087401 -0.673690
1  0.404705 0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312
2  1.643563 -1.469388 0.357021 -0.674600 -1.776904 -0.968914 -1.294524
3 -0.013960 -0.362543 -0.006154 -0.923061 0.895717 0.805244 -1.206412
4 -1.170299 -0.226169 0.410835 0.813850 0.132003 -0.827317 -0.076467

In [32]: pd.set_option('expand_frame_repr', False)
In [33]: df
Out[33]:
      0  1 2 3 4 5 6 7 8 9
0 0 -1.039575 0.271860 -0.424972 0.567020 0.276232 -1.087401 -0.673690 0.113648 -1.478427 0.524988
1 1 0.404705 0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312 0.844885 1.075770 -0.109050
2 2 1.643563 -1.469388 0.357021 -0.674600 -1.776904 -0.968914 -1.294524 0.413738 0.276662 -0.472035
3 3 2.565646 1.431256 1.340309
4 4 -1.187678 1.130127 -1.436737

In [34]: pd.reset_option('expand_frame_repr')
```

display\_large\_repr lets you select whether to display dataframes that exceed \texttt{max\_columns} or \texttt{max\_rows} as a truncated frame, or as a summary.

```python
In [35]: df = pd.DataFrame(np.random.randn(10,10))
In [36]: pd.set_option('max\_rows', 5)
In [37]: pd.set_option('large\_repr', 'truncate')
```
In [38]: df
Out[38]:
          0         1         2         3         4         5         6
0  -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466
1   0.545952 -1.219217 -1.226825  0.769804 -1.281247 -0.727707 -0.121306
..    ...       ...       ...       ...       ...       ...    ...
8 -2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349  1.474071
9  -1.071357  0.441153  2.353925  0.583787  0.221471 -0.744471  0.758527
       7       8       9
0 -2.006747 -0.410001 -0.078638
1  0.097883  0.695775  0.341734
..    ...       ...       ...
8 -0.064034 -1.282782  0.781836
9  1.729689 -0.964980 -0.845696
[10 rows x 10 columns]

In [39]: pd.set_option('large_repair', 'info')

In [40]: df
Out[40]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
   0 10 non-null float64
   1 10 non-null float64
   2 10 non-null float64
   3 10 non-null float64
   4 10 non-null float64
   5 10 non-null float64
   6 10 non-null float64
   7 10 non-null float64
   8 10 non-null float64
   9 10 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [41]: pd.reset_option('large_repair')

In [42]: pd.reset_option('max_rows')

display.max_colwidth sets the maximum width of columns. Cells of this length or longer will be truncated with an ellipsis.

In [43]: df = pd.DataFrame(np.array([[['foo', 'bar', 'bim', 'uncomfortably long string ...'], ['horse', 'cow', 'banana', 'apple']]]))

In [44]: pd.set_option('max_colwidth', 40)

In [45]: df
Out[45]:
       0      1      2      3
0  horse   bar    bim  uncomfortably long string
1  apple   banana  'uncomfortably long string

11.4. Frequently Used Options
In [46]: pd.set_option('max_colwidth', 6)

In [47]: df
Out[47]:
   0   1   2   3
0  foo  bar  bim  un...
1  horse  cow  ba...  apple

In [48]: pd.reset_option('max_colwidth')

display.max_info_columns sets a threshold for when by-column info will be given.

In [49]: df = pd.DataFrame(np.random.randn(10,10))

In [50]: pd.set_option('max_info_columns', 11)

In [51]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0   10 non-null float64
   0.0
1   10 non-null float64
   1.0
2   10 non-null float64
   0.0
3   10 non-null float64
   1.0
4   10 non-null float64
   0.0
5   10 non-null float64
   1.0
6   10 non-null float64
   0.0
7   10 non-null float64
   1.0
8   10 non-null float64
   0.0
9   10 non-null float64
   1.0
dtypes: float64(10)
memory usage: 880.0 bytes

In [52]: pd.set_option('max_info_columns', 5)

In [53]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Columns: 10 entries, 0 to 9
dtypes: float64(10)
memory usage: 880.0 bytes

In [54]: pd.reset_option('max_info_columns')

display.max_info_rows: 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. Note that you can specify the option df.info(null_counts=True) to override on showing a particular frame.

In [55]: df = pd.DataFrame(np.random.choice([0,1,np.nan], size=(10,10)))

In [56]: df
Out[56]:
    0  1  2  3  4  5  6  7  8  9
0  0.0 1.0 1.0 0.0 1.0 0.0 NaN 1.0 NaN
1  1.0 NaN 0.0 0.0 1.0 1.0 NaN 1.0 0.0 1.0
2  NaN NaN NaN 1.0 1.0 0.0 NaN 0.0 1.0 NaN
In [57]: pd.set_option('max_info_rows', 11)

In [58]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0 8 non-null float64
1 5 non-null float64
2 8 non-null float64
3 7 non-null float64
4 5 non-null float64
5 7 non-null float64
6 6 non-null float64
7 6 non-null float64
8 8 non-null float64
9 3 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [59]: pd.set_option('max_info_rows', 5)

In [60]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0 float64
1 float64
2 float64
3 float64
4 float64
5 float64
6 float64
7 float64
8 float64
9 float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [61]: pd.reset_option('max_info_rows')

display.precision sets the output display precision in terms of decimal places. This is only a suggestion.

In [62]: df = pd.DataFrame(np.random.randn(5,5))

In [63]: pd.set_option('precision',7)

In [64]: df
Out[64]:
0 1 2 3 4
0 -2.0490276 2.8466122 -1.2080493 -0.4503923 2.4239054

11.4. Frequently Used Options
In [65]: pd.set_option('precision', 4)

In [66]: df
Out[66]:
   0  1     2  3     4
0 -2.0490 2.8466 -1.2080 -0.4504 2.4239
1  0.1211 0.2669 0.8438 -0.2225 2.0220
2 -0.7168 -2.2245 -1.0611 -0.2328 0.4308
3 -0.6655 1.8298 -1.4065 1.0782 0.3228
4  0.2003 0.8900 0.1948 0.3516 0.4489

display.chop_threshold sets at what level pandas rounds to zero when it displays a Series of DataFrame. Note, this does not effect the precision at which the number is stored.

In [67]: df = pd.DataFrame(np.random.randn(6,6))

In [68]: pd.set_option('chop_threshold', 0)

In [69]: df
Out[69]:
   0  1     2  3     4  5
0 -0.1979 0.9657 -1.5229 -0.1166 0.2956 -1.0477
1  1.6406 1.9058  2.7721  0.0888 1.1442  0.6334
2  0.9254 -0.0064 -0.8204  0.6009  1.0393  0.8248
3 -0.8241  0.3377  0.9278  0.8401  0.2485  0.1093
4  0.4320  0.4607  0.3365  3.2076  1.5359  0.4098
5  0.6731  0.7411  0.1109  2.6729  0.8645  0.0609

In [70]: pd.set_option('chop_threshold', .5)

In [71]: df
Out[71]:
   0  1     2  3     4  5
0  0.0000 0.9657 -1.5229  0.0000  0.0000  0.2956 -1.0477
1  1.6406 1.9058  2.7721  0.0000  0.0000  1.1442  0.6334
2  0.9254  0.0000 -0.8204  0.6009  0.0000  1.0393  0.8248
3 -0.8241  0.0000 -0.9278  0.8401  0.0000  0.2485  0.1093
4  0.0000  0.0000  0.0000  3.2076  1.5359  0.0000  0.0000
5 -0.6731  0.7411  0.0000  2.6729  0.8645  0.0000  0.0000

In [72]: pd.reset_option('chop_threshold')

display.colheader_justify controls the justification of the headers. Options are ‘right’, and ‘left’.

In [73]: df = pd.DataFrame(np.array([np.random.randn(6), np.random.randint(1,9,6)*.1,                   
                      np.zeros(6)]).T,
                      columns=['A', 'B', 'C'], dtype='float')

In [74]: pd.set_option('colheader_justify', 'right')

In [75]: df
Out[75]:
   0  1     2  3     4  5
0 -0.1217 0.8697 -0.5030 -0.0440 0.3363 0.000
1  1.9084 1.1980  2.2245  0.0000 0.0000 0.000
2  0.8999  0.0000 -0.8850  0.0000 0.0000 0.000
3 -0.7444  0.0000 -1.0279  0.0000 0.0000 0.000
4  0.0000  0.0000  0.0000  0.0000 0.0000 0.000
5 -0.6772  0.7411  0.0000  2.6729  0.8645  0.000
In [76]: pd.set_option('colheader_justify', 'left')

In [77]: df

Out[77]:
   A    B    C
0  0.9331  0.3  0.0
1  0.2888  0.2  0.0
2  1.3250  0.2  0.0
3  0.5892  0.7  0.0
4  0.5314  0.1  0.0
5 -1.1987  0.7  0.0

In [78]: pd.reset_option('colheader_justify')

11.5 Available Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>display.chop_threshold</td>
<td>None</td>
<td>If set to a float value, all float values smaller then the given threshold will be displayed as exactly 0.</td>
</tr>
<tr>
<td>display.colheader_justify</td>
<td>right</td>
<td>Controls the justification of column headers, used by DataFrameFormatter.</td>
</tr>
<tr>
<td>display.column_space</td>
<td>12</td>
<td>No description available.</td>
</tr>
<tr>
<td>display.date_dayfirst</td>
<td>False</td>
<td>When True, prints and parses dates with the day first, eg 20/01/2005</td>
</tr>
<tr>
<td>display.date_yearfirst</td>
<td>False</td>
<td>When True, prints and parses dates with the year first, eg 2005/01/20</td>
</tr>
<tr>
<td>display.encoding</td>
<td>UTF-8</td>
<td>Defaults to the detected encoding of the console.</td>
</tr>
<tr>
<td>display.expand_frame_repr</td>
<td>True</td>
<td>Whether to print out the full DataFrame repr for wide DataFrames across multiple lines.</td>
</tr>
<tr>
<td>display.float_format</td>
<td>None</td>
<td>The callable should accept a floating point number and return a string with the desired format of the number.</td>
</tr>
<tr>
<td>display.large_repr</td>
<td>truncate</td>
<td>For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table.</td>
</tr>
<tr>
<td>display.latex.escape</td>
<td>True</td>
<td>Escapes special characters in DataFrames, when using the to_latex method.</td>
</tr>
<tr>
<td>display.latex.longtable</td>
<td>False</td>
<td>Specifies if the to_latex method of a DataFrame uses the longtable format.</td>
</tr>
<tr>
<td>display.latex.multicolumn</td>
<td>True</td>
<td>Combines columns when using a MultiIndex.</td>
</tr>
<tr>
<td>display.latex.multicolumn_format</td>
<td>T</td>
<td>Alignment of multicolumn labels</td>
</tr>
<tr>
<td>display.latex.multirow</td>
<td>False</td>
<td>Combines rows when using a MultiIndex. Centered instead of top-aligned, separated by horizontal rules.</td>
</tr>
<tr>
<td>display.max_columns</td>
<td>20</td>
<td>max_rows and max_columns are used in <strong>repr</strong>() methods to decide if to_string() should be used.</td>
</tr>
<tr>
<td>display.max_columnwidth</td>
<td>50</td>
<td>The maximum width in characters of a column in the repr of a pandas data structure.</td>
</tr>
<tr>
<td>display.max_info_columns</td>
<td>100</td>
<td>max_info_columns is used in DataFrame.info method to decide if per column information should be displayed.</td>
</tr>
<tr>
<td>display.max_info_rows</td>
<td>1690785</td>
<td>df.info() will usually show null-counts for each column. For large frames this can be much slower.</td>
</tr>
<tr>
<td>display.max_rows</td>
<td>60</td>
<td>This sets the maximum number of rows pandas should output when printing out various representations.</td>
</tr>
<tr>
<td>display.max_seq_items</td>
<td>100</td>
<td>When pretty-printing a long sequence, no more then max_seq_items will be printed.</td>
</tr>
<tr>
<td>display.memory_usage</td>
<td>True</td>
<td>This specifies if the memory usage of a DataFrame should be displayed when the df.info() method is called.</td>
</tr>
<tr>
<td>display.multi_sparse</td>
<td>True</td>
<td>“Sparsify” MultiIndex display (don’t display repeated elements in outer levels within a MultiIndex).</td>
</tr>
<tr>
<td>display.notebook_repr_html</td>
<td>True</td>
<td>When True, IPython notebook will use html representation for pandas objects (if it is available).</td>
</tr>
<tr>
<td>display.pprint_nest_depth</td>
<td>3</td>
<td>Controls the number of nested levels to process when pretty-printing.</td>
</tr>
</tbody>
</table>
11.6 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.

For instance:

In [79]: import numpy as np
In [80]: pd.set_eng_float_format(accuracy=3, use_eng_prefix=True)
In [81]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [82]: s/1.e3
Out[82]:
    a   -236.866u
    b    846.974u
    c   -685.597u
    d    609.099u
    e   -303.961u
          dtype: float64

In [83]: s/1.e6
Out[83]:
   a  -236.866n
   b    846.974n
   c   -685.597n
   d    609.099n
   e   -303.961n
          dtype: float64

To round floats on a case-by-case basis, you can also use round() and round().

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>display.precision</td>
<td>6</td>
<td>Floating point output precision in terms of number of places after the decimal, for regular formatting as well as scientific notation. Similar to numpy's precision print option.</td>
</tr>
<tr>
<td>display.show_dimensions</td>
<td>truncate</td>
<td>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).</td>
</tr>
<tr>
<td>display.width</td>
<td>80</td>
<td>Width of the display in characters. In case python/IPython is running in a terminal the width is determined by the prompt width. In other cases it is determined by using xwpgen or searching for 'width' in the source of the ipython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width.</td>
</tr>
<tr>
<td>display.html.table_schema</td>
<td>False</td>
<td>Whether to publish a Table Schema representation for frontends that support it.</td>
</tr>
<tr>
<td>display.html.border</td>
<td>1</td>
<td>A border=value attribute is inserted in the &lt;table&gt; tag for the DataFrame HTML repr.</td>
</tr>
<tr>
<td>io.excel.xls.writer</td>
<td>xlwt</td>
<td>The default Excel writer engine for 'xls' files. Available options: 'xlwt' (the default).</td>
</tr>
<tr>
<td>io.excel.xlsm.writer</td>
<td>openpyxl</td>
<td>The default Excel writer engine for 'xlsm' files. Available options: 'openpyxl' (the default).</td>
</tr>
<tr>
<td>io.excel.xlsx.writer</td>
<td>openpyxl</td>
<td>The default Excel writer engine for 'xlsx' files. Available options: 'openpyxl' (the default).</td>
</tr>
<tr>
<td>io.hdf.default_format</td>
<td>None</td>
<td>default format writing format, if None, then put default to 'fixed' and append with 'fixed' if needed.</td>
</tr>
<tr>
<td>io.hdf.dropna_table</td>
<td>True</td>
<td>drop ALL nan rows when appending to a table.</td>
</tr>
<tr>
<td>io.parquet.engine</td>
<td>None</td>
<td>The engine to use as a default for parquet reading and writing. If None then try 'pyarrow' as default.</td>
</tr>
<tr>
<td>mode.chained_assignment</td>
<td>warn</td>
<td>Raise an exception, warn, or no action if trying to use chained assignment, The default is warn.</td>
</tr>
<tr>
<td>mode.sim_interactive</td>
<td>False</td>
<td>Whether to simulate interactive mode for purposes of testing.</td>
</tr>
<tr>
<td>mode.use_inf_as_na</td>
<td>False</td>
<td>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).</td>
</tr>
<tr>
<td>compute.use_bottleneck</td>
<td>True</td>
<td>Use the bottleneck library to accelerate computation if it is installed.</td>
</tr>
<tr>
<td>compute.use_numexpr</td>
<td>True</td>
<td>Use the numexpr library to accelerate computation if it is installed.</td>
</tr>
</tbody>
</table>
11.7 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.

**Note:** Screen captures are attached for each output to show the actual results.

```python
In [84]: df = pd.DataFrame({u'': ['UK', u''], u'': ['Alice', u'']})
In [85]: df;
```

```shell
>>> df = pd.DataFrame({u'国籍': ['UK', u'日本'], u'名前': ['Alice', u'しのぶ']})
>>> df
 名前 国籍
0 Alice UK
1 しのぶ 日本
```

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 [86]: pd.set_option('display.unicode.east_asian_width', True)
In [87]: df;
```

```shell
>>> pd.set_option('display.unicode.east_asian_width', True)
>>> df
```

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 [88]: df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
In [89]: df;
```

```shell
>>> df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
>>> df
 a  b
0 xxx yyy
1 ¡¡ ¡¡
```

11.7. Unicode Formatting 605
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 [90]: pd.set_option('display.unicode.ambiguous_as_wide', True)
In [91]: df;
```

```plaintext
>>> pd.set_option('display.unicode.ambiguous_as_wide', True)
>>> df
    a  b
0  xxx  yyy
1   ii  ii
```

### 11.8 Table Schema Display

New in version 0.20.0.

`DataFrame` and `Series` will publish a Table Schema representation by default. False by default, this can be enabled globally with the `display.html.table_schema` option:

```python
In [92]: pd.set_option('display.html.table_schema', True)
```

Only 'display.max_rows' are serialized and published.
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. Expect more work to be invested in higher-dimensional data structures (including Panel) in the future, especially in label-based advanced indexing.

**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

**Warning:** Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see [here](#).

See the MultiIndex / Advanced Indexing for MultiIndex and more advanced indexing documentation.

See the cookbook for some advanced strategies

### 12.1 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! - also see Slicing with labels)
- A boolean array
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

    New in version 0.18.1.

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
    - A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

    New in version 0.18.1.

See more at Selection by Position

See more at 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 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 equiv 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>
<tr>
<td>Panel</td>
<td>p.loc[item_indexer, major_indexer, minor_indexer]</td>
</tr>
</tbody>
</table>

### 12.2 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. Thus,

<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>
<tr>
<td>Panel</td>
<td>panel[itemname]</td>
<td>DataFrame corresponding to the itemname</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]:

<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>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-0.044236</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.370647</td>
<td>-1.157892</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
</tbody>
</table>

In [4]: panel = pd.Panel({'one': df, 'two': df - df.mean()})

In [5]: panel
Out[5]:

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D

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 [6]: s = df['A']

In [7]: s[dates[5]]
Out[7]: -0.67368970808837059

In [8]: panel['two']

<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>0.409571</td>
<td>0.113086</td>
<td>-0.610826</td>
<td>-0.936507</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.152571</td>
<td>0.222735</td>
<td>1.017442</td>
<td>-0.845111</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.921390</td>
<td>-1.708620</td>
<td>0.403304</td>
<td>1.270929</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.662014</td>
<td>-0.310822</td>
<td>-0.141342</td>
<td>0.470985</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.484513</td>
<td>0.962970</td>
<td>1.174465</td>
<td>-0.888276</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.733231</td>
<td>0.509598</td>
<td>-0.580194</td>
<td>0.724113</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.345164</td>
<td>0.972995</td>
<td>-0.816769</td>
<td>-0.840143</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.430188</td>
<td>-0.761943</td>
<td>-0.446079</td>
<td>1.044010</td>
</tr>
</tbody>
</table>

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 [9]: df
Out[9]:

<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>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
</tbody>
</table>
In [10]: `df[['B', 'A']] = df[['A', 'B']]`

In [11]: `df`

Out[11]:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.282863</td>
<td>0.469112</td>
<td>-1.509059</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
<td>0.119209</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-2.104569</td>
<td>-0.861849</td>
<td>-0.494929</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.721555</td>
<td>-1.039575</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
<td>0.276232</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
<td>-0.673690</td>
<td>-1.478427</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
<td>0.404705</td>
<td>-1.715002</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.157892</td>
<td>-0.370647</td>
<td>-1.344312</td>
</tr>
</tbody>
</table>

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 [12]: `df[['A', 'B']]`

Out[12]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.282863</td>
<td>0.469112</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-2.104569</td>
<td>-0.861849</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.721555</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
<td>-0.673690</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
<td>0.404705</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.157892</td>
<td>-0.370647</td>
</tr>
</tbody>
</table>

In [13]: `df.loc[:, ['B', 'A']] = df[['A', 'B']]`

In [14]: `df[['A', 'B']]`

Out[14]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.282863</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-2.104569</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.157892</td>
</tr>
</tbody>
</table>

The correct way is to use raw values
12.3 Attribute Access

You may access an index on a Series, column on a DataFrame, and an item on a Panel directly as an attribute:

```
In [17]: sa = pd.Series([1,2,3],index=list('abc'))
In [18]: dfa = df.copy()
```

```
In [19]: sa.b
Out[19]:

In [20]: dfa.A
Out[20]:

In [21]: panel.one
```

```
In [22]: sa.a = 5
In [23]: sa
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

Out[23]:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

dtype: int64

In [24]: dfa.A = list(range(len(dfa.index))) # ok if A already exists

In [25]: dfa

Out[25]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td></td>
<td>1 -0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td></td>
<td>2 -2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td></td>
<td>3 -0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td></td>
<td>4 0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td></td>
<td>5 0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td></td>
<td>6 0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td></td>
<td>7 -1.157892</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
</tbody>
</table>

In [26]: dfa['A'] = list(range(len(dfa.index))) # use this form to create a new column

In [27]: dfa

Out[27]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td></td>
<td>1 -0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td></td>
<td>2 -2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td></td>
<td>3 -0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td></td>
<td>4 0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td></td>
<td>5 0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td></td>
<td>6 0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td></td>
<td>7 -1.157892</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
</tbody>
</table>

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.
- Similarly, the attribute will not be available if it conflicts with any of the following list: index, major_axis.minor_axis.items.labels.
- 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:

In [28]: x = pd.DataFrame({'x': [1, 2, 3], 'y': [3, 4, 5]})

In [29]: x.iloc[1] = dict(x=9, y=99)

In [30]: x
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`:

```
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]:
<table>
<thead>
<tr>
<th>one</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>3.0</td>
</tr>
</tbody>
</table>
```

### 12.4 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:

```
In [31]: s[:5]
```

```
Out[31]:
| 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 [32]: s[::2]
```

```
| 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 [33]: s[::-1]
```

```
| 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 |
```
Note that setting works as well:

```python
In [34]: s2 = s.copy()
In [35]: s2[:5] = 0
In [36]: s2
```

```
Out[36]:
2000-01-01 0.000000
2000-01-02 0.000000
2000-01-03 0.000000
2000-01-04 0.000000
2000-01-06 -0.673690
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.

```python
In [37]: df[:3]
```

```
Out[37]:
     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
```

```python
In [38]: df[::-1]
```

```
     A         B         C         D
2000-01-08 -0.370690  0.113648 -1.478427  0.524988
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
```

### 12.5 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 [39]: df = pd.DataFrame(np.random.randn(5,4), columns=list('ABCD'), index=pd.
→date_range('20130101',periods=5))
In [40]: df.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 [41]: df.loc['20130102':'20130104']
Out[41]:
   A    B    C    D
2013-01-02 -0.674600 -1.776904 -0.968914
2013-01-03 -0.413738  0.276662 -0.472035
2013-01-04 -0.362543 -0.006154 -0.923061
```

Warning: Starting in 0.21.0, pandas will show a FutureWarning if indexing with a list with missing labels. In the future this will raise a KeyError. 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. All of the labels for which you ask, 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! - also See Slicing with labels)
- A boolean array
- A callable, see Selection By Callable

```
In [42]: s1 = pd.Series(np.random.randn(6),index=list('abcdef'))
In [43]: s1
Out[43]:
  a  1.431256
```

12.5. Selection By Label
In [44]: s1.loc['c':]

        c    -1.17030
       d    -0.22617
       e     0.41084
       f     0.81385
       dtype: float64

In [45]: s1.loc['b']

       b    1.34031

Note that setting works as well:

In [46]: s1.loc['c':] = 0

In [47]: s1

       a    1.43126
       b    1.34031
       c    0.00000
       d    0.00000
       e    0.00000
       f    0.00000
       dtype: float64

With a DataFrame

In [48]: df1 = pd.DataFrame(np.random.randn(6,4),
   index=list('abcdef'),
   columns=list('ABCD'))

In [49]: df1

          A         B         C         D
       a  0.132003  -0.827317  -0.076467  -1.18768
       b  1.130127  -1.436737  -1.413681   1.60792
       c  1.024180   0.569605   0.875906  -2.21137
       d  0.974466  -2.006747  -0.410001  -0.07864
       e  0.545952  -1.219217  -1.226825   0.76980
       f -1.281247  -0.727707  -0.121306  -0.09788

In [50]: df1.loc[['a', 'b', 'd'], :]

          A         B         C         D
       a  0.132003  -0.827317  -0.076467  -1.18768
       b  1.130127  -1.436737  -1.413681   1.60792
Accessing via label slices

```
In [51]: df1.loc['d':, 'A':'C']
Out[51]:
     A      B      C
    d  0.97447  2.00675 -0.410001
    e  0.54595  1.21921  1.226825
    f -1.28125  0.727707  0.121306
```

For getting a cross section using a label (equiv to df.xs('a'))

```
In [52]: df1.loc['a']
Out[52]:
     A      B      C
    a  0.132003  1.130127  1.024180
    b  0.97447  2.00675 -0.410001
    c  0.54595  1.21921  1.226825
    d  0.54595  1.21921  1.226825
    e -1.28125  0.727707  0.121306
Name: a, dtype: float64
```

For getting values with a boolean array

```
In [53]: df1.loc['a'] > 0
Out[53]:
     A
    a   True
    b  False
    c  False
    d  False
Name: a, dtype: bool
```

In [54]: df1.loc[:, df1.loc['a'] > 0]
```
Out[54]:
     A
    a  0.132003
    b  1.130127
    c  1.024180
    d  0.97447
    e  0.54595
    f -1.28125
```

For getting a value explicitly (equiv to deprecated df.get_value('a','A'))

```
# this is also equivalent to `df1.at['a','A']`
In [55]: df1.loc['a', 'A']
Out[55]:
```
      0
a  0.132003
```

### 12.5.1 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:

```
In [56]: s = pd.Series(list('abcde'), index=[0,3,2,5,4])
In [57]: s.loc[3:5]
Out[57]:
3   b
2   c
```

### 12.5. Selection By Label
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 [58]: s.sort_index()
Out[58]:
0 a
2 c
3 b
4 e
5 d
dtype: object
```

```python
In [59]: s.sort_index().loc[1:6]
Out[59]:
2 c
3 b
4 e
5 d
dtype: object
```

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`.

### 12.6 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 bounds 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 [60]: s1 = pd.Series(np.random.randn(5), index=list(range(0,10,2)))
In [61]: s1
Out[61]:
0  0.695775
2  0.341734
4  0.959726
```
Note that setting works as well:

```
In [64]: s1.iloc[:3] = 0
```
```
In [65]: s1
```
```
Out[65]:
\[\begin{array}{cccc}
0 & 0.000000 & 0.000000 & 0.000000 \\
2 & 0.000000 & 0.000000 & 0.000000 \\
4 & -1.110336 & -0.619976 & 0.000000 \\
6 & -1.110336 & -0.619976 & 0.000000 \\
8 & -0.619976 & 0.000000 & 0.000000 \\
\end{array}\]
dtype: float64
```

With a DataFrame

```
In [66]: df1 = pd.DataFrame(np.random.randn(6,4),
    index=list(range(0,12,2)),
    columns=list(range(0,8,2)))
```
```
In [67]: df1
```
```
Out[67]:
\[\begin{array}{cccc}
0 & 2 & 4 & 6 \\
0 & 0.149748 & -0.732339 & 0.687738 & 0.176444 \\
12 & 0.403310 & -0.154951 & 0.301624 & -2.179861 \\
4 & -1.369849 & -0.954208 & 1.462696 & -1.743161 \\
6 & -0.826591 & -0.345352 & 1.314271 & 0.690579 \\
8 & 0.995761 & 2.396780 & 0.014871 & 3.357427 \\
10 & -0.317441 & -1.236269 & 0.896171 & -0.487602 \\
\end{array}\]
```

Select via integer slicing

```
In [68]: df1.iloc[:3]
```
```
Out[68]:
\[\begin{array}{cccc}
0 & 2 & 4 & 6 \\
0 & 0.149748 & -0.732339 & 0.687738 & 0.176444 \\
2 & 0.403310 & -0.154951 & 0.301624 & -2.179861 \\
4 & -1.369849 & -0.954208 & 1.462696 & -1.743161 \\
6 & -0.826591 & -0.345352 & 1.314271 & 0.690579 \\
8 & 0.995761 & 2.396780 & 0.014871 & 3.357427 \\
10 & -0.317441 & -1.236269 & 0.896171 & -0.487602 \\
\end{array}\]
```
```
In [69]: df1.iloc[1:5, 2:4]
```
```
Out[69]:
\[\begin{array}{cccc}
0 & 2 & 4 & 6 \\
0 & 0.149748 & -0.732339 & 0.687738 & 0.176444 \\
1 & 0.403310 & -0.154951 & 0.301624 & -2.179861 \\
2 & -1.369849 & -0.954208 & 1.462696 & -1.743161 \\
4 & -0.826591 & -0.345352 & 1.314271 & 0.690579 \\
6 & 0.995761 & 2.396780 & 0.014871 & 3.357427 \\
8 & -0.317441 & -1.236269 & 0.896171 & -0.487602 \\
\end{array}\]
```
Select via integer list

```python
In [70]: df1.iloc[[1, 3, 5], [1, 3]]
Out[70]:
   2     4
0  0.301624 -2.179861
1  1.314232  0.690579
2 -0.345352  0.690579
3  0.014871  3.357427
4 -1.236269 -0.487602
```

```python
In [71]: df1.iloc[1:3, :]
Out[71]:
   0   2   4   6
0  0.403310 -0.154951  0.301624 -2.179861
1 -1.369849 -0.954208  1.462696 -1.743161
2  0.403310 -0.154951  0.301624 -2.179861
```

```python
In [72]: df1.iloc[:, 1:3]
Out[72]:
   2    4
0 -0.732339  0.687738
1 -0.154951  0.301624
2 -0.954208  1.462696
3 -0.345352  1.314232
4  2.396780  0.014871
5 -1.236269  0.896171
```

```python
# this is also equivalent to `df1.iat[1,1]`
In [73]: df1.iloc[1, 1]
Out[73]:
0 0.403310
2 -0.154951
4 0.301624
6 -2.179861
Name: 2, dtype: float64
```

For getting a cross section using an integer position (equiv to df.xs(1))

```python
In [74]: df1.iloc[1]
Out[74]:
   0   2   4   6
0  0.403310 -0.154951  0.301624 -2.179861
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
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/Numpy.

```python
# these are allowed in python/numpy.
In [75]: x = list('abcdef')
In [76]: x
Out[76]: ['a', 'b', 'c', 'd', 'e', 'f']
In [77]: x[4:10]
```

```python
In [78]: x[8:10]
```
In [79]: s = pd.Series(x)

In [80]: s
Out[80]:
0   a
1   b
2   c
3   d
4   e
5   f
dtype: object

In [81]: s.iloc[4:10]
Out[81]:
4   e
5   f
dtype: object

In [82]: s.iloc[8:10]
→ 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 [83]: dfl = pd.DataFrame(np.random.randn(5,2), columns=list('AB'))

In [84]: dfl
Out[84]:
     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 [85]: dfl.iloc[:, 2:3]
→ Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [86]: dfl.iloc[:, 1:3]
→
     A   B
0  -2.182937
1   0.084844
2   1.519970
3   0.600178
4   0.132885

In [87]: dfl.iloc[4:6]
→
     A   B

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

dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

## 12.7 Selection By Callable

New in version 0.18.1.

`.loc`, `.iloc`, and also `[]` indexing can accept a `callable` as indexer. The `callable` must be a function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing.

```python
In [88]: df1 = pd.DataFrame(np.random.randn(6, 4),
                      index=list('abcdef'),
                      columns=list('ABCD'))

In [89]: df1
Out[89]:
    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

In [90]: df1.loc[lambda df: df.A > 0, :]
Out[90]:
      A      B      C      D
  c  0.299368 -0.863838  0.408204 -1.048089
e  1.289997  0.082423 -0.055758  0.536580

In [91]: df1.iloc[:, lambda df: ['A', 'B']]
Out[91]:
     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 [92]: df1.iloc[:, lambda df: [0, 1]]
Out[92]:
     A      B
a -0.023688  2.410179
```
In [93]: df1[lambda df: df.columns[0]]

→

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 [94]: df1.A.loc[lambda s: s > 0]

Out[94]:
c  0.299368
e  1.289997
Name: A, dtype: float64

Using these methods / indexers, you can chain data selection operations without using temporary variable.

In [95]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [96]: (bb.groupby(['year', 'team']).sum()  
    ....: .loc[lambda df: df.r > 100])

Out[96]:

<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>
</tr>
</thead>
<tbody>
<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></td>
<td>2</td>
<td>36</td>
<td>125.0</td>
<td>10.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>DET</td>
<td>5</td>
<td>301</td>
<td>1062</td>
<td>162</td>
<td>283</td>
<td>54</td>
<td></td>
<td>4</td>
<td>37</td>
<td>144.0</td>
<td>24.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>HOU</td>
<td>4</td>
<td>311</td>
<td>926</td>
<td>109</td>
<td>218</td>
<td>47</td>
<td></td>
<td>6</td>
<td>14</td>
<td>77.0</td>
<td>10.0</td>
<td>4.0</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td></td>
<td>NYN</td>
<td>13</td>
<td>622</td>
<td>1854</td>
<td>240</td>
<td>509</td>
<td>101</td>
<td>3</td>
<td>61</td>
<td>243.0</td>
<td>22.0</td>
<td>4.0</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>SFN</td>
<td>5</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>
</tr>
<tr>
<td></td>
<td>TEX</td>
<td>2</td>
<td>198</td>
<td>729</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>
</tr>
<tr>
<td></td>
<td>TOR</td>
<td>4</td>
<td>459</td>
<td>1408</td>
<td>187</td>
<td>378</td>
<td>96</td>
<td>2</td>
<td>58</td>
<td>223.0</td>
<td>4.0</td>
<td>2.0</td>
<td>190</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>year</th>
<th>team</th>
<th>so</th>
<th>ibb</th>
<th>hbp</th>
<th>sh</th>
<th>sf</th>
<th>gldp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>CIN</td>
<td>127.0</td>
<td>14.0</td>
<td>1.0</td>
<td>1.0</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>DET</td>
<td>176.0</td>
<td>3.0</td>
<td>10.0</td>
<td>4.0</td>
<td>8.0</td>
<td>28.0</td>
</tr>
<tr>
<td></td>
<td>HOU</td>
<td>212.0</td>
<td>3.0</td>
<td>9.0</td>
<td>16.0</td>
<td>6.0</td>
<td>17.0</td>
</tr>
<tr>
<td></td>
<td>LAN</td>
<td>141.0</td>
<td>8.0</td>
<td>9.0</td>
<td>3.0</td>
<td>8.0</td>
<td>29.0</td>
</tr>
<tr>
<td></td>
<td>NYN</td>
<td>310.0</td>
<td>24.0</td>
<td>23.0</td>
<td>18.0</td>
<td>15.0</td>
<td>48.0</td>
</tr>
<tr>
<td></td>
<td>SFN</td>
<td>188.0</td>
<td>51.0</td>
<td>8.0</td>
<td>16.0</td>
<td>6.0</td>
<td>41.0</td>
</tr>
<tr>
<td></td>
<td>TEX</td>
<td>140.0</td>
<td>4.0</td>
<td>5.0</td>
<td>2.0</td>
<td>8.0</td>
<td>16.0</td>
</tr>
<tr>
<td></td>
<td>TOR</td>
<td>265.0</td>
<td>16.0</td>
<td>12.0</td>
<td>4.0</td>
<td>16.0</td>
<td>38.0</td>
</tr>
</tbody>
</table>
12.8 IX Indexer is Deprecated

**Warning:** Starting in 0.20.0, 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. To wit, `.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 recommended methods of indexing are:

- `.loc` if you want to **label** index
- `.iloc` if you want to **positionally** index.

```python
In [97]: dfd = pd.DataFrame({'A': [1, 2, 3],
                         'B': [4, 5, 6]},
                        index=list('abc'))

In [98]: dfd
Out[98]:
   A  B
a 1  4
b 2  5
c 3  6
```

Previous Behavior, where you wish to get the 0th and the 2nd elements from the index in the ‘A’ column.

```python
In [3]: dfd.ix[[0, 2], 'A']
Out[3]:
a 1  
c 3  
Name: A, dtype: int64
```

Using `.loc`. Here we will select the appropriate indexes from the index, then use **label** indexing.

```python
In [99]: dfd.loc[dfd.index[[0, 2]], 'A']
Out[99]:
a 1  
c 3  
Name: A, dtype: int64
```

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 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
```
12.9 Indexing with list with missing labels is Deprecated

**Warning:** Starting in 0.21.0, using `.loc` or `[]` with a list with one or more missing labels, is deprecated, 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 is deprecated and will show a warning message pointing to this section. The recommended alternative is to use `.reindex()`.

For example:

```
In [102]: s = pd.Series([1, 2, 3])

In [103]: s
Out[103]:
0  1
1  2
2  3
dtype: int64
```

Selection with all keys found is unchanged.

```
In [104]: s.loc[[1, 2]]
Out[104]:
1  2
2  3
dtype: int64
```

Previous Behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1  2.0
2  3.0
3  NaN
dtype: float64
```

Current Behavior

```
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:

```
Out[4]:
1  2.0
2  3.0
3  NaN
dtype: float64
```
12.9.1 Reindexing

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`. See also the section on reindexing.

```python
In [105]: s.reindex([1, 2, 3])
Out[105]:
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.

```python
In [106]: labels = [1, 2, 3]
In [107]: s.loc[s.index.intersection(labels)]
Out[107]:
1   2
2   3
dtype: int64
```

Having a duplicated index will raise for a `.reindex()`:

```python
In [108]: s = pd.Series(np.arange(4), index=['a', 'a', 'b', 'c'])
In [109]: labels = ['c', 'd']
```

```python
In [17]: s.reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

Generally, you can intersect the desired labels with the current axis, and then reindex.

```python
In [110]: s.loc[s.index.intersection(labels)].reindex(labels)
Out[110]:
c   3.0
d   NaN
dtype: float64
```

However, this would still raise if your resulting index is duplicated.

```python
In [41]: labels = ['a', 'd']
In [42]: s.loc[s.index.intersection(labels)].reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

12.10 Selecting Random Samples

A random selection of rows or columns from a Series, DataFrame, or Panel 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.

```python
In [111]: s = pd.Series([0,1,2,3,4,5])
```

# When no arguments are passed, returns 1 row.
# One may specify either a number of rows:

```python
In [113]: s.sample(n=3)
```

```
Out[113]:
0 0
4 4
1 1
dtype: int64
```

# Or a fraction of the rows:

```python
In [114]: s.sample(frac=0.5)
```

```
Out[114]:
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:

```python
In [115]: s = pd.Series([0,1,2,3,4,5])

# Without replacement (default):
In [116]: s.sample(n=6, replace=False)
```

```
Out[116]:
0 0
1 1
5 5
3 3
2 2
4 4
dtype: int64
```

# With replacement:

```python
In [117]: s.sample(n=6, replace=True)
```

```
Out[117]:
0 0
4 4
3 3
2 2
4 4
4 4
dtype: int64
```

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:

```python
In [118]: s = pd.Series([0,1,2,3,4,5])

In [119]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]
```

# 12.10. Selecting Random Samples
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.

sample also allows users to sample columns instead of rows using the axis argument.

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.
12.11 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

```python
In [130]: se = pd.Series([1,2,3])

In [131]: se
Out[131]:
0   1
1   2
2   3
dtype: int64


In [133]: se
Out[133]:
0   1.0
1   2.0
2   3.0
5   5.0
dtype: float64
```

A `DataFrame` can be enlarged on either axis via `.loc`

```python
In [134]: dfi = pd.DataFrame(np.arange(6).reshape(3,2),
                    columns=['A','B'])

In [135]: dfi
Out[135]:
   A  B
0  0  1
1  2  3
2  4  5

In [136]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [137]: dfi
Out[137]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an `append` operation on the `DataFrame`.

```python
In [138]: dfi.loc[3] = 5

In [139]: dfi
Out[139]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```
12.12 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 [140]: s.iat[5]
Out[140]: 5

In [141]: df.at[dates[5], 'A']
\n\nOut[141]: -0.67368970808837059

In [142]: df.iat[3, 0]
\n\nOut[142]: 0.72155516224436689
```

You can also set using these same indexers.

```
In [143]: df.at[dates[5], 'E'] = 7
In [144]: df.iat[3, 0] = 7
```

at may enlarge the object in-place as above if the indexer is missing.

```
In [145]: df.at[dates[-1]+1, 0] = 7

In [146]: df
Out[146]:
   A         B         C         D         E  
0 2000-01-01 0.469112 -1.509059 -1.135632  NaN  NaN
1 2000-01-02 1.212112 -0.173215  0.119209 -1.044236  NaN  NaN
2 2000-01-03 -0.861849 -2.104569 -0.494929  1.071804  NaN  NaN
3 2000-01-04  7.000000 -0.706771 -1.039575  0.271860  NaN  NaN
4 2000-01-05 -0.424972  0.567020  0.276232 -1.087401  NaN  NaN
5 2000-01-06 -0.673690  0.113648 -1.478427  0.524988   7.0  NaN
6 2000-01-07  0.404705  0.577046 -1.715002 -1.039268  NaN  NaN
7 2000-01-08 -0.370647 -1.157892 -1.344312  0.844885  NaN  NaN
8 2000-01-09  NaN   NaN   NaN   NaN   NaN   7.0
```

12.13 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.

Using a boolean vector to index a Series works exactly as in a numpy ndarray:

```
In [147]: s = pd.Series(range(-3, 4))

In [148]: s
Out[148]:
   0 -3
   1 -2
   2 -1
```
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):

```python
In [152]: df[df['A'] > 0]
Out[152]:
      A        B        C        D       E
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  2000-01-04  7.000000 -0.706771 -1.039575  0.271860
3  2000-01-07  0.404705  0.577046 -1.715002 -1.039268
```

List comprehensions and map method of Series can also be used to produce more complex criteria:

```python
In [153]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                        'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                        'c': np.random.randn(7)})

In [154]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [155]: df2[criterion]
Out[155]:
   a  b   c
0  two y 0.041290
```

---

12.13. Boolean indexing
three x 0.361719
two y -0.238075

# equivalent but slower
In [156]: df2[[x.startswith('t') for x in df2['a']]]

\→
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.041290</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.361719</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-0.238075</td>
<td></td>
</tr>
</tbody>
</table>

# Multiple criteria
In [157]: df2[criterion & (df2['b'] == 'x')]

\→
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>x</td>
<td>0.361719</td>
</tr>
</tbody>
</table>

Note, 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 [158]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
Out[158]:
\→
<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 x</td>
<td>0.361719</td>
</tr>
</tbody>
</table>

12.14 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 [159]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')

In [160]: s
Out[160]:
\→
<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

dtype: int64

In [161]: s.isin([2, 4, 6])
\→
<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

dtype: bool

In [162]: s[s.isin([2, 4, 6])]
\→
<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 [163]: s[s.index.isin([2, 4, 6])]
Out[163]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
dtype: int64

# compare it to the following
In [164]: s.reindex([2, 4, 6])
Out[164]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>NaN</td>
</tr>
</tbody>
</table>
dtype: float64

In addition to that, MultiIndex allows selecting a separate level to use in the membership check:

In [165]: s_mi = pd.Series(np.arange(6),
  index=pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c']]))

In [166]: s_mi
Out[166]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a 0</td>
</tr>
<tr>
<td></td>
<td>b 1</td>
</tr>
<tr>
<td></td>
<td>c 2</td>
</tr>
<tr>
<td>1</td>
<td>a 3</td>
</tr>
<tr>
<td></td>
<td>b 4</td>
</tr>
<tr>
<td></td>
<td>c 5</td>
</tr>
</tbody>
</table>
dtype: int64

In [167]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
Out[167]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c 2</td>
</tr>
<tr>
<td>1</td>
<td>a 3</td>
</tr>
</tbody>
</table>
dtype: int64

In [168]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
Out[168]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a 0</td>
</tr>
<tr>
<td></td>
<td>b 2</td>
</tr>
<tr>
<td></td>
<td>c 5</td>
</tr>
</tbody>
</table>
dtype: int64

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.
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.

```
In [172]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [173]: df.isin(values)
Out[173]:
   ids  ids2  vals
0  True  False  True
1  True  False  False
2  False  False  True
3  False  False  False
```

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:

```
In [174]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}

In [175]: row_mask = df.isin(values).all(1)

In [176]: df[row_mask]
Out[176]:
   ids  ids2  vals
0   a   a     1
```

### 12.15 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

```
In [177]: s[s > 0]
Out[177]:
        3
2  2
1  3
0  4
dtype: int64
```

To return a Series of the same shape as the original
In [178]: s.where(s > 0)
Out[178]:
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. Equivalent is `df.where(df < 0)`

In [179]: df[df < 0]
Out[179]:
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 -1.233203
2000-01-05  NaN -0.605656 -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.

In [180]: df.where(df < 0, -df)
Out[180]:
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 -0.048788 -0.808838
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:

In [181]: s2 = s.copy()
In [182]: s2[s2 < 0] = 0
In [183]: s2
Out[183]:
4  0
3  1
2  2
1  3
0  4
dtype: int64
In [184]: df2 = df.copy()
In [185]: df2[df2 < 0] = 0
In [186]: df2
Out[186]:
+----------------+----------------+----------------+----------------+
<p>| | | | |
|                |                |                |                |</p>
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.485855</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.000000</td>
<td>0.390389</td>
<td>0.000000</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.000000</td>
<td>0.299674</td>
<td>0.000000</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.846958</td>
<td>0.000000</td>
<td>0.600705</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.669692</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.868584</td>
<td>0.000000</td>
<td>2.297780</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.168904</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.801196</td>
<td>1.392071</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
+----------------+----------------+----------------+----------------+

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:

In [187]: df_orig = df.copy()
In [188]: df_orig.where(df > 0, -df, inplace=True);
In [189]: df_orig
Out[189]:
+----------------+----------------+----------------+----------------+
<p>| | | | |
|                |                |                |                |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.352480</td>
<td>0.390389</td>
<td>1.192319</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.864883</td>
<td>0.299674</td>
<td>0.227870</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.846958</td>
<td>1.122082</td>
<td>0.600705</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.669692</td>
<td>0.605656</td>
<td>1.169184</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.868584</td>
<td>0.948458</td>
<td>2.297780</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>2.670153</td>
<td>0.114722</td>
<td>0.168904</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.801196</td>
<td>1.392071</td>
<td>0.048788</td>
</tr>
</tbody>
</table>
+----------------+----------------+----------------+----------------+

Note: The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2).

In [190]: df.where(df < 0, -df) == np.where(df < 0, df, -df)

Out[190]:
+----------------+----------------+----------------+----------------+
<p>| | | | |
|                |                |                |                |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>2000-01-02</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>
</tr>
<tr>
<td>2000-01-04</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>
</tr>
<tr>
<td>2000-01-06</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>
</tr>
<tr>
<td>2000-01-08</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 [191]: df2 = df.copy()
In [192]: df2[df2[1:4] > 0 ] = 3
In [193]: df2

636 Chapter 12. Indexing and Selecting Data
Where can also accept `axis` and `level` parameters to align the input when performing the `where`.

```
In [194]: df2 = df.copy()

In [195]: df2.where(df2>0,df2['A'],axis='index')
```

```
Out[195]:
       A    B    C    D
2000-01-01 -2.104139 -2.104139 0.485855 0.245166
2000-01-02 -0.352480 0.390389 -0.352480 1.655824
2000-01-03 -0.864883 0.299674 -0.864883 0.281059
2000-01-04 0.846958 0.669692 0.669692 0.342416
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 (but faster than) the following.

```
In [196]: df2 = df.copy()

In [197]: df.apply(lambda x, y: x.where(x>0,y), y=df['A'])
```

```
Out[197]:
       A    B    C    D
2000-01-01 -2.104139 -2.104139 0.485855 0.245166
2000-01-02 -0.352480 0.390389 -0.352480 1.655824
2000-01-03 -0.864883 0.299674 -0.864883 0.281059
2000-01-04 0.846958 0.669692 0.669692 0.342416
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
```

New in version 0.18.1.

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 [198]: df3 = pd.DataFrame({'A': [1, 2, 3],
                      'B': [4, 5, 6],
                      'C': [7, 8, 9]})

In [199]: df3.where(lambda x: x > 4, lambda x: x + 10)
```

```
Out[199]:
     A   B    C
0  11  14  7
1  12  5  8
```

12.15. The `where()` Method and Masking
mask

mask is the inverse boolean operation of where.

```
In [201]: s.mask(s >= 0)
Out[201]:
   0 1 2 3
--- --- --- ---
  NaN NaN NaN NaN
dtype: float64
```

```
In [201]: df.mask(df >= 0)
```

### 12.16 The `query()` Method (Experimental)

`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 [202]: n = 10

In [203]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [204]: df
Out[204]:
   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 [205]: df[(df.a < df.b) & (df.b < df.c)]
```
Do the same thing but fall back on a named index if there is no column with the name a.

```python
In [207]: df = pd.DataFrame(np.random.randint(n / 2, size=(n, 2)), columns=list('bc'))
In [208]: df.index.name = 'a'
In [209]: df
Out[209]:
   b  c
a
0  0  4
1  1  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
In [210]: df.query('a < b and b < c')
```

If instead you don’t want to or cannot name your index, you can use the name index in your query expression:

```python
In [211]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)), columns=list('bc'))
In [212]: df
Out[212]:
   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
```
Note: If the name of your index overlaps with a column name, the column name is given precedence. For example,

```python
In [214]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})
In [215]: df.index.name = 'a'
In [216]: df.query('a > 2')  # uses the column 'a', not the index
Out[216]:
   a
0  3
1  3
2  3
3  3
```

You can still use the index in a query expression by using the special identifier ‘index’:

```python
In [217]: df.query('index > 2')
Out[217]:
   a
0  3
1  3
2  4
3  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.

### 12.16.1 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 [218]: n = 10
In [219]: colors = np.random.choice(['red', 'green'], size=n)
In [220]: foods = np.random.choice(['eggs', 'ham'], size=n)
In [221]: colors
Out[221]:
array(['red', 'red', 'red', 'green', 'green', 'green', 'green',
      'green', 'green', 'green'],
       dtype='<U5')
In [222]: foods
Out[222]:
array(['ham', 'ham', 'eggs', 'eggs', 'eggs', 'ham', 'ham', 'eggs',
      'eggs', 'eggs'],
       dtype='<U5')
```
In [223]: index = pd.MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
In [224]: df = pd.DataFrame(np.random.randn(n, 2), index=index)
In [225]: df
Out[225]:
   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
green eggs -2.029766  0.792652
     ham  0.461007 -0.542749
     eggs -0.305384 -0.479195
green eggs -0.707140 -0.773882
green eggs  0.229453  0.304418

In [226]: df.query('color == "red"')

If the levels of the MultiIndex are unnamed, you can refer to them using special names:

In [227]: df.index.names = [None, None]
In [228]: df
Out[228]:
   0  1
red  ham  0.194889 -0.381994
     ham  0.318587  2.089075
     eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
green eggs -2.029766  0.792652
     ham  0.461007 -0.542749
     eggs -0.305384 -0.479195
green eggs -0.707140 -0.773882
green eggs  0.229453  0.304418

In [229]: df.query('ilevel_0 == "red"')

The convention is ilevel_0, which means “index level 0” for the 0th level of the index.

12.16. The query() Method (Experimental)
A use case for `query()` is when you have a collection of `DataFrame` objects that have a subset of column names (or index levels/labels) in common. You can pass the same query to both frames without having to specify which frame you’re interested in querying.

```python
In [230]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [231]: df
Out[231]:
   a          b          c
0  0.224283  0.736107  0.139168
1  0.302827  0.657803  0.713897
2  0.611850  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 [232]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)

In [233]: df2
Out[233]:
   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 [234]: expr = '0.0 <= a <= c <= 0.5'

In [235]: map(lambda frame: frame.query(expr), [df, df2])
Out[235]: <map at 0x12cb5cc18>
```

### 12.16.3 query () Python versus pandas Syntax Comparison

Full numpy-like syntax

```python
In [236]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=list('abc'))

In [237]: df
Out[237]:
   a  b  c
0  7  8  9
1  1  0  7
2  2  7  2
```
In [238]: df.query('(a < b) & (b < c)')
\→
  a  b  c
0  7  8  9

In [239]: df[(df.a < df.b) & (df.b < df.c)]
\→
  a  b  c
0  7  8  9

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than \&|)

In [240]: df.query('a < b & b < c')
Out[240]:
  a  b  c
0  7  8  9

Use English instead of symbols

In [241]: df.query('a < b and b < c')
Out[241]:
  a  b  c
0  7  8  9

Pretty close to how you might write it on paper

In [242]: df.query('a < b < c')
Out[242]:
  a  b  c
0  7  8  9

12.16.4 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`.

# get all rows where columns "a" and "b" have overlapping values
In [243]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aabbccddeeff'),
                      'c': np.random.randint(5, size=12),
                      'd': np.random.randint(9, size=12)})

In [244]: df
Out[244]:
   a  b  c  d
0  a  a  2   6
In [245]: df.query('a in b')

    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

# How you’d do it in pure Python
In [246]: df[df.a.isin(df.b)]

    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 [247]: df.query('a not in b')

    a  b  c  d
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 [248]: df[~df.a.isin(df.b)]

    a  b  c  d
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
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 [249]: df.query('a in b and c < d')
```

```text
Out[249]:
   a  b  c  d
 0  a  a  2  6
 1  a  a  4  7
 2  b  a  1  6
 3  c  b  3  6
 4  c  b  0  2
```

```python
# pure Python
In [250]: df[df.b.isin(df.a) & (df.c < df.d)]
```

```text
Out[250]:
   a  b  c  d
 0  a  a  2  6
 1  a  a  4  7
 2  b  a  1  6
 3  c  b  3  6
 4  c  b  0  2
 5  f  c  0  6
 6  f  c  1  2
```

**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.

### 12.16.5 Special use of the `==` operator with list objects

Comparing a list of values to a column using `==/!=` works similarly to `in/not in`

```python
In [251]: df.query('b == ["a", "b", "c"]')
```

```text
Out[251]:
   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 [252]: df[df.b.isin(['a', 'b', 'c'])]

```
  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 [253]: df.query('c == [1, 2]')

```
  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 [254]: df.query('c != [1, 2]')

```
  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
```

# using in/not in

### In [255]: df.query('1, 2 in c')

```
  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 [256]: df.query('1, 2 not in c')

```
  a  b  c  d
1  a  a  4  7
4  c  b  3  6
5  c  b  0  2
```

---

Chapter 12. Indexing and Selecting Data
12.16.6 Boolean Operators

You can negate boolean expressions with the word not or the ~ operator.

Of course, expressions can be arbitrarily complex too

# short query syntax
In [263]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [264]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]

In [265]: shorter
Out[265]:
   a    b    c  bools
 0  7.0  0.275  0.691 False
In [266]: longer
   a    b    c  bools
 0  7.0  0.275  0.691 False
In [267]: shorter == longer
   a    b    c  bools
 0  True  True  True  True

12.16.7 Performance of `query()`

`DataFrame.query()` using `numexpr` is slightly faster than Python for large frames.

\[\text{Dataframe.query()}
\]

\[\text{python}
\]
\[\text{numexpr}
\]

\[\text{Note: You will only see the performance benefits of using the numexpr engine with DataFrame.query() if your frame has more than approximately 200,000 rows}\]
This plot was created using a DataFrame with 3 columns each containing floating point values generated using numpy.random.randn().

### 12.17 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 [268]: 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 [269]: df2
Out[269]:
   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
```
In [270]: df2.duplicated('a')

˓→
0   False
1   True
2   False
3   True
4   True
5   False
6   False
dtype: bool

In [271]: df2.duplicated('a', keep='last')

˓→
0   True
1   False
2   True
3   True
4   False
5   False
6   False
dtype: bool

In [272]: df2.duplicated('a', keep=False)

˓→
0   True
1   True
2   True
3   True
4   True
5   False
6   False
dtype: bool

In [273]: df2.drop_duplicates('a')

˓→
   a  b  c
0  one x -1.067137
2  two x -0.211056
5  three x -1.964475
6  four x  1.298329

In [274]: df2.drop_duplicates('a', keep='last')

˓→
   a  b  c
1  one y  0.309500
4  two x -0.390820
5  three x -1.964475
6  four x  1.298329

In [275]: df2.drop_duplicates('a', keep=False)

˓→
Also, you can pass a list of columns to identify duplications.

```python
In [276]: df2.duplicated(['a', 'b'])
Out[276]:
   0  False
   1  False
   2  False
   3  False
   4   True
   5  False
   6  False
dtype: bool
```

```python
In [277]: df2.drop_duplicates(['a', 'b'])
```

To drop duplicates by index value, use `Index.duplicated` then perform slicing. Same options are available in `keep` parameter.

```python
In [278]: df3 = pd.DataFrame({'a': np.arange(6),
       'b': np.random.randn(6),
       'c': np.random.randn(6)},
       index=['a', 'a', 'b', 'c', 'b', 'a'])
In [279]: df3
Out[279]:
   a  b
a  0  1.440455
a  1  2.456086
b  2  1.038402
c  3  0.683536
b  4  0.894409
a  5  3.082764
```

```python
In [280]: df3.index.duplicated()
Out[280]:
array([False, True, False, False, True, True], dtype=bool)
```

```python
In [281]: df3[~df3.index.duplicated()]
```

12.17. Duplicate Data
12.18 Dictionary-like `get()` method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.

### Example

```python
In [284]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [285]: s.get('a')  # equivalent to s['a']
Out[285]: 1

In [286]: s.get('x', default=-1)
```

12.19 The `lookup()` Method

Sometimes you want to extract a set of values given a sequence of row labels and column labels, and the `lookup` method allows for this and returns a numpy array. For instance,

### Example

```python
In [287]: dflookup = pd.DataFrame(np.random.rand(20, 4), columns=['A', 'B', 'C', 'D'])

In [288]: dflookup.lookup(list(range(0, 10, 2)), ['B', 'C', 'A', 'B', 'D'])
```

12.20 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 [289]: index = pd.Index(['e', 'd', 'a', 'b'])

In [290]: index
Out[290]: Index(['e', 'd', 'a', 'b'], dtype='object')
```
In [291]: 'd' in index

Out[291]: True

You can also pass a name to be stored in the index:

In [292]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')

In [293]: index.name
Out[293]: 'something'

The name, if set, will be shown in the console display:

In [294]: index = pd.Index(list(range(5)), name='rows')

In [295]: columns = pd.Index(['A', 'B', 'C'], name='cols')

In [296]: df = pd.DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [297]: df
Out[297]:
   A   B   C
rows
0  1.295989  0.185778  0.436259
1  0.678101  0.311369 -0.528378
2 -0.674808 -1.103529 -0.656157
3  1.889957  2.076651 -1.102192
4 -1.211795 -0.791746  0.634724

In [298]: df['A']

Out[298]:
   rows
0  1.295989
1  0.678101
2 -0.674808
3  1.889957
4 -1.211795
Name: A, dtype: float64

12.20.1 Setting metadata

Indexes are “mostly immutable”, but it is possible to set and change their metadata, like the index name (or, for MultiIndex, levels and labels).

You can use the rename, set_names, set_levels, and set_labels to set these attributes directly. They default to returning a copy; however, you can specify inplace=True to have the data change in place.

See Advanced Indexing for usage of MultiIndexes.

In [299]: ind = pd.Index([1, 2, 3])

In [300]: ind.rename("apple")
Out[300]: Int64Index([1, 2, 3], dtype='int64', name='apple')

In [301]: ind
Out[301]: Int64Index([1, 2, 3], dtype='int64')
In [302]: ind.set_names(["apple"], inplace=True)
In [303]: ind.name = "bob"
In [304]: ind
Out[304]: Int64Index([1, 2, 3], dtype='int64', name='bob')

set_names, set_levels, and set_labels also take an optional level argument
In [305]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first', 'second'])
In [306]: index
Out[306]: MultiIndex(levels=[[0, 1, 2], ['one', 'two']],
               labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
               names=['first', 'second'])
In [307]: index.levels[1]
Out[307]: Index(['one', 'two'], dtype='object', name='second')
In [308]: index.set_levels(["a", "b"], level=1)
Out[308]: MultiIndex(levels=[[0, 1, 2], ['a', 'b']],
               labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
               names=['first', 'second'])

12.20.2 Set operations on Index objects

The two main operations are union (|), intersection (&) These can be directly called as instance methods or used via overloaded operators. Difference is provided via the .difference() method.

In [309]: a = pd.Index(['c', 'b', 'a'])
In [310]: b = pd.Index(['c', 'e', 'd'])
In [311]: a | b
Out[311]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
In [312]: a & b
Out[312]: Index(['c', 'b', 'c', 'd', 'e'], dtype='object')
In [313]: a.difference(b)
Out[313]: Index(['a', 'b'], dtype='object')

Also available is the symmetric_difference (^) operation, which returns elements that appear in either idx1 or idx2 but not both. This is equivalent to the Index created by idx1.difference(idx2).union(idx2.difference(idx1)), with duplicates dropped.

In [314]: idx1 = pd.Index([1, 2, 3, 4])
In [315]: idx2 = pd.Index([2, 3, 4, 5])
In [316]: idx1.symmetric_difference(idx2)
Out[316]: Int64Index([1, 5], dtype='int64')
In [317]: idx1 ^ idx2
Out[317]: Int64Index([1, 5], dtype='int64')

Note: The resulting index from a set operation will be sorted in ascending order.

12.20.3 Missing values

New in version 0.17.1.

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 [318]: idx1 = pd.Index([1, np.nan, 3, 4])
In [319]: idx1
Out[319]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')
In [320]: idx1.fillna(2)
Out[320]: Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')
In [322]: idx2
Out[322]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'], dtype='datetime64[ns], freq=None')
In [323]: idx2.fillna(pd.Timestamp('2011-01-02'))
Out[323]: DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], dtype='datetime64[ns], freq=None')

12.21 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.
12.21.1 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, indexed DataFrame:

```python
In [324]: data
Out[324]:
   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 [325]: indexed1 = data.set_index('c')
In [326]: indexed1
Out[326]:
   c    a    b    d
   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 [327]: indexed2 = data.set_index(['a', 'b'])
In [328]: indexed2
Out[328]:
   c    d
   a    b
   z  bar one  z  1.0
       two y  2.0
       x  foo one  x  3.0
       two w  4.0
In [329]: frame = data.set_index('c', drop=False)
In [330]: frame = frame.set_index(['a', 'b'], append=True)
In [331]: frame
Out[331]:
   c    d
   a    b
   z  bar one  z  1.0
       two y  2.0
       x  foo one  x  3.0
       two w  4.0
In [332]: data.set_index('c', drop=False)
Out[332]:
   a   b   c   d
   c
   z  bar one  z  1.0
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```python
In [329]: frame = data.set_index('c', drop=False)
In [330]: frame = frame.set_index(['a', 'b'], append=True)
In [331]: frame
Out[331]:
   c    d
   a    b
   z  bar one  z  1.0
       two y  2.0
       x  foo one  x  3.0
       two w  4.0
```

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 [332]: data.set_index('c', drop=False)
Out[332]:
   a   b   c   d
   c
   z  bar one  z  1.0
```
12.21.2 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 to `set_index`.

You can use the `level` keyword to remove only a portion of the index:
reset_index takes an optional parameter drop which if true simply discards the index, instead of putting index values in the DataFrame’s columns.

Note: The reset_index method used to be called delevel which is now deprecated.

12.21.3 Adding an ad hoc index

If you create an index yourself, you can just assign it to the index field:

data.index = index

12.22 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 [339]: dfmi = pd.DataFrame([list('abcd'),
                      ....: list('efgh'),
                      ....: list('ijkl'),
                      ....: list('mnop')],
                      ....: columns=pd.MultiIndex.from_product([['one','two'],
                      ....: ['first','second']))

In [340]: dfmi
Out[340]:
   one  two
first    first    second
  0    a    b    c    d
  1    e    f    g    h
  2    i    j    k    l
  3    m    n    o    p

In [341]: dfmi['one']['second']
Out[341]:
  0    b
  1    f
  2    j
  3    n
Name: second, dtype: object
```
In [342]: dfmi.loc[:,('one','second')]
Out[342]:
   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' happens. 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.

12.22.1 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 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:

def do_something(df):
    foo = df[['bar', 'baz']]  # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...

12.22. Returning a view versus a copy
Yikes!

### 12.22.2 Evaluation order matters

Furthermore, in chained expressions, the order may determine whether a copy is returned or not. If an expression will set values on a copy of a slice, then a SettingWithCopy warning will be issued.

You can control the action of a chained assignment via the option `mode.chained_assignment`, which can take the values `['raise', 'warn', None]`, where showing a warning is the default.

```python
In [343]: 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 [344]: 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`

This is the correct access method

```python
In [345]: dfc = pd.DataFrame({'A': ['aaa','bbb','ccc'],'B':[1,2,3]})

In [346]: dfc.loc[0,'A'] = 11

In [347]: dfc
Out[347]:
   A  B
0  11 1
1  bbb 2
2  ccc 3
```

This *can* work at times, but is not guaranteed, and so should be avoided

```python
In [348]: dfc = dfc.copy()

In [349]: dfc['A'][0] = 111
```
This will **not** work at all, and so should be avoided

```python
>>> pd.set_option('mode.chained_assignment','raise')
>>> dfc.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.
This section covers indexing with a MultiIndex and more 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

### 13.1 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

### 13.1.1 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), or a crossed set of iterables (using MultiIndex.from_product). The Index constructor will attempt to return a MultiIndex when it is passed a list of tuples. The following examples demo different ways to initialize MultiIndexes.

In [1]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], ...
   One: ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']] ...
   ...

In [2]: tuples = list(zip(*arrays))

In [3]: tuples
Out[3]:
[['bar', 'one'],...
In 

```python
index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

In 

```python
index
```

Out

```python
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
names=['first', 'second'])
```

In 

```python
s = pd.Series(np.random.randn(8), index=index)
```

In 

```python
s
```

Out

```python
first  second
bar one  0.469112
        two -0.282863
baz one  -1.509059
        two -1.135632
foo one  1.212112
        two -0.173215
qux one  0.119209
        two -1.044236
dtype: float64
```

When you want every pairing of the elements in two iterables, it can be easier to use the `MultiIndex.from_product` function:

In 

```python
iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]
```

In 

```python
pd.MultiIndex.from_product(iterables, names=['first', 'second'])
```

Out

```python
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
names=['first', 'second'])
```

As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

In 

```python
arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
          np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
```

In 

```python
s = pd.Series(np.random.randn(8), index=arrays)
```

In 

```python
s
```

Out

```python
bar one  -0.861849
two  -2.104569
baz one  -0.494929
two  1.071804
foo one  0.721555
two  -1.044236
dtype: float64
```
In [13]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)

In [14]: df
Out[14]:
        0     1     2     3
bar  one -0.424972  0.567020  0.276232 -1.087401
two  0.673690  0.113648 -1.478427  0.524988
baz  one  0.404705  0.577046 -1.715002 -1.039268
two  0.370647 -1.157892  0.844885  0.113648
foo  one  1.075770 -0.109050  1.643563 -1.469388
two  0.357021 -0.674600 -1.776994 -0.968914
qux  one -1.294524  0.413738  0.276662 -0.472035
two  0.013960 -0.362543 -0.061540 -0.923061

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 [15]: df.index.names
Out[15]: 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 [16]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'],
                      columns=index)

In [17]: df
Out[17]:
       first bar  baz  foo  qux
second one  two  one  two  one  two
A  0.895717 0.805244 -1.206412 2.565646 1.431256 1.340309 -1.170299
B  0.410835 0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127
C -1.413681 1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466

We’ve “sparsified” the higher levels of the indexes to make the console output a bit easier on the eyes.

13.1. Hierarchical indexing (MultiIndex) 665
It’s worth keeping in mind that there’s nothing preventing you from using tuples as atomic labels on an axis:

```python
In [19]: pd.Series(np.random.randn(8), index=tuples)
Out[19]:
(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.

Note that how the index is displayed be controlled using the `multi_sparse` option in `pandas.set_options()`:

```python
In [20]: pd.set_option('display.multi_sparse', False)
In [21]: df
Out[21]:
   first   bar      bar      baz      baz      foo      foo      qux
  second  one    two    one    two    one    two    one
   A  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309 -1.170299
   B  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127
   C -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466
   first   qux
  second  two
   A   -0.226169
   B   -1.436737
   C   -2.006747
```

```python
In [22]: pd.set_option('display.multi_sparse', True)
```

### 13.1.2 Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

```python
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')
```
### 13.1.3 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:

```python
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']

   A    0.895717
   B    0.410835
   C   -1.413681
Name: (bar, one), dtype: float64

In [27]: df['bar']['one']

   A    0.895717
   B    0.410835
   C   -1.413681
Name: one, dtype: float64

In [28]: s['qux']

   one   -1.039575
   two    0.271860
dtype: float64
```

See *Cross-section with hierarchical index* for how to select on a deeper level.

### 13.1.4 Defined Levels

The repr of a MultiIndex shows ALL the defined levels of an index, even if the they are not actually used. When slicing an index, you may notice this. For example:

```python
# original multi-index
In [29]: df.columns
Out[29]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
          labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
          names=['first', 'second'])

# sliced
In [30]: df[['foo','qux']].columns
Out[30]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
          labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
          names=['first', 'second'])
```

13.1. Hierarchical indexing (MultiIndex)
This is done to avoid a recomputation of the levels in order to make slicing highly performant. If you want to see the actual used levels.

```python
In [31]: df[['foo', 'qux']].columns.values
Out[31]: array([('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')], dtype=object)
```

# for a specific level
```python
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

New in version 0.20.0.

```python
In [33]: df[['foo', 'qux']].columns.remove_unused_levels()
Out[33]: MultiIndex(levels=[['foo', 'qux'], ['one', 'two']], labels=[[0, 0, 1, 1], [0, 1, 0, 1]], names=['first', 'second'])
```

### 13.1.5 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 [34]: s + s[:-2]
Out[34]:
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
```

```python
In [35]: s + s[::2]
```
```
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
```

`reindex` can be called with another `MultiIndex` or even a list or array of tuples:

```python
In [36]: s.reindex(index[:3])
Out[36]:
first  second
```
```
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
```
13.2 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. For example the following works as you would expect:

```python
In [38]: df = df.T

In [39]: df
Out[39]:
   A      B      C
first
  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 -0.226169 -1.436737 -2.006747

In [40]: df.loc['bar']
Out[40]:
   A      B      C
  one  0.895717  0.410835 -1.413681
two  0.805244  0.813850  1.607920

In [41]: df.loc['bar', 'two']
Out[41]:
       A     B     C
Name: (bar, two), dtype: float64

Partial" slicing also works quite nicely.

```
You can slice with a ‘range’ of values, by providing a slice of tuples.

```
In [43]: df.loc[('baz', 'two'):('qux', 'one')]
Out[43]:
   A    B    C
first second
baz 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
```

```
In [44]: df.loc[('baz', 'two'):'foo']
```

Passing a list of labels or tuples works similar to reindexing:

```
In [45]: df.loc[[('bar', 'two'), ('qux', 'one')]]
Out[45]:
   A    B    C
first second
bar two 0.805244 0.813850 1.607920
qux one -1.170299 1.130127 0.974466
```

### 13.2.1 Using slicers

You can slice a multi-index 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(0)`, to select all the contents of that level. You do not need to specify all the deeper levels, they will be implied as `slice(0)`.

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-interpretied as indexing both axes, rather than into say the MultiIndex for the rows.

You should do this:

```
df.loc[({"slice\('A1', 'A3')", .....}, :]
```

rather than this:
df.loc[('A1','A3'),:]

In [46]: def mklbl(prefix,n):
   ....:     return ["%s%s" % (prefix,i) for i in range(n)]
   ....:

In [47]: miindex = pd.MultiIndex.from_product([mklbl('A',4),
   ....:     mklbl('B',2),
   ....:     mklbl('C',4),
   ....:     mklbl('D',2))
   ....:

In [48]: micolumns = pd.MultiIndex.from_tuples([('a','foo'),('a','bar'),
   ....:     ('b','foo'),('b','bah')],
   ....:     names=['lvl0', 'lvl1'])
   ....:

In [49]: dfmi = pd.DataFrame(np.arange(len(miindex)*len(micolumns)).reshape((len(miindex),len(micolumns))),
   ....:     index=miindex,
   ....:     columns=micolumns).sort_index().sort_index(axis=1)
   ....:

In [50]: dfmi
Out[50]:

lvl0 lvl1 a b
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 21 20 23 22
C3 D0 25 24 27 26
 ... ... ... ... ...
A3 B1 C0 D1 229 228 231 230
 C1 D0 233 232 235 234
 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 multi-index slicing using slices, lists, and labels.

In [51]: dfmi.loc[('A1','A3'), slice(None), ['C1', 'C3'], :]
Out[51]:

lvl0 lvl1 a b
A1 B0 C1 D0 73 72 75 74
 D1 77 76 79 78
C2 D0 89 88 91 90
 D1 93 92 95 94
B1 C1 D0 105 104 107 106

13.2. Advanced indexing with hierarchical index
You can use a `pd.IndexSlice` to have a more natural syntax using `:` rather than using `slice(None)`

```
In [52]: idx = pd.IndexSlice

In [53]: dfmi.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
     D1  44  46
  C3  D0  56  58
... ... ...
  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
[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]: dfmi.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
     D1  84  86
  C3  D0  88  90
... ... ...
  B1  C0  D1  100  102
  C1  D0  104  106
     D1  108  110
  C2  D0  112  114
     D1  116  118
```

672 Chapter 13. MultiIndex / Advanced Indexing
Using a boolean indexer you can provide selection related to the values.

```python
In [56]: mask = dfmi[('a', 'foo')] > 200

In [57]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```python
In [58]: dfmi.loc(axis=0)[ :, :, ['C1', 'C3']]
```
Furthermore you can set the values using these methods

```python
In [59]: df2 = dfmi.copy()

In [60]: df2.loc(axis=0)[[:, :, ['C1', 'C3']]] = -10

In [61]: df2
```

```
Out[61]:
        a     b
   lvl0
   lvl1  bar  foo  bah  foo
   A0  B0  C0  D0     1     0     3     2
       5     4     7     6
       -10   -10   -10   -10
       -10   -10   -10   -10
       17    16    19    18
       21    20    23    22
       229   228   231   230
   ...   ...   ...   ...   ...
   A3  B1  C0  D1  229  228  231  230
   C1  D0  -10   -10   -10   -10
       -10   -10   -10   -10
       241   240   243   242
       245   244   247   246
   C3  D0  -10   -10   -10   -10
       -10   -10   -10   -10
       8000   8000  11000  10000
       13000  12000  15000  14000
       17     16     19     18
       21     20     23     22
       25000  24000  27000  26000
   ...   ...   ...   ...   ...
   A3  B1  C0  D1  229  228  231  230
   C1  D0  233000 232000 235000 234000
```

[64 rows x 4 columns]

You can use a right-hand-side of an alignable object as well.

```python
In [62]: df2 = dfmi.copy()

In [63]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [64]: df2
```

```
Out[64]:
        a     b
   lvl0
   lvl1  bar  foo  bah  foo
   A0  B0  C0  D0     1     0     3     2
       5     4     7     6
       9000  8000 11000 10000
       13000 12000 15000 14000
       17     16     19     18
       21     20     23     22
       25000 24000 27000 26000
   ...   ...   ...   ...   ...
   A3  B1  C0  D1  229  228  231  230
   C1  D0  233000 232000 235000 234000
```

[64 rows x 4 columns]
13.2.2 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 [65]: df
Out[65]:
   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  -0.226169 -1.436737 -2.006747

In [66]: df.xs('one', level='second')
Out[66]:
   A    B    C
bar   0.895717  0.410835 -1.413681
baz  -1.206412  0.132003  1.024180
foo   1.431256 -0.076467  0.875906
qux  -1.170299  1.130127  0.974466

# using the slicers
In [67]: df.loc[(slice(None),'one'),:]
Out[67]:
   A    B    C
first second
bar  one  0.895717  0.410835 -1.413681
baz  one -1.206412  0.132003  1.024180
foo  one  1.431256 -0.076467  0.875906
qux  one -1.170299  1.130127  0.974466

You can also select on the columns with `xs()`, by providing the `axis` argument

```python
In [68]: df = df.T

In [69]: df.xs('one', level='second', axis=1)
Out[69]:
   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
```

13.2. Advanced indexing with hierarchical index
# using the slicers

In [70]: df.loc[:,(slice(None),'one')]

Out[70]:

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
<th>baz</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>0.895717</td>
<td>-1.206412</td>
<td>1.431256</td>
<td>-1.170299</td>
</tr>
<tr>
<td>second</td>
<td>0.410835</td>
<td>0.132003</td>
<td>-0.076467</td>
<td>1.130127</td>
</tr>
<tr>
<td></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 [71]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)

Out[71]:

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>0.895717</td>
</tr>
<tr>
<td>second</td>
<td>0.410835</td>
</tr>
<tr>
<td></td>
<td>-1.413681</td>
</tr>
</tbody>
</table>

You can pass drop_level=False to xs() to retain the level that was selected

In [73]: df.xs('one', level='second', axis=1, drop_level=False)

Out[73]:

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
<th>baz</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>0.895717</td>
<td>-1.206412</td>
<td>1.431256</td>
<td>-1.170299</td>
</tr>
<tr>
<td>second</td>
<td>0.410835</td>
<td>0.132003</td>
<td>-0.076467</td>
<td>1.130127</td>
</tr>
<tr>
<td></td>
<td>-1.413681</td>
<td>1.024180</td>
<td>0.875906</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

versus the result with drop_level=True (the default value)

In [74]: df.xs('one', level='second', axis=1, drop_level=True)

Out[74]:

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
<th>baz</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>0.895717</td>
<td>-1.206412</td>
<td>1.431256</td>
<td>-1.170299</td>
</tr>
<tr>
<td>second</td>
<td>0.410835</td>
<td>0.132003</td>
<td>-0.076467</td>
<td>1.130127</td>
</tr>
<tr>
<td></td>
<td>-1.413681</td>
<td>1.024180</td>
<td>0.875906</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

13.2.3 Advanced reindexing and alignment

The parameter level has been added to the reindex and align methods of pandas objects. This is useful to broadcast values across a level. For instance:

In [75]: midx = pd.MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
                             labels=[[1,1,0,0],[1,0,1,0]])

In [76]: df = pd.DataFrame(np.random.randn(4,2), index=midx)
13.2.4 Swapping levels with `swaplevel()`

The `swaplevel` function can switch the order of two levels:

```
In [84]: df[:5]
Out[84]:
          0      1
    one   y  1.519970 -0.493662
           x  0.600178  0.274230
    zero   y  0.132885 -0.023688
           x  2.410179  1.450520
```
13.2.5 Reordering levels with `reorder_levels()`

The `reorder_levels` function generalizes the `swaplevel` function, allowing you to permute the hierarchical index levels in one step:

```
In [86]: df[:5].reorder_levels([1,0], axis=0)
```

```
Out[86]:
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
```

13.3 Sorting a MultiIndex

For MultiIndex-ed objects to be indexed & sliced effectively, they need to be sorted. As with any index, you can use `sort_index`.

```
In [87]: import random; random.shuffle(tuples)
In [88]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))
In [89]: s
Out[89]:
qux one  0.206053
bar one  -0.251905
foo one  -2.213588
bar two   1.063327
foo two   1.266143
qux two   0.299368
baz two  -0.863838
one       0.408204
dtype: float64
```

```
In [90]: s.sort_index()
```

```
bar one  -0.251905
two   1.063327
baz one  0.408204
two  -0.863838
foo one  -2.213588
two   1.266143
```
You may also pass a level name to `sort_index` if the MultiIndex levels are named.

```python
In [93]: s.index.set_names(['L1', 'L2'], inplace=True)

In [94]: s.sort_index(level='L1')

Out[94]:
L1  L2
bar one -0.251905
      two  1.063327
baz one  0.408204
      two -0.863838
foo one -2.213588
      two  1.266143
qux one  0.206053
      two  0.299368
dtype: float64

In [95]: s.sort_index(level='L2')
```

### 13.3. Sorting a MultiIndex

```python
qux one  0.206053
two  0.299368
dtype: float64

In [91]: s.sort_index(level=0)

bar one  -0.251905
two  1.063327
baz one  0.408204
two -0.863838
foo one -2.213588
two  1.266143
qux one  0.206053
two  0.299368
dtype: float64
```

You may also pass a level name to `sort_index` if the MultiIndex levels are named.

```python
In [93]: s.index.set_names(['L1', 'L2'], inplace=True)

In [94]: s.sort_index(level='L1')

Out[94]:
L1  L2
bar one -0.251905
      two  1.063327
baz one  0.408204
      two -0.863838
foo one -2.213588
      two  1.266143
qux one  0.206053
      two  0.299368
dtype: float64

In [95]: s.sort_index(level='L2')
```
On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

```python
In [96]: df.T.sort_index(level=1, axis=1)
Out[96]:
          zero    one    zero    one
         x      x      y      y
0  2.410179  0.600178  0.132885  1.519970
1  1.450520  0.274230 -0.023688 -0.493662
```

Indexing will work even if the data are not sorted, but will be rather inefficient (and show a `PerformanceWarning`). It will also return a copy of the data rather than a view:

```python
In [97]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
                       'joe': ['x', 'x', 'z', 'y'],
                       'jolie': np.random.rand(4)})

In [98]: dfm = dfm.set_index(['jim', 'joe'])

In [99]: dfm
Out[99]:
       jolie
    jim  joe
0      x  0.490671
      x  0.120248
1      z  0.537020
      y  0.110968

In [4]: dfm.loc[(1, 'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.
Out[4]:
       jolie
    jim  joe
1      z  0.64094
```

Furthermore if you try to index something that is not fully lexsorted, this can raise:

```python
In [5]: dfm.loc[(0, 'y'):(1, 'z')]
UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'
```

The `is_lexsorted()` method on an `Index` show if the index is sorted, and the `lexsort_depth` property returns the sort depth:

```python
In [100]: dfm.index.is_lexsorted()
Out[100]: False

In [101]: dfm.index.lexsort_depth
Out[101]: 1
```

```python
In [102]: dfm = dfm.sort_index()

In [103]: dfm
```

```
```
In [104]: dfm.index.is_lexsorted()
True

In [105]: dfm.index.lexsort_depth
2

And now selection works as expected.

In [106]: dfm.loc[(0, 'y'): (1, 'z')]
Out[106]:
jolie
jim joe
1 y 0.110968
z 0.537020

13.4 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 [107]: index = pd.Index(np.random.randint(0, 1000, 10))
In [108]: index
Out[108]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64')
In [109]: positions = [0, 9, 3]
In [110]: index[positions]
Out[110]: Int64Index([214, 329, 567], dtype='int64')
In [111]: index.take(positions)
Out[111]: Int64Index([214, 329, 567], dtype='int64')
In [112]: ser = pd.Series(np.random.randn(10))
In [113]: ser.iloc[positions]
Out[113]:
0  -0.179666
9   1.824375
3   0.392149
dtype: float64
In [114]: ser.take(positions)

13.4. Take Methods
For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```python
In [115]: frm = pd.DataFrame(np.random.randn(5, 3))
In [116]: frm.take([1, 4, 3])
Out[116]:
   0   1   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.

```python
In [118]: arr = np.random.randn(10)
In [119]: arr.take([False, False, True, True])
Out[119]: array([-1.1935, -1.1935, 0.6775, 0.6775])
In [120]: arr[[0, 1]]
Out[120]: array([-1.1935, 0.6775])
In [121]: ser = pd.Series(np.random.randn(10))
In [122]: ser.take([False, False, True, True])
Out[122]:
   0  0.233141
   1 -0.223540
dtype: float64
In [123]: ser.iloc[[0, 1]]
Out[123]:
   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.
13.5 Index Types

We have discussed MultiIndex in the previous sections pretty extensively. DatetimeIndex and PeriodIndex are shown here. TimedeltaIndex are here.

In the following sub-sections we will highlight some other index types.

13.5.1 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.

```
In [124]: from pandas.api.types import CategoricalDtype

In [125]: df = pd.DataFrame({'A': np.arange(6),
                           'B': list('aabca'))

In [126]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))  

In [127]: df
Out[127]:   A   B
0   0  a
1   1  a
2   2  b
3   3  b
4   4  c
5   5  a

In [128]: df.dtypes
Out[128]:
A   int64
B   category
dtype: object

In [129]: df.B.cat.categories
Out[129]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index, will create a CategoricalIndex

```
In [130]: df2 = df.set_index('B')

In [131]: df2.index
Out[131]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Indexing with __getitem__/.iloc/.loc works similarly to an Index with duplicates. The indexers MUST be in the category or the operation will raise.

```
In [132]: df2.loc['a']
Out[132]:
   A
0 a
1 a
2 b
3 b
4 c
5 a
```
These PRESERVE the `CategoricalIndex`

```python
In [133]: df2.loc['a'].index
Out[133]: CategoricalIndex([‘a’, ’a’, ’a’], categories=['c', 'a', 'b'], ordered=False,
name='B', dtype='category')
```

Sorting will order by the order of the categories

```python
In [134]: df2.sort_index()
Out[134]:
   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 [135]: df2.groupby(level=0).sum()
Out[135]:
   A   B
   c  4
   a  6
   b  5
```

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.

```python
In [137]: df2.reindex([‘a’, ‘e’])
Out[137]:
   A   B
   a  0.0
   a  1.0
   a  5.0
   e  NaN
```

```python
In [138]: df2.reindex([‘a’, ‘e’]).index
```

```python
In [139]: df2.reindex(pd.Categorical([‘a’, ‘e’], categories=list(‘abcde’)))
```
13.5.2 Int64Index and RangeIndex

**Warning:** Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see [here](#).

Int64Index is a fundamental basic index in pandas. This is an Immutable array implementing an ordered, sliceable set. Prior to 0.18.0, the Int64Index would provide the default index for all NDFrame objects.

RangeIndex is a sub-class of Int64Index added in version 0.18.0, now providing 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.

13.5.3 Float64Index

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.
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 [145]: sf[3]
Out[145]: 2

In [146]: sf[3.0]
Out[146]: 2

In [147]: sf.loc[3]
Out[147]: 2

In [148]: sf.loc[3.0]
Out[148]: 2

The only positional indexing is via iloc

In [149]: sf.iloc[3]
Out[149]: 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 [150]: sf[2:4]
Out[150]:
2.0  1
3.0  2
dtype: int64

In [151]: sf.loc[2:4]
Out[151]:
2.0  1
3.0  2
dtype: int64

In [152]: sf.iloc[2:4]
Out[152]:
2.0  1
3.0  2
4.5  3
dtype: int64

In float indexes, slicing using floats is allowed

In [153]: sf[2.1:4.6]
Out[153]:
3.0  2
In non-float indexes, slicing using floats will 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
```

**Warning:** Using a scalar float indexer for `.iloc` has been removed in 0.18.0, so the following will raise a `TypeError`

```
In [3]: pd.Series(range(5)).iloc[3.0]
TypeError: cannot do positional indexing on <class 'pandas.indexes.range.RangeIndex'> with these indexers [3.0] of <type 'float'>
```

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.

```
In [155]: dfir = pd.concat([pd.DataFrame(np.random.randn(5,2),
    index=np.arange(5) * 250.0,
    columns=list('AB')),
    pd.DataFrame(np.random.randn(6,2),
    index=np.arange(4,10) * 250.1,
    columns=list('AB'))])

In [156]: dfir
Out[156]:
       A       B
0  0.92218  1.34758
250.0  0.53179  0.54072
500.0 -0.86531  0.86334
750.0 -1.11981 -1.11238
1000.0 -0.87052  0.18584
1000.4  0.52672  0.41591
1250.5 -0.66537  0.47226
1500.6 -0.33089  0.86532
1750.7 -0.57198  0.96090
2000.8  1.38497  0.38213
2250.9 -0.28947  1.38213

Selection operations then will always work on a value basis, for all selection operators.

```
In [157]: dfir[0:1000.4]
Out[157]:
       A       B
0  0.99729 -1.69332
250.0 -0.84216 -0.06172
500.0  0.73078  0.50302
750.0  0.07423 -0.70263
1000.0 -0.71010 -0.25007
1000.4  0.31050 -1.44398
1250.5 -0.66909  0.16581
1500.6 -0.23082  0.43024
1750.7 -0.17948 -0.67804
2000.8 -2.57637  0.06379
2250.9 -0.27581  0.46387
```

---

13.5. Index Types
In [158]: dfir.loc[0:1001,'A']

    0.0      0.997289
     250.0    -0.179129
      500.0     0.936914
      750.0    -1.003401
     1000.0    -0.724626
     1000.4     0.310610

Name: A, dtype: float64

In [159]: dfir.loc[1000.4]

   A   0.310610
   B  -0.108002

Name: 1000.4, dtype: float64

You could then easily pick out the first 1 second (1000 ms) of data then.

In [160]: dfir[0:1000]

   A   B
   0.0 0.997289 -1.693316
   250.0 -0.179129 -1.598062
   500.0  0.936914  0.912560
   750.0 -1.003401  1.632781
  1000.0 -0.724626  0.178219

Of course if you need integer based selection, then use iloc

In [161]: dfir.iloc[0:5]

   A   B
   0.0 0.997289 -1.693316
   250.0 -0.179129 -1.598062
   500.0  0.936914  0.912560
   750.0 -1.003401  1.632781
  1000.0 -0.724626  0.178219

13.5.4 IntervalIndex

```
IntervalIndex together with its own dtype, interval 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().

New in version 0.20.0.
**Warning:** These indexing behaviors are provisional and may change in a future version of pandas.

An `IntervalIndex` can be used in `Series` and in `DataFrame` as the index.

```python
In [162]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
                       index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
In [163]: df
Out[163]:
   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.

```python
In [164]: df.loc[2]  
Out[164]:
   A
2
Name: (1, 2], dtype: int64
In [165]: df.loc[[2, 3]]
Out[165]:
   A
(1, 2]  2
(2, 3]  3
```

If you select a label `contained` within an interval, this will also select the interval.

```python
In [166]: df.loc[2.5]  
Out[166]:
   A
3
Name: (2, 3], dtype: int64
In [167]: df.loc[[2.5, 3.5]]
Out[167]:
   A
(2, 3]  3
(3, 4]  4
```

Interval and `IntervalIndex` are used by `cut` and `qcut`:

```python
In [168]: c = pd.cut(range(4), bins=2)
In [169]: c
Out[169]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
In [170]: c.categories
→
IntervalIndex([(-0.003, 1.5], (1.5, 3.0])
```
Furthermore, `IntervalIndex` allows one to bin *other* data with these same bins, with NaN representing a missing value similar to other dtypes.

```python
In [171]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[171]: 
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]
Categories (2, interval[float64]): 
[(-0.003, 1.5] < (1.5, 3.0]]
```

## 13.6 Miscellaneous indexing FAQ

### 13.6.1 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:

```python
s = pd.Series(range(5))
s[-1]
df = pd.DataFrame(np.random.randn(5, 4))
df
df.loc[-2:]
```

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).

### 13.6.2 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 [172]: df = pd.DataFrame(index=[2,3,3,4,5], columns=['data'], data=list(range(5)))
In [173]: df.index.is_monotonic_increasing
Out[173]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [174]: df.loc[0:4, :]

<table>
<thead>
<tr>
<th></th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

# slice is are outside the index, so empty DataFrame is returned
In [175]: df.loc[13:15, :]

Empty DataFrame
```

690 Chapter 13. MultiIndex / Advanced Indexing
On the other hand, if the index is not monotonic, then both slice bounds must be unique members of the index.

```python
In [176]: df = pd.DataFrame(index=[2,3,1,4,3,5], columns=['data'],
                         data=list(range(6)))
In [177]: df.index.is_monotonic_increasing
Out[177]: False

# OK because 2 and 4 are in the index
In [178]: df.loc[2:4, :]
   data
  2  0
  3  1
  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 `Index.is_unique()`.

```python
In [179]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])
In [180]: weakly_monotonic
Out[180]: Index(['a', 'b', 'c', 'c'], dtype='object')
In [181]: weakly_monotonic.is_monotonic_increasing
Out[181]: True
In [182]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_unique
Out[182]: False
```

### 13.6.3 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:

```python
In [183]: s = pd.Series(np.random.randn(6), index=list('abcdef'))
In [184]: s
Out[184]:
       a    b
0  0.112246  0.871721
```

13.6. Miscellaneous indexing FAQ
Suppose we wished to slice from `c` to `e`, using integers this would be

```python
In [185]: s[2:5]
Out[185]:
c -0.816064
d -0.784880
e 1.030659
dtype: float64
```

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:

```python
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 to make label-based slicing include both endpoints:

```python
In [186]: s.loc['c':'e']
Out[186]:
c -0.816064
d -0.784880
e 1.030659
dtype: float64
```

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.

### 13.6.4 Indexing potentially changes underlying Series dtype

The different indexing operation can potentially change the dtype of a `Series`.

```python
In [187]: series1 = pd.Series([1, 2, 3])
In [188]: series1.dtype
Out[188]: dtype('int64')
In [189]: res = series1.reindex([0, 4])
In [190]: res.dtype
Out[190]: dtype('float64')
```

```python
In [191]: res
Out[191]:
0 1.0
4 NaN
dtype: float64
```

```python
In [192]: series2 = pd.Series([True])
In [193]: series2.dtype
```

```python
```

---

Chapter 13. MultiIndex / Advanced Indexing
Out[193]: dtype('bool')

In [194]: res = series2.reindex_like(series1)

In [195]: res.dtype
Out[195]: dtype('O')

In [196]: res

Out[196]:
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 this old issue for a more detailed discussion.
14.1 Statistical Functions

14.1.1 Percent Change

Series, DataFrame, and Panel all 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).

```python
In [1]: ser = pd.Series(np.random.randn(8))
In [2]: ser.pct_change()
Out[2]:
   0    NaN
   1 -1.60298
   2  4.33494
   3 -0.24746
   4 -2.06735
   5 -1.14290
   6 -1.68821
   7 -9.75973
   dtype: float64
```

```python
In [3]: df = pd.DataFrame(np.random.randn(10, 4))
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 -0.21832 1.98714 -0.51018 1.98714
4 -0.43912 0.64971 4.82281 -4.82281
5 -0.12783 -5.86660 -1.77677 -5.86660
6 -2.96833 0.41697 1.38083 -1.38083
7 -0.11782 -0.03609 -0.86797 -0.86797
8 2.49261 -1.55869 -1.00374 -1.55869
9 -1.01298 2.32456 0.43130 2.32456
```

14.1.2 Covariance

The `Series` object has a method `cov` to compute covariance between series (excluding NA/null values).
Analogously, DataFrame has a method `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.

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.
14.1.3 Correlation

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.

Note: Please see the caveats associated with this method of calculating correlation matrices in the covariance section.

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]: 0.013479040400098794

In [18]: frame['a'].corr(frame['b'], method='spearman')
Out[18]: -0.0072898851595406371

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:

   a    b    c    d    e
a  1.00  0.01  0.00 -0.00 -0.02
b  0.01  1.00 -0.02  0.00  0.00
c -0.00 -0.02  1.00  0.01 -0.05
d -0.00  0.00  0.01  1.00 -0.02
e -0.02  0.00  0.05 -0.05  1.00

Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

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  1.00 -0.12  0.07
b -0.12  1.00  0.05
c  0.07  0.05  1.00

In [24]: frame.corr(min_periods=12)

   a    b    c
a  1.00 NaN  0.07
b NaN  1.00  0.05
c  0.07  0.05  1.00
A related method `corrwith` is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

```python
In [25]: index = ['a', 'b', 'c', 'd', 'e']
In [26]: columns = ['one', 'two', 'three', 'four']
In [27]: df1 = pd.DataFrame(np.random.randn(5, 4), index=index, columns=columns)
In [28]: df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4], columns=columns)
In [29]: df1.corrwith(df2)
Out[29]:
   one   two  
one -0.125501  
two -0.493244  
three 0.344056  
four  0.004183  
dtype: float64
```

```python
In [30]: df2.corrwith(df1, axis=1)
   a   b  
a -0.675817 0.458296  
b 0.190809  
cd -0.186275 NaN  
dtype: float64
```

### 14.1.4 Data ranking

The `rank` method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```python
In [31]: s = pd.Series(np.random.randn(5), index=list('abcde'))
In [32]: s['d'] = s['b'] # so there's a tie
In [33]: s.rank()
Out[33]:
   a  5.0
   b  2.5
c  1.0
d  2.5
e  NaN
dtype: float64
```

`rank` is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). NaN values are excluded from the ranking.

```python
In [34]: df = pd.DataFrame(np.random.randn(10, 6))
```

Chapter 14. Computational tools
In [36]: df
Out[36]:
   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  0.204848  NaN    0.463347
6  0.376892  0.959292  0.095572 -0.593740  NaN    -0.069180
7 -1.002601  1.957794 -0.120708  0.094214  NaN    -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254  NaN    -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477  NaN    1.375064

In [37]: df.rank(1)

   rank
0  4.0  3.0  1.5  5.0  1.5  6.0
1  2.0  6.0  4.5  1.0  4.5  3.0
2  1.0  6.0  3.5  5.0  3.5  2.0
3  4.0  5.0  1.5  3.0  1.5  6.0
4  5.0  3.0  1.5  4.0  1.5  6.0
5  1.0  2.0  5.0  3.0  NaN  4.0
6  4.0  5.0  3.0  1.0  NaN  2.0
7  2.0  5.0  3.0  4.0  NaN  1.0
8  2.0  5.0  3.0  4.0  NaN  1.0
9  2.0  3.0  1.0  4.0  NaN  5.0

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

### 14.2 Window Functions

**Warning**: Prior to version 0.18.0, `pd.rolling_*`, `pd.expanding_*`, and `pd.ewm*` were module level functions and are now deprecated. These are replaced by using the `Rolling`, `Expanding` and `EWM` objects and a corresponding method call.

The deprecation warning will show the new syntax, see an example [here](#). You can view the previous documentation [here](#).

For working with data, a number of windows functions are provided for computing common window or rolling statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis.
pandas: powerful Python data analysis toolkit, Release 0.21.0

Starting in version 0.18.1, the `rolling()` and `expanding()` functions can be used directly from DataFrameGroupBy objects, see the `groupby docs`.

**Note:** The API for window statistics is quite similar to the way one works with GroupBy objects, see the documentation [here](#).

We work with rolling, expanding and exponentially weighted data through the corresponding objects, Rolling, Expanding and EWM.

```python
In [38]: s = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [39]: s = s.cumsum()
In [40]: s
Out[40]:
2000-01-01 -0.268824
2000-01-02 -1.771855
2000-01-03 -0.818003
2000-01-04 -0.659244
2000-01-05 -1.942133
2000-01-06 -1.869391
2000-01-07  0.563674
...  2002-09-20 -68.233054
2002-09-21 -66.765687
2002-09-22 -67.457323
2002-09-23 -69.253182
2002-09-24 -70.296818
2002-09-25 -70.844674
2002-09-26 -72.475016
Freq: D, Length: 1000, dtype: float64
```

These are created from methods on Series and DataFrame.

```python
In [41]: r = s.rolling(window=60)
In [42]: r
Out[42]: Rolling [window=60, center=False, axis=0]
```

These object provide tab-completion of the avaible methods and properties.

```python
In [14]: r.
r.agg  r.apply  r.count  r.exclusions  r.max  r.median  r.
r.name  r.skew  r.sum
r.aggregate  r.corr  r.cov  r.kurt  r.mean  r.min  r.
r.quantile  r.std  r.var
```

Generally these methods all have the same interface. They all accept the following arguments:

- **window**: size of moving window
- **min_periods**: threshold of non-null data points to require (otherwise result is NA)
- **center**: boolean, whether to set the labels at the center (default is False)
Warning: The `freq` and `how` arguments were in the API prior to 0.18.0 changes. These are deprecated in the new API. You can simply resample the input prior to creating a window function.

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').max().rolling(window=5).max()`, which first resamples the data to daily data, then provides a rolling 5 day window.

We can then call methods on these rolling objects. These return like-indexed objects:

```python
In [43]: r.mean()
Out[43]:
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05    NaN
2000-01-06    NaN
2000-01-07    NaN
          ...  
2002-09-20   -62.694135
2002-09-21   -62.812190
2002-09-22   -62.914971
2002-09-23   -63.061867
2002-09-24   -63.213876
2002-09-25   -63.375074
2002-09-26   -63.539734
Freq: D, Length: 1000, dtype: float64
```

```python
In [44]: s.plot(style='k--')
Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x121ceb748>
```

```python
In [45]: r.mean().plot(style='k')

Out[45]: <matplotlib.axes._subplots.AxesSubplot at 0x121ceb748>
```
They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrame’s columns:

```python
In [46]: df = pd.DataFrame(np.random.randn(1000, 4),
       index=pd.date_range('1/1/2000', periods=1000),
       columns=['A', 'B', 'C', 'D'])

In [47]: df = df.cumsum()

In [48]: df.rolling(window=60).sum().plot(subplots=True)
```

Out[48]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x116d566d8>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x121eb3320>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x121ff3160>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x12201ed30>], dtype=object)
```
14.2.1 Method Summary

We provide a number of the common statistical functions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Number of non-null 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>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>std()</td>
<td>Bessel-corrected sample standard deviation</td>
</tr>
<tr>
<td>var()</td>
<td>Unbiased variance</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>apply()</td>
<td>Generic apply</td>
</tr>
<tr>
<td>cov()</td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td>corr()</td>
<td>Correlation (binary)</td>
</tr>
</tbody>
</table>

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. Suppose we wanted to compute the mean absolute deviation on a rolling basis:

```python
In [49]: mad = lambda x: np.fabs(x - x.mean()).mean()
In [50]: s.rolling(window=60).apply(mad).plot(style='k')
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x122067d68>
```
### 14.2.2 Rolling Windows

Passing `win_type` to `.rolling` generates a generic rolling window computation, that is weighted according the `win_type`. The following methods are available:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum()</code></td>
<td>Sum of values</td>
</tr>
<tr>
<td><code>mean()</code></td>
<td>Mean of values</td>
</tr>
</tbody>
</table>

The weights used in the window are specified by the `win_type` keyword. The list of recognized types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

```
In [51]: ser = pd.Series(np.random.randn(10), index=pd.date_range('1/1/2000', periods=10))

In [52]: ser.rolling(window=5, win_type='triang').mean()
Out[52]:
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05   -1.037870
2000-01-06   -0.767705
2000-01-07   -0.383197
2000-01-08   -0.395513
2000-01-09   -0.558440
2000-01-10   -0.672416
Freq: D, dtype: float64
```

Note that the `boxcar` window is equivalent to `mean()`.

```
In [53]: ser.rolling(window=5, win_type='boxcar').mean()
Out[53]:
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

```
In [54]: ser.rolling(window=5).mean()
```

```
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [55]: ser.rolling(window=5, win_type='gaussian').mean(std=0.1)
Out[55]:
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05   -1.309989
2000-01-06   -1.429976
2000-01-07   -1.455531
2000-01-08   -1.429976
2000-01-09   -1.309989
2000-01-10   -1.309989
Freq: D, dtype: float64
```

14.2. Window Functions
Note: For `.sum()` with a `win_type`, there is no normalization done to the weights for the window. Passing custom weights of `[1, 1, 1]` will yield a different result than passing weights of `[2, 2, 2]`, for example. When passing a `win_type` instead of explicitly specifying the weights, the weights are already normalized so that the largest weight is 1.

In contrast, the nature of the `.mean()` calculation is such that the weights are normalized with respect to each other. Weights of `[1, 1, 1]` and `[2, 2, 2]` yield the same result.

### 14.2.3 Time-aware Rolling

New in version 0.19.0.

New in version 0.19.0 are the ability to pass an offset (or convertible) to a `.rolling()` method and have it produce variable sized windows based on the passed time window. For each time point, this includes all preceding values occurring within the indicated time delta.

This can be particularly useful for a non-regular time frequency index.

```
In [56]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
                   index=pd.date_range('20130101 09:00:00', periods=5, freq='s'))
...:
In [57]: dft
Out[57]:
    B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [58]: dft.rolling(2).sum()
Out[58]:
     B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN
```

```
In [59]: dft.rolling(2, min_periods=1).sum()
  ...:
     B
2013-01-01 09:00:00  0.0
```
Specifying an offset allows a more intuitive specification of the rolling frequency.

```python
In [60]: dft.rolling('2s').sum()
Out[60]:
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 09:00:04 2.0
```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```python
In [61]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4],
   ....: index = pd.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')],
   ....: name='foo'))
   ....:)
In [62]: dft
Out[62]:
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
```

In [63]: dft.rolling(2).sum()

Using the time-specification generates variable windows for this sparse data.

```python
In [64]: dft.rolling('2s').sum()
```

### 14.2. Window Functions
Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a DataFrame.

```python
In [65]: dft = dft.reset_index()

In [66]: dft
Out[66]:
   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

In [67]: dft.rolling('2s', on='foo').sum()
   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
```

### 14.2.4 Rolling Window Endpoints

New in version 0.20.0.

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

<table>
<thead>
<tr>
<th>closed</th>
<th>Description</th>
<th>Default for</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>close right endpoint</td>
<td>time-based windows</td>
</tr>
<tr>
<td>left</td>
<td>close left endpoint</td>
<td></td>
</tr>
<tr>
<td>both</td>
<td>close both endpoints</td>
<td>fixed windows</td>
</tr>
<tr>
<td>neither</td>
<td>open endpoints</td>
<td></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 [68]: 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 [69]: df['right'] = df.rolling('2s', closed='right').x.sum()  # default

In [70]: df['both'] = df.rolling('2s', closed='both').x.sum()

In [71]: df['left'] = df.rolling('2s', closed='left').x.sum()

In [72]: df['neither'] = df.rolling('2s', closed='neither').x.sum()
```
Currently, this feature is only implemented for time-based windows. For fixed windows, the closed parameter cannot
be set and the rolling window will always have both endpoints closed.

### 14.2.5 Time-aware Rolling vs. Resampling

Using `.rolling()` with a time-based index is quite similar to `resampling`. They both operate and perform reductive
operations on time-indexed pandas objects.

When using `.rolling()` with an offset. The offset is a time-delta. Take a backwards-in-time looking window, and
aggregate all of the values in that window (including the end-point, but not the start-point). This is the new value at
that point in the result. These are variable sized windows in time-space for each point of the input. You will get a same
sized result as the input.

When using `.resample()` with an offset. Construct a new index that is the frequency of the offset. For each
frequency bin, aggregate points from the input within a backwards-in-time looking window that fall in that bin. The
result of this aggregation is the output for that frequency point. The windows are fixed size size in the frequency space.
Your result will have the shape of a regular frequency between the min and the max of the original input object.

To summarize, `.rolling()` is a time-based window operation, while `.resample()` is a frequency-based window
operation.

### 14.2.6 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.
14.2.7 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 MultiIndexed `DataFrame` whose index are the dates in question (see the next section).

For example:

```python
In [76]: df = pd.DataFrame(np.random.randn(1000, 4),
                      index=pd.date_range('1/1/2000', periods=1000),
                      columns=['A', 'B', 'C', 'D'])
In [77]: df = df.cumsum()
In [78]: df2 = df[:20]
In [79]: df2.rolling(window=5).corr(df2['B'])
```

```
Out[79]:
     A      B      C      D
2000-01-01 NaN    NaN    NaN    NaN
2000-01-02 NaN    NaN    NaN    NaN
2000-01-03 NaN    NaN    NaN    NaN
2000-01-04 NaN    NaN    NaN    NaN
2000-01-05 0.768775 1.0 -0.977990 0.800252
2000-01-06 0.744106 1.0 -0.967912 0.830021
2000-01-07 0.683257 1.0 -0.928969 0.384916
...       ...    ...    ...    ...
2000-01-14 -0.392318 1.0  0.570240 -0.591056
2000-01-15  0.017217 1.0  0.649900 -0.896258
2000-01-16  0.691078 1.0  0.807450 -0.939302
2000-01-17  0.274506 1.0  0.582601 -0.902954
2000-01-18  0.330459 1.0  0.515707 -0.545268
2000-01-19  0.046756 1.0 -0.104334 -0.419799
2000-01-20 -0.328241 1.0 -0.650974 -0.777777
[20 rows x 4 columns]
```
14.2.8 Computing rolling pairwise covariances and correlations

**Warning:** Prior to version 0.20.0 if `pairwise=True` was passed, a `Panel` would be returned. This will now return a 2-level MultiIndexed DataFrame, see the `whatsnew` here.

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:

**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.

```python
In [80]: covs = df[['B','C','D']].rolling(window=50).cov(df[['A','B','C']], pairwise=True)

In [81]: covs.loc['2002-09-22':]

Out[81]:
   B       C       D
2002-09-22  1.367   8.677 -8.047
2002-09-23  3.067   0.866 -1.052
2002-09-24  0.866   7.739 -4.943

In [82]: correls = df.rolling(window=50).corr()

In [83]: correls.loc['2002-09-22':]

Out[83]:
   A       B       C       D
2002-09-22  1.000  0.186  0.745 -0.769
2002-09-23  0.186  1.000  0.177 -0.240
2002-09-24  0.745  0.177  1.000 -0.712
```

14.2. Window Functions 711
You can efficiently retrieve the time series of correlations between two columns by reshaping and indexing:

```python
In [84]: corr.unstack().loc[('A', 'C')].plot()
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x122699320>
```

![Graph showing time series of correlations between two columns](image)

## 14.3 Aggregation

Once the Rolling, Expanding or EWM objects have been created, several methods are available to perform multiple computations on the data. These operations are similar to the `aggregating API`, `groupby API`, and `resample API`.

```python
In [85]: dfa = pd.DataFrame(np.random.randn(1000, 3),
                      index=pd.date_range('1/1/2000', periods=1000),
                      columns=['A', 'B', 'C'])

In [86]: r = dfa.rolling(window=60, min_periods=1)

In [87]: r
Out[87]: Rolling [window=60, min_periods=1, center=False, axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a Series (or multiple Series) via standard
Pandas: powerful Python data analysis toolkit, Release 0.21.0

getitem.

```
In [88]: r.aggregate(np.sum)
Out[88]:
     A      B      C
2000-01-01 -0.289838 -0.370545 -1.284206
2000-01-02 -0.216612 -1.675528 -1.169415
2000-01-03  1.154661 -1.634017 -1.566620
2000-01-04  2.969393 -4.003274 -1.816179
2000-01-05  4.690630 -4.682017 -2.717209
2000-01-06  3.880630 -4.447700 -1.078947
2000-01-07  4.001957 -2.884072 -3.116903
          ...  ...  ...  ...
2002-09-20  2.652493 -10.528875  9.867805
2002-09-21  0.844497  -9.280944  9.522649
2002-09-22  2.860036  -9.270337  6.415245
2002-09-23  3.510163  -8.151439  5.177219
2002-09-24  6.524983 -10.168078  5.792639
2002-09-25  6.409626  -9.956226  5.704050
2002-09-26  5.093787  -7.074515  6.905823
[1000 rows x 3 columns]
```

```
In [89]: r["A"].aggregate(np.sum)

       A
2000-01-01 -0.289838
2000-01-02 -0.216612
2000-01-03  1.154661
2000-01-04  2.969393
2000-01-05  4.690630
2000-01-06  3.880630
2000-01-07  4.001957
          ...  ...
2002-09-20  2.652493
2002-09-21  0.844497
2002-09-22  2.860036
2002-09-23  3.510163
2002-09-24  6.524983
2002-09-25  6.409626
2002-09-26  5.093787
Freq: D, Name: A, Length: 1000, dtype: float64
```

```
In [90]: r[['A','B']].aggregate(np.sum)

     A      B
2000-01-01 -0.289838 -0.370545
2000-01-02 -0.216612 -1.675528
2000-01-03  1.154661 -1.634017
2000-01-04  2.969393 -4.003274
2000-01-05  4.690630 -4.682017
2000-01-06  3.880630 -4.447700
2000-01-07  4.001957 -2.884072
          ...  ...  ...
2002-09-20  2.652493 -10.528875
2002-09-21  0.844497  -9.280944
2002-09-22  2.860036  -9.270337
2002-09-23  3.510163  -8.151439
2002-09-24  6.524983 -10.168078
2002-09-25  6.409626  -9.956226
2002-09-26  5.093787  -7.074515
Freq: D, Name: [A, B], Length: 1000, dtype: float64
```

14.3. Aggregation
As you can see, the result of the aggregation will have the selected columns, or all columns if none are selected.

### 14.3.1 Applying multiple functions

With windowed `Series` you can also pass a list of functions to do aggregation with, outputting a DataFrame:

```
In [91]: r['A'].agg([np.sum, np.mean, np.std])
```

```
Out[91]:
          sum       mean      std
2002-09-24 -10.168078  6.524983 NaN
2002-09-25  -9.956226  6.409626 NaN
2002-09-26  -7.074515  5.093787 NaN
[1000 rows x 3 columns]
```

On a windowed DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [92]: r.agg([np.sum, np.mean])
```

```
Out[92]:
          A      B      C
          sum     sum     sum
2002-09-24 -10.168078 -9.956226 -7.074515
2002-09-25  -9.956226  -6.496266  -5.093787
2002-09-26  -7.074515   5.093787   6.095823
[1000 rows x 6 columns]
```
Passing a dict of functions has different behavior by default, see the next section.

### 14.3.2 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
In [93]: r.agg({'A' : np.sum,
          ....:         'B' : lambda x: np.std(x, ddof=1)})
```

```
Out[93]:
          A         B
2000-01-01 -0.289838  NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
2000-01-06  3.880630  0.966848
2000-01-07  4.001957  1.240137
...
2002-09-20  2.652493  1.114814
2002-09-21  0.844497  1.113220
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199
[1000 rows x 2 columns]
```

The function names can also be strings. In order for a string to be valid it must be implemented on the windowed object

```python
In [94]: r.agg({'A' : 'sum', 'B' : 'std'})
```

```
Out[94]:
          A         B
2000-01-01 -0.289838  NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
2000-01-06  3.880630  0.966848
2000-01-07  4.001957  1.240137
...
2002-09-20  2.652493  1.114814
2002-09-21  0.844497  1.113220
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199
[1000 rows x 2 columns]
```

Furthermore you can pass a nested dict to indicate different aggregations on different columns.

14.3. Aggregation
In [95]: `r.agg({'A' : ['sum','std'], 'B' : ['mean','std'] })
Out[95]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>std</td>
</tr>
<tr>
<td>2000-01-01</td>
<td>-0.289838</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.216612</td>
<td>0.256725</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>1.154661</td>
<td>0.873311</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>2.969393</td>
<td>1.009734</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>4.690630</td>
<td>0.977914</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>3.880630</td>
<td>1.128883</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>4.001957</td>
<td>1.049487</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2002-09-20</td>
<td>2.652493</td>
<td>1.164919</td>
</tr>
<tr>
<td>2002-09-21</td>
<td>0.844497</td>
<td>1.148231</td>
</tr>
<tr>
<td>2002-09-22</td>
<td>2.860036</td>
<td>1.132051</td>
</tr>
<tr>
<td>2002-09-23</td>
<td>3.510163</td>
<td>1.134296</td>
</tr>
<tr>
<td>2002-09-24</td>
<td>6.524983</td>
<td>1.144204</td>
</tr>
<tr>
<td>2002-09-25</td>
<td>6.409626</td>
<td>1.142913</td>
</tr>
<tr>
<td>2002-09-26</td>
<td>5.093787</td>
<td>1.151416</td>
</tr>
</tbody>
</table>

[1000 rows x 4 columns]

14.4 Expanding Windows

A common alternative to rolling statistics is to use an expanding window, which yields the value of the statistic with all the data available up to that point in time.

These follow a similar interface to .rolling, with the .expanding method returning an Expanding object.

As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

In [96]: `df.rolling(window=len(df), min_periods=1).mean()[:5]
Out[96]:
<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>0.314226</td>
<td>-0.001675</td>
<td>0.071823</td>
<td>0.892566</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.654522</td>
<td>-0.171495</td>
<td>0.179278</td>
<td>0.853361</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.708733</td>
<td>-0.064489</td>
<td>-0.238271</td>
<td>1.371111</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.987613</td>
<td>0.163472</td>
<td>-0.919693</td>
<td>1.566485</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>1.426971</td>
<td>0.288267</td>
<td>-1.358877</td>
<td>1.808650</td>
</tr>
</tbody>
</table>

In [97]: `df.expanding(min_periods=1).mean()[:5]

These have a similar set of methods to .rolling methods.
## 14.4.1 Method Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Number of non-null 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>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>std()</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>var()</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>skew()</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>kurt()</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>quantile()</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>apply()</td>
<td>Generic apply</td>
</tr>
<tr>
<td>cov()</td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td>corr()</td>
<td>Correlation (binary)</td>
</tr>
</tbody>
</table>

Aside from not having a window parameter, these functions have the same interfaces as their .rolling counterparts. Like above, the parameters they all accept are:

- min_periods: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No NaNs will be output once min_periods non-null data points have been seen.
- center: boolean, whether to set the labels at the center (default is False)

**Note:** The output of the .rolling and .expanding methods do not return a NaN if there are at least min_periods non-null values in the current window. For example,

```python
In [98]: sn = pd.Series([1, 2, np.nan, 3, np.nan, 4])
In [99]: sn
Out[99]:
0    1.0
1    2.0
2  NaN
3    3.0
4  NaN
5    4.0
dtype: float64
In [100]: sn.rolling(2).max()
Out[100]:
0   NaN
1    2.0
2  NaN
3  NaN
4  NaN
5  NaN
dtype: float64
In [101]: sn.rolling(2, min_periods=1).max()
```

### 14.4. Expanding Windows
In case of expanding functions, this differs from `cumsum()`, `cumprod()`, `cummax()`, and `cummin()`, which return NaN in the output wherever a NaN is encountered in the input. In order to match the output of `cumsum` with expanding, use `fillna()`:

```python
In [102]: sn.expanding().sum()
Out[102]:
0  1.0
1  3.0
2  3.0
3  6.0
4  6.0
5 10.0
dtype: float64
```

```python
In [103]: sn.cumsum()
Out[103]:
0  1.0
1  3.0
2  NaN
3  6.0
4  NaN
5 10.0
dtype: float64
```

```python
In [104]: sn.cumsum().fillna(method='ffill')
```

An expanding window statistic will be more stable (and less responsive) than its rolling window counterpart as the increasing window size decreases the relative impact of an individual data point. As an example, here is the `mean()` output for the previous time series dataset:

```python
In [105]: s.plot(style='k--')
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x1226fc5c0>
```

```python
In [106]: s.expanding().mean().plot(style='k')
```

```python
In [107]: <matplotlib.axes._subplots.AxesSubplot at 0x1226fc5c0>
```
14.5 Exponentially Weighted Windows

A related set of functions are exponentially weighted versions of several of the above statistics. A similar interface to .rolling and .expanding is accessed through the .ewm method to receive an EWM object. A number of expanding EW (exponentially weighted) methods are provided:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean()</td>
<td>EW moving average</td>
</tr>
<tr>
<td>var()</td>
<td>EW moving variance</td>
</tr>
<tr>
<td>std()</td>
<td>EW moving standard deviation</td>
</tr>
<tr>
<td>corr()</td>
<td>EW moving correlation</td>
</tr>
<tr>
<td>cov()</td>
<td>EW moving covariance</td>
</tr>
</tbody>
</table>

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 and \( y_t \) is the result.

The EW functions support two variants of exponential weights. The default, adjust=True, uses the weights \( w_i = (1 - \alpha)^i \) which gives

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + \ldots + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots + (1 - \alpha)^t} \]
When `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:

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + ...}{1 + (1 - \alpha) + (1 - \alpha)^2 + ...} \]

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of \( 1 - \alpha \) we have

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + ...}{1 - (1-\alpha)} \]
\[ = \left[ x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + ... \right] \alpha \]
\[ = \alpha x_t + [(1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + ...] \alpha \]
\[ = \alpha x_t + (1 - \alpha)[x_{t-1} + (1 - \alpha)x_{t-2} + ...] \alpha \]
\[ = \alpha x_t + (1 - \alpha)y_{t-1} \]

which shows the equivalence of the above two variants for infinite series. When `adjust=True` we have \( y_0 = x_0 \) and from the last representation above we have \( y_t = \alpha x_t + (1 - \alpha)y_{t-1} \), 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 since version 0.18.0 it has been 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:

\[
\alpha = \begin{cases} 
\frac{2}{s+1}, & \text{for span } s \geq 1 \\
\frac{1}{1+c}, & \text{for center of mass } c \geq 0 \\
1 - \exp^{-\log_{0.5}\frac{h}{c}}, & \text{for half-life } h > 0 
\end{cases}
\]

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.

Here is an example for a univariate time series:

```python
In [107]: s.plot(style='k--')
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x121e3bb70>

In [108]: s.ewm(span=20).mean().plot(style='k')
```

---

**pandas: powerful Python data analysis toolkit, Release 0.21.0**

Chapter 14. Computational tools
EWM has a `min_periods` argument, which has the same meaning it does for all the `.expanding` and `.rolling` methods: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window.

EWM 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 for further details.

14.5. Exponentially Weighted Windows
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. It differs from the MaskedArray approach of, for example, scikits.timeseries. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

See the cookbook for some advanced strategies

15.1 Missing data basics

15.1.1 When / why does data become missing?

Some might quibble over our usage of missing. By “missing” we simply mean NA (“not available”) or “not present for whatever reason”. Many data sets simply arrive with missing data, either because it exists and was not collected or it never existed. For example, in a collection of financial time series, some of the time series might start on different dates. Thus, values prior to the start date would generally be marked as missing.

In pandas, one of the most common ways that missing data is introduced into a data set is by reindexing. For example

```
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 -0.166778  0.501113 -0.355322    bar  False
   c -0.337890  0.580967  0.983801    bar  False
   e  0.057802  0.761948 -0.712964    bar   True
   f -0.443160 -0.974602  1.047704    bar  False
   h -0.717852 -1.053898 -0.019369    bar  False

In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
Out[6]:
```
15.1.2 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.

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]:
a  -0.166778
b   NaN
... 
h  -0.717852
Name: one, dtype: float64
In [8]: pd.isna(df2['one'])
→
a  False
b  True
... 
h  False
Name: one, dtype: bool
In [9]: df2['four'].notna()
→
a  True
b  False
... 
c  True
```
In [10]: df2.isna()

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>b</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>c</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>d</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>e</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>f</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>g</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>h</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Name: four, dtype: bool

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
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]:
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
</tbody>
</table>

Name: one, dtype: bool

15.2 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 intercompatibility between NaT and NaN.

In [14]: df2 = df.copy()

In [15]: df2['timestamp'] = pd.Timestamp('20120101')

In [16]: df2
15.3 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:

```
In [20]: s = pd.Series([1, 2, 3])
In [21]: s.loc[0] = None
In [22]: s
Out[22]:
   0   NaN
   1  2.0
   2  3.0
dtype: float64
```

Likewise, datetime containers will always use NaT.

For object containers, pandas will use the value given:

```
In [23]: s = pd.Series(["a", "b", "c"])
In [24]: s.loc[0] = None
In [25]: s.loc[1] = np.nan
```
15.4 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

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 NA
- Methods like cumsum and cumprod ignore NA values, but preserve them in the resulting arrays
```python
In [31]: df['one'].sum()
   → -0.3853826528461409

In [32]: df.mean(1)
   →
   a   0.072895  
c   0.782384  
e   0.035595   
f  -0.123353   
h  -0.536633  
dtype: float64

In [33]: df.cumsum()
   →
   one    two    three
   a  NaN   0.501113 -0.355322
   c  NaN   1.082080  0.628479
   e  0.057802 1.844028 -0.084485
   f -0.385358 0.869426  0.963219
   h  NaN  -0.184472  0.943850
```

15.4.1 Sum/Prod of Empties/Nans

**Warning:** This behavior is now standard as of v0.21.0; previously sum/prod would give different results if the `bottleneck` package was installed. See the [here](#).

With `sum` or `prod` on an empty or all-NaNSeries, or columns of a DataFrame, the result will be all-NaN.

```python
In [34]: s = pd.Series([np.nan])

In [35]: s.sum()
Out[35]: nan

Summing of an empty Series
```

```python
In [36]: pd.Series([]).sum()
Out[36]: nan
```

**Warning:** These behaviors differ from the default in `numpy` where an empty sum returns zero.

```python
In [37]: np.nansum(np.array([np.nan]))
Out[37]: 0.0

In [38]: np.nansum(np.array([]))
   \n   Out[38]: 0.0
```
15.4.2 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example:

```
In [39]: df
Out[39]:
   one     two      three
a  NaN    0.501113 -0.355322
b  NaN    0.580967  0.983801
c  0.057802  0.761948  0.712964
d -0.443160 -0.974602  1.047704
e  0.057802  0.761948  0.712964
f  NaN   -1.053898 -0.019369
```

```
In [40]: df.groupby('one').mean()
```

See the groupby section here for more information.

15.5 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

15.5.1 Filling missing values: fillna

The `fillna` function can “fill in” NA values with non-NA data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [41]: df2
Out[41]:
   one     two      three     four    five      timestamp
a  NaN    0.501113 -0.355322  bar  False        NaT
b  NaN    0.580967   0.983801  bar  False        NaT
c  0.057802  0.761948 -0.712964  bar   True  00:00:00
f -0.443160 -0.974602  1.047704  bar  False        NaT
h  NaN   -1.053898 -0.019369  bar  False        NaT
```

```
In [42]: df2.fillna(0)
```

```
In [43]: df2['four'].fillna('missing')
```

15.5. Cleaning / filling missing data
Fill gaps forward or backward

Using the same filling arguments as reindexing, we can propagate non-NA values forward or backward:

```
In [44]: df
Out[44]:
      one  two    three
a   NaN 0.501113 -0.355322
c  NaN  0.580967  0.983801
e -0.443160 -0.974602  1.047704
f  NaN -1.053898 -0.019369
h  NaN -1.053898 -0.019369

In [45]: df.fillna(method='pad')
Out[45]:
      one  two    three
a   NaN 0.501113 -0.355322
c  NaN  0.580967  0.983801
e   NaN  0.580967  0.983801
f   NaN  0.580967  0.983801
h -0.443160 -1.053898 -0.019369
```

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 [46]: df
Out[46]:
      one  two    three
a   NaN 0.501113 -0.355322
c  NaN  0.580967  0.983801
e  NaN  NaN   NaN
f  NaN  NaN   NaN
h -1.053898 -0.019369

In [47]: df.fillna(method='pad', limit=1)
Out[47]:
      one  two    three
a   NaN 0.501113 -0.355322
c  NaN  0.580967  0.983801
e  NaN  0.580967  0.983801
f  NaN  NaN   NaN
h -1.053898 -0.019369
```

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.

The `ffill()` function is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`.

### 15.5.2 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.

```python
In [48]: dff = pd.DataFrame(np.random.randn(10,3), columns=list('ABC'))
In [49]: dff.iloc[3:5,0] = np.nan
In [50]: dff.iloc[4:6,1] = np.nan
In [51]: dff.iloc[5:8,2] = np.nan
In [52]: dff
Out[52]:
   A      B      C
0 0.758887 2.340598 0.219039
1 -1.235583 0.031785 0.701683
2 -1.557016 -0.636986 -1.238610
3 NaN -1.002278 0.654052
4 NaN NaN 1.053999
5 0.651981 NaN NaN
6 0.109001 -0.533294 NaN
7 -1.037831 -1.150016 NaN
8 -0.687693 1.921056 -0.121113
9 -0.258742 -0.706329 0.402547
```

```python
In [53]: dff.fillna(dff.mean())
Out[53]:
   A      B      C
0 0.758887 2.340598 0.219039
1 -1.235583 0.031785 0.701683
2 -1.557016 -0.636986 -1.238610
3 -0.407125 -1.002278 0.654052
4 -0.407125 0.033067 1.053999
5 0.651981 0.033067 0.238800
6 0.109001 -0.533294 0.238800
7 -1.037831 -1.150016 0.238800
8 -0.687693 1.921056 -0.121113
9 -0.258742 -0.706329 0.402547
```

```python
In [54]: dff.fillna(dff.mean() \['B': 'C'\])
Out[54]:
   A      B      C
0 0.758887 2.340598 0.219039
1 -1.235583 0.031785 0.701683
2 -1.557016 -0.636986 -1.238610
3 NaN -1.002278 0.654052
4 NaN 0.033067 1.053999
```
Same result as above, but is aligning the ‘fill’ value which is a Series in this case.

```
In [55]: dff.where(pd.notna(dff), dff.mean(), axis='columns')
Out[55]:
   A    B    C
0  0.758887  2.340598  0.219039
1 -1.235583  0.031785  0.701683
2 -1.557016 -0.636986 -1.238610
3 -0.407125 -1.002278  0.654052
4  0.651981  0.033067  0.238800
5  0.109001 -0.533294  0.238800
6 -1.037831 -1.150016  0.238800
7 -0.687693  1.921056 -0.121113
8 -0.258742 -0.706329  0.402547
9  0.651981  0.033067  0.238800
```

15.5.3 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 the `dropna` method:

```
In [56]: df
Out[56]:
   one    two    three
  a  NaN  0.501113 -0.355322
  c  NaN  0.580967  0.983801
  e  NaN  0.000000  0.000000
  f  NaN  0.000000  0.000000
  h  NaN -1.053898 -0.019369

In [57]: df.dropna(axis=0)
→ Empty DataFrame
Columns: [one, two, three]
Index: []

In [58]: df.dropna(axis=1)
→
   two    three
  a  0.501113 -0.355322
  c  0.580967  0.983801
  e  0.000000  0.000000
  f  0.000000  0.000000
  h -1.053898 -0.019369

In [59]: df['one'].dropna()
→ Series([], Name: one, dtype: float64)
```
Series.dropna is a simpler method as it only has one axis to consider. DataFrame.dropna has considerably more options than Series.dropna, which can be examined in the API.

### 15.5.4 Interpolation

New in version 0.17.0: The limit_direction keyword argument was added. Both Series and DataFrame objects have an interpolate method that, by default, performs linear interpolation at missing datapoints.

```python
In [60]: ts
Out[60]:
2000-01-31  0.469112
2000-02-29   NaN
2000-03-31   NaN
2000-04-28   NaN
2000-05-31   NaN
2000-06-30   NaN
2000-07-31   NaN
...  
2007-10-31  -3.305259
2007-11-30  -5.485119
2007-12-31  -6.854968
2008-01-31  -7.809176
2008-02-29  -6.346480
2008-03-31  -8.089641
2008-04-30  -8.916232
Freq: BM, Length: 100, dtype: float64

In [61]: ts.count()
Out[61]:
61

In [62]: ts.interpolate().count()
Out[62]:
100

In [63]: ts.interpolate().plot()
Out[63]:
<matplotlib.axes._subplots.AxesSubplot at 0x13130c160>
```
Index aware interpolation is available via the `method` keyword:

```
In [64]: ts2
Out[64]:
2000-01-31    0.469112
2000-02-29     NaN
2002-07-31   -5.689738
2005-01-31     NaN
2008-04-30  -8.916232
dtype: float64

In [65]: ts2.interpolate()

2000-01-31    0.469112
2000-02-29   -2.610313
2002-07-31   -5.689738
2005-01-31  -7.302985
2008-04-30  -8.916232
dtype: float64

In [66]: ts2.interpolate(method='time')
```

```
2000-01-31    0.469112
2000-02-29   0.273272
2002-07-31  -5.689738
2005-01-31  -7.095568
2008-04-30  -8.916232
dtype: float64
```
For a floating-point index, use `method='values'`:

```python
In [67]: ser
Out[67]:
0.0 0.0
1.0 NaN
10.0 10.0
dtype: float64

In [68]: ser.interpolate()
Out[68]:
0.0 0.0
1.0 5.0
10.0 10.0
dtype: float64

In [69]: ser.interpolate(method='values')
Out[69]:
0.0 0.0
1.0 1.0
10.0 10.0
dtype: float64
```

You can also interpolate with a DataFrame:

```python
In [70]: 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]})

In [71]: df
Out[71]:
   A   B
0  1.0  0.25
1  2.1   NaN
2   NaN   NaN
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40

In [72]: df.interpolate()
Out[72]:
   A   B
0  1.0  0.25
1  2.1  1.00
2  3.4  2.75
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40
```

The `method` argument gives access to fancier interpolation methods. If you have `scipy` installed, you can set 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, use method='akima'.

**Warning:** These methods require scipy.

```
In [73]: df.interpolate(method='barycentric')
Out[73]:
   A     B
0  1.00  0.250
1  2.10 -7.660
2  3.53 -4.515
3  4.70  4.000
4  5.60 12.200
5  6.80 14.400
```

```
In [74]: df.interpolate(method='pchip')
Out[74]:
   A     B
0  1.00000  0.250000
1  2.10000  0.672808
2  3.43454  1.928950
3  4.70000  4.000000
4  5.60000 12.200000
5  6.80000 14.400000
```

```
In [75]: df.interpolate(method='akima')
Out[75]:
   A     B
0  1.000000  0.250000
1  2.100000 -0.873316
2  3.406667  0.320034
3  4.700000  4.000000
4  5.600000 12.200000
5  6.800000 14.400000
```

When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [76]: df.interpolate(method='spline', order=2)
Out[76]:
   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
```

```
In [77]: df.interpolate(method='polynomial', order=2)
Out[77]:
   A     B
0  1.000000  0.250000
```

736 Chapter 15. Working with missing data
Compare several methods:

```python
In [78]: np.random.seed(2)

In [79]: ser = pd.Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))

In [80]: bad = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])

In [81]: ser[bad] = np.nan

In [82]: methods = ['linear', 'quadratic', 'cubic']

In [83]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})

In [84]: df.plot()
```

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 [85]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# reindex at new_index
```
**15.5.4.1 Interpolation Limits**

Like other pandas fill methods, `interpolate` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive interpolations, keeping NaN values for interpolations that are too far from the last valid observation:

```python
In [89]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13])
In [90]: ser.interpolate(limit=2)
Out[90]:
0    NaN
1    NaN
2     5.0
3     7.0
4     9.0
5    NaN
6    13.0
dtype: float64
```

By default, `limit` applies in a forward direction, so that only NaN values after a non-NaN value can be filled. If you provide 'backward' or 'both' for the `limit_direction` keyword argument, you can fill NaN values before non-NaN values, or both before and after non-NaN values, respectively:

```python
In [91]: ser.interpolate(limit=1)  # limit_direction == 'forward'
Out[91]:
0    NaN
1    NaN
2     5.0
3     7.0
4    NaN
5    NaN
6    13.0
dtype: float64
```

```python
In [92]: ser.interpolate(limit=1, limit_direction='backward')
Out[92]:
0    NaN
1     5.0
```

738 Chapter 15. Working with missing data
2  5.0
3  NaN
4  NaN
5  11.0
6  13.0
dtype: float64

In [93]: ser.interpolate(limit=1, limit_direction='both')

Out[93]:
   0   NaN
   1   5.0
   2   5.0
   3   7.0
   4   NaN
   5   11.0
   6   13.0
dtype: float64

### 15.5.5 Replacing Generic Values

Often times we want to replace arbitrary values with other values. The `replace` method in Series/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:

In [94]: ser = pd.Series([0., 1., 2., 3., 4.])

In [95]: ser.replace(0, 5)
Out[95]:
   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:

In [96]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[96]:
   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 [97]: ser.replace({0: 10, 1: 100})
Out[97]:
   0   10.0
   1  100.0
   2   2.0
   3   3.0

### 15.5. Cleaning / filling missing data
For a DataFrame, you can specify individual values by column:

```python
In [98]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})
In [99]: df.replace({'a': 0, 'b': 5}, 100)
Out[99]:
   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:

```python
In [100]: ser.replace([1, 2, 3], method='pad')
Out[100]:
0 0.0
1 0.0
2 0.0
3 0.0
4 4.0
dtype: float64
```

### 15.5.6 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)

```python
In [101]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}
In [102]: df = pd.DataFrame(d)
In [103]: df.replace('.', np.nan)
Out[103]:
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)

```python
In [104]: df.replace(r'\s*\s*', np.nan, regex=True)
Out[104]:
a  b  c
0  0  a  a
1  1  b  b
```
Replace a few different values (list -> list)

```python
In [105]: df.replace(['a', '.'], ['b', np.nan])
Out[105]:
       a    b    c
0  0.0  0.0  0.0
1  1.1  1.1  1.1
2  NaN  NaN  NaN
3  NaN  NaN  d
```

list of regex -> list of regex

```python
In [106]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
Out[106]:
       a    b    c
0  {stuff {stuff
1  1.1  1.1  1.1
2  dot  NaN  NaN
3  dot  d   d
```

Only search in column 'b' (dict -> dict)

```python
In [107]: df.replace({'b': '.'}, {'b': np.nan})
Out[107]:
       a    b    c
0  0.0  0.0  0.0
1  1.1  1.1  1.1
2  NaN  NaN  NaN
3  NaN  NaN  d
```

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict)

```python
In [108]: df.replace({'b': {r'\s*\.:\s*': np.nan}}, regex=True)
Out[108]:
       a    b    c
0  0.0  0.0  0.0
1  1.1  1.1  1.1
2  NaN  NaN  NaN
3  NaN  NaN  d
```

You can pass nested dictionaries of regular expressions that use `regex=True`

```python
In [109]: df.replace({'b': {'b': r'\s*\.:\s*'}}, regex=True)
Out[109]:
       a    b    c
0  0.0  0.0  0.0
1  1.1  1.1  1.1
2  NaN  NaN  NaN
3  NaN  NaN  d
```

or you can pass the nested dictionary like so

```python
In [110]: df.replace(regex={'b': {r'\s*\.:\s*': np.nan}})
Out[110]:
       a    b    c
```

15.5. Cleaning / filling missing data
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 [111]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
Out[111]:
   a   b   c
0  a   a   a
1  b   b   b
2  .ty  NaN
3  .ty  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 [112]: df.replace([r'\s*\.(\.)\s*', r'a|b'], np.nan, regex=True)
Out[112]:
   a   b   c
0  NaN NaN
1  NaN NaN
2  NaN NaN
3  NaN  d
```

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 [113]: df.replace(regex=[r'\s*\.(\.)\s*', r'a|b'], value=np.nan)
Out[113]:
   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.

### 15.5.7 Numeric Replacement

Similar to `DataFrame.fillna`

```
In [114]: df = pd.DataFrame(np.random.randn(10, 2))
In [115]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5
In [116]: df.replace(1.5, np.nan)
Out[116]:
   a   b
0  NaN NaN
1  NaN NaN
2  NaN NaN
3  NaN  d
```

---

**Chapter 15. Working with missing data**

---
Replacing more than one value via lists works as well

```
In [117]: df00 = df.values[0, 0]

In [118]: df.replace([1.5, df00], [np.nan, 'a'])
Out[118]:
   0  1
0  a -1.02141
1  0.432396 -0.32358
2  0.423825  0.79918
3  1.262615  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
```

You can also operate on the DataFrame in place

```
In [120]: df.replace(1.5, np.nan, inplace=True)
```

```
Warning: When replacing multiple bool or datetime64 objects, the first argument to replace (to_replace) must match the type of the value being replaced type. For example.
```
```
s = pd.Series([True, False, True])
s.replace({'a string': 'new value', True: False}) # raises

```

```
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

will raise a `TypeError` because one of the dict keys is not of the correct type for replacement.

However, when replacing a single object such as,

```
In [121]: s = pd.Series([True, False, True])
```

```
In [122]: s.replace('a string', 'another string')
Out[122]:
   0    True
   1    False
   2    True
dtype: bool
```

```
15.5. Cleaning / filling missing data
```
the original \texttt{NDFrame} object will be returned untouched. We’re working on unifying this API, but for backwards compatibility reasons we cannot break the latter behavior. See GH6354 for more details.

### 15.6 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 reindexing will cause missing data to be introduced into, say, a Series or DataFrame. Here they are:

<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:

```python
In [123]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [124]: s > 0
Out[124]:
0   True
2   True
4   True
6   True
7   True
dtype: bool

In [125]: (s > 0).dtype
Out[125]: dtype('bool')

In [126]: crit = (s > 0).reindex(list(range(8)))

In [127]: crit
Out[127]:
0   True
1   NaN
2   True
3   NaN
4   True
5   NaN
6   True
7   True
dtype: object

In [128]: crit.dtype
Out[128]: dtype('O')
```

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 [129]: reindexed = s.reindex(list(range(8))).fillna(0)

In [130]: reindexed[crit]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-130-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]

~/Envs/pandas-dev/lib/python3.6/site-packages/pandas/pandas/core/series.py in __

_getitem__(self, key)
    659         key = list(key)
    660
--> 661     if com.is_bool_indexer(key):
    662         key = check_bool_indexer(self.index, key)
    663

~/Envs/pandas-dev/lib/python3.6/site-packages/pandas/pandas/core/common.py in is_bool_

__indexer(key)
    189             if not lib.is_bool_array(key):
    190                 if isna(key).any():
--> 191                     raise ValueError('cannot index with vector containing '
    192                         'NA / NaN values')
    193             return False

ValueError: cannot index with vector containing NA / NaN values

However, these can be filled in using `fillna` and it will work fine:

In [131]: reindexed[crit.fillna(False)]
Out[131]:
    0   0.126504
    2   0.696198
    4   0.697416
    6   0.601516
    7   0.003659
dtype: float64

In [132]: reindexed[crit.fillna(True)]
Out[132]:
    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
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

Of these, the split step is the most straightforward. In fact, in many situations you may wish to split the data set into groups and do something with those groups yourself. In the apply step, we might wish to one of the following:

- **Aggregation**: computing a summary statistic (or statistics) about each group. Some examples:
  - Compute group sums or means
  - Compute group sizes / counts
- **Transformation**: perform some group-specific computations and return a like-indexed. Some examples:
  - Standardizing data (zscore) within 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:
  - Discarding data that belongs to groups with only a few members
  - Filtering 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:

```sql
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
16.1 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 do the following:

```python
# default is axis=0
>>> grouped = obj.groupby(key)
>>> grouped = obj.groupby(key, axis=1)
>>> grouped = obj.groupby([key1, key2])
```

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 a column to be used to group. Of course \( df \cdot \text{groupby}('A') \) is just syntactic sugar for \( df \cdot \text{groupby}(df['A']) \), but it makes life simpler
- For DataFrame objects, a string indicating an index level to be used to group.
- 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 [1]: 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 [2]: df
Out[2]:
       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
```

We could naturally group by either the A or B columns or both:

```
In [3]: grouped = df.groupby('A')
In [4]: grouped = df.groupby(['A', 'B'])
```
These will split the DataFrame on its index (rows). We could also split by the columns:

```python
In [5]: def get_letter_type(letter):
   ...:     if letter.lower() in 'aeiou':
   ...:         return 'vowel'
   ...:     else:
   ...:         return 'consonant'
   ...:
In [6]: 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 [7]: lst = [1, 2, 3, 1, 2, 3]
In [8]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)
In [9]: grouped = s.groupby(level=0)
In [10]: grouped.first()
Out[10]:
   1  1
   2  2
   3  3
dtype: int64
In [11]: grouped.last()
Out[11]:
   1  10
   2  20
   3  30
dtype: int64
In [12]: grouped.sum()
Out[12]:
   1  11
   2  22
   3  33
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.

### 16.1.1 GroupBy sorting

By default the group keys are sorted during the groupby operation. You may however pass sort=False for potential speedups.
In [13]: df2 = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [14]: df2.groupby(['X']).sum()
Out[14]:
X  Y
A 7
B 3

In [15]: df2.groupby(['X'], sort=False).sum()
Out[15]:
X  Y
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:

In [16]: df3 = pd.DataFrame({'X': ['A', 'B', 'A', 'B'], 'Y': [1, 4, 3, 2]})

In [17]: df3.groupby(['X']).get_group('A')
Out[17]:
X  Y
0 A 1
2 A 3

In [18]: df3.groupby(['X']).get_group('B')
Out[18]:
X  Y
1 B 4
3 B 2

16.1.2 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:

In [19]: df.groupby('A').groups
Out[19]:
{'bar': Int64Index([1, 3, 5], dtype='int64'),
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}

In [20]: df.groupby(get_letter_type, axis=1).groups
Out[20]:
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}

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:

In [21]: grouped = df.groupby(['A', 'B'])

In [22]: grouped.groups
Out[22]:
```python
In [23]: len(grouped)
Out[23]:

In [24]: df
Out[24]:

In [25]: gb = df.groupby('gender')
In [26]: gb.<TAB>
gb.agg gb.boxplot gb.cummin gb.describe gb.filter gb.get_groups
gb.aggregate gb.clast gb.median gb.ngroups gb.plot gb.rank
gb.std gb.transform
gb.hist gb.count gb.cumprod gb.dtype gb.first gb.groups
gb.sum gb.var
gb.apply gb.cummax gb.cumsum gb.fillna gb.gender gb.head
gb.indices gb.mean gb.name gb.ohlc gb.quantile gb.size
gb.tail gb.weight

16.1.3 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 [27]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
In [28]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])
In [29]: s = pd.Series(np.random.randn(8), index=index)
In [30]: s
Out[30]:

16.1. Splitting an object into groups 751
We can then group by one of the levels in `s`.

```
In [31]: grouped = s.groupby(level=0)

In [32]: grouped.sum()
Out[32]:
first  
bar   one   -0.962232
      two   -0.962232
baz   one   1.237723
      two   1.237723
foo   one   0.785980
      two   0.785980
qux   one   1.911055
      two   1.911055
dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```
In [33]: s.groupby(level='second').sum()
Out[33]:
second  
one     0.980950
      two   1.991575
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

```
In [34]: s.sum(level='second')
Out[34]:
second  
one     0.980950
      two   1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```
In [35]: s
Out[35]:
first  
bar   doo  one   -1.131345
      two   -0.089329
baz   bee  one   0.337863
      two   -0.945867
foo   bop  one  -0.932132
      two    1.956030
qux   bop  one   0.017587
      two  -0.016692
dtype: float64
```
16.1.4 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.

The following example groups `df` by the `second` index level and the `A` column.
**16.1.5 DataFrame column selection in GroupBy**

Once you have created the GroupBy object from a DataFrame, for example, 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 [45]: grouped = df.groupby(['A'])
In [46]: grouped_C = grouped['C']
In [47]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```python
In [48]: df['C'].groupby(df['A'])
Out[48]: <pandas.core.groupby.SeriesGroupBy object at 0x11513a550>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.
16.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to itertools.groupby:

```python
In [49]: grouped = df.groupby('A')

In [50]: 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:

```python
In [51]: 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
   6 foo one  -0.408530  0.268520
('foo', 'three')
    A    B    C    D
   7 foo three  -0.862495  0.024580
('foo', 'two')
    A    B    C    D
   2 foo two  -1.143704  1.627081
   4 foo two  1.193555  -0.441652
```

It's standard Python-fu but remember you can unpack the tuple in the for loop statement if you wish: `for (k1, k2), group in grouped:`.
16.3 Selecting a group

A single group can be selected using `GroupBy.get_group()`:

```python
In [52]: grouped.get_group('bar')
Out[52]:
       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 [53]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[53]:
       A     B     C      D
1  bar one 0.254161 1.511763
```

16.4 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 functions API`, and `resample API`.

An obvious one is aggregation via the `aggregate` or equivalently `agg` method:

```python
In [54]: grouped = df.groupby('A')
In [55]: grouped.aggregate(np.sum)
Out[55]:
          C     D
A
bar  0.392940 1.732707
foo -1.796421 2.824590

In [56]: grouped = df.groupby(['A', 'B'])
In [57]: grouped.aggregate(np.sum)
Out[57]:
     C     D
A B
bar one 0.254161 1.511763
     three 0.215897 -0.990582
     two -0.077118 1.211526
foo one -0.983776 1.614581
     three -0.862495 0.024580
     two 0.049851 1.185429
```

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 `MultiIndex` by default, though this can be changed by using the `as_index` option:

```python
In [58]: grouped = df.groupby(['A', 'B'], as_index=False)
In [59]: grouped.aggregate(np.sum)
Out[59]:
```

756 Chapter 16. Group By: split-apply-combine
In [60]: df.groupby('A', as_index=False).sum()

Out[60]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>0.392940</td>
<td>1.732707</td>
</tr>
<tr>
<td>1</td>
<td>foo</td>
<td>-1.796421</td>
<td>2.824590</td>
</tr>
</tbody>
</table>

Note that you could use the `reset_index` DataFrame function to achieve the same result as the column names are stored in the resulting `MultiIndex`:

In [61]: df.groupby(['A', 'B']).sum().reset_index()

Out[61]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>one</td>
<td>0.254161</td>
<td>1.511763</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>three</td>
<td>0.215897</td>
<td>-0.990582</td>
</tr>
<tr>
<td>2</td>
<td>bar</td>
<td>two</td>
<td>-0.077118</td>
<td>1.211526</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>one</td>
<td>-0.983776</td>
<td>1.614581</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>three</td>
<td>-0.862495</td>
<td>0.024580</td>
</tr>
<tr>
<td>5</td>
<td>foo</td>
<td>two</td>
<td>0.049851</td>
<td>1.185429</td>
</tr>
</tbody>
</table>

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a Series whose index are the group names and whose values are the sizes of each group.

In [62]: grouped.size()

Out[62]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>three</td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
<tr>
<td>foo</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>three</td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
</tbody>
</table>

dtype: int64

In [63]: grouped.describe()

Out[63]:

<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>C</td>
<td>0.10</td>
<td>0.254161 NaN</td>
<td>0.254161</td>
<td>0.254161</td>
<td>0.254161</td>
<td>0.254161</td>
<td>0.254161</td>
<td>0.254161</td>
</tr>
<tr>
<td></td>
<td>1.10</td>
<td>NaN</td>
<td>0.215897 NaN</td>
<td>0.215897</td>
<td>0.215897</td>
<td>0.215897</td>
<td>0.215897</td>
<td>0.215897</td>
</tr>
<tr>
<td></td>
<td>2.10</td>
<td>-0.077118 NaN</td>
<td>-0.077118</td>
<td>-0.077118</td>
<td>-0.077118</td>
<td>-0.077118</td>
<td>-0.077118</td>
<td>-0.077118</td>
</tr>
<tr>
<td></td>
<td>3.20</td>
<td>-0.491888 0.117887</td>
<td>-0.575247</td>
<td>-0.533567</td>
<td>-0.491888</td>
<td>-0.450209</td>
<td>-0.408530</td>
<td>-0.408530</td>
</tr>
<tr>
<td></td>
<td>4.10</td>
<td>-0.862495 NaN</td>
<td>-0.862495</td>
<td>-0.862495</td>
<td>-0.862495</td>
<td>-0.862495</td>
<td>-0.862495</td>
<td>-0.862495</td>
</tr>
<tr>
<td></td>
<td>5.20</td>
<td>0.024925 1.652692</td>
<td>-1.143704</td>
<td>-0.559389</td>
<td>0.024925</td>
<td>0.609240</td>
<td>1.193555</td>
<td>1.193555</td>
</tr>
</tbody>
</table>

<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>D</td>
<td>0.10</td>
<td>1.511763 NaN</td>
<td>1.511763</td>
<td>1.511763</td>
<td>1.511763</td>
<td>1.511763</td>
<td>1.511763</td>
<td>1.511763</td>
</tr>
<tr>
<td></td>
<td>1.10</td>
<td>NaN</td>
<td>-0.990582 NaN</td>
<td>-0.990582</td>
<td>-0.990582</td>
<td>-0.990582</td>
<td>-0.990582</td>
<td>-0.990582</td>
</tr>
</tbody>
</table>

16.4. Aggregation
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 ones that reduce the dimension of the returned objects, for example: mean, sum, size, count, std, var, sem, describe, first, last, nth, min, max. This is what happens when you do for example DataFrame.sum() and get back a Series.

nth can act as a reducer or a filter, see here

### 16.4.1 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 [64]: grouped = df.groupby('A')

In [65]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[65]:
         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 [66]: grouped.agg([np.sum, np.mean, np.std])
Out[66]:
      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 [67]: (grouped['C'].agg([np.sum, np.mean, np.std])
        ....: .rename(columns={'sum': 'foo',
        ....:                     'mean': 'bar',
        ....:                     'std': 'baz'}))
        ....: )
    Out[67]:
      foo  bar  baz
    A     
   bar  0.392940  0.130980  0.181231
  foo -1.796421 -0.359284  0.912265
```
For a grouped DataFrame, you can rename in a similar manner:

```python
In [68]: (grouped.agg([np.sum, np.mean, np.std])
    ....: .rename(columns={'sum': 'foo',
    ....:                      'mean': 'bar',
    ....:                      'std': 'baz'})
    ....:)
```

```text
Out[68]:
   C   D
  foo bar baz foo bar baz
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
```

### 16.4.2 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
In [69]: grouped.agg({'C' : np.sum,
    ....:                  'D' : lambda x: np.std(x, ddof=1)})
```

```text
Out[69]:
   C   D
  A  bar 0.392940 1.366330
  foo -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:

```python
In [70]: grouped.agg({'C' : 'sum', 'D' : 'std'})
```

```text
Out[70]:
   C   D
  A  bar 0.392940 1.366330
  foo -1.796421 0.884785
```

**Note:** If you pass a dict to `aggregate`, the ordering of the output columns is non-deterministic. If you want to be sure the output columns will be in a specific order, you can use an `OrderedDict`. Compare the output of the following two commands:

```python
In [71]: grouped.agg({'D': 'std', 'C': 'mean'})
```

```text
Out[71]:
    D   C
A  bar 1.366330 0.130980
  foo 0.884785 -0.359284
```

```python
In [72]: grouped.agg(OrderedDict([('D', 'std'), ('C', 'mean')]))
```

```text
Out[72]:
---
    D   C
A
```
16.4.3 Cython-optimized aggregation functions

Some common aggregations, currently only sum, mean, std, and sem, have optimized Cython implementations:

```python
In [73]: df.groupby('A').sum()
Out[73]:
     C     D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [74]: df.groupby(['A', 'B']).mean()
     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).

16.5 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.

For example, suppose we wished to standardize the data within each group:

```python
In [75]: index = pd.date_range('10/1/1999', periods=1100)
In [76]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
In [77]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()
```
We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
# Original Data
In [83]: grouped = ts.groupby(key)

In [84]: grouped.mean()
Out[84]:
2000 0.442441
2001 0.526246
2002 0.459365
dtype: float64

In [85]: grouped.std()
Out[85]:
2000 0.131752
2001 0.210945
2002 0.128753
dtype: float64

# Transformed Data
In [86]: grouped_trans = transformed.groupby(key)

In [87]: grouped_trans.mean()
Out[87]:
2000 1.168208e-15
2001 1.454544e-15
2002 1.726657e-15
dtype: float64

In [88]: grouped_trans.std()
Out[88]:
```
We can also visually compare the original and transformed data sets.

```python
In [89]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})
In [90]: compare.plot()
Out[90]: <matplotlib.axes._subplots.AxesSubplot at 0x12772ff98>
```

Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```python
In [91]: data_range = lambda x: x.max() - x.min()
In [92]: ts.groupby(key).transform(data_range)
Out[92]:
   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
   2000-01-13    0.623893
   2000-01-14    0.623893
   ...          ...
   2002-09-28    0.558275
   2002-09-29    0.558275
   2002-09-30    0.558275
```
Alternatively the built-in methods can be could be used to produce the same outputs

```
In [93]: ts.groupby(key).transform('max') - ts.groupby(key).transform('min')
Out[93]:
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
2000-01-13 0.623893
2000-01-14 0.623893
...  
2002-09-28 0.558275
2002-09-29 0.558275
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
```

Another common data transform is to replace missing data with the group mean.

```
In [94]: data_df
Out[94]:
   A       B       C
0  1.539708 -1.166480  0.533026
1  1.302092 -0.505754    NaN
2 -0.371983  1.104803 -0.651520
3 -1.309622  1.118697 -1.161657
4  0.815643  0.367816 -0.469478
5 -0.030651  1.376106 -0.645129
6   ... ... ... ... ...
993  0.012359  0.554602 -1.976159
994  0.042312 -1.628835  1.013822
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 [95]: countries = np.array(['US', 'UK', 'GR', 'JP'])
In [96]: key = countries[np.random.randint(0, 4, 1000)]
In [97]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [98]: grouped.count()
```

16.5. Transformation
We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.

```python
In [101]: grouped_trans = transformed.groupby(key)

In [102]: grouped.mean()  # original group means

Out[102]:
    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 [103]: grouped_trans.mean()  # transformation did not change group means

    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 [104]: grouped.count()  # original has some missing data points

    A     B     C
--|--|--|---|
GR 209  217  189
JP 240  255  217
UK 216  231  193
US 239  250  217

In [105]: grouped_trans.count()  # counts after transformation

    A     B     C
--|--|--|---|
GR 228  228  228
JP 267  267  267
UK 247  247  247
US 258  258  258

In [106]: grouped_trans.size()  # Verify non-NA count equals group size

    GR  JP  UK  US
--|--|--|--|
  228  267  247  258
Note: Some functions when applied to a groupby object will automatically transform the input, 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 [107]: grouped.ffill()
Out[107]:
   A      B      C
0  1.539708 -1.166480  0.533026
1  1.302092 -0.505754  0.533026
2 -0.371983  1.104803 -0.651520
3 -1.309622  1.186970 -1.161657
4 -1.924296  0.396437  0.812436
5  0.815643  0.367816 -0.469478
6 -0.030651  1.376106 -0.645129
   ..   ...   ...
993 0.012359  0.554602 -1.976159
994 0.042312 -1.628835  1.013822
995-0.093110  0.683847 -0.774753
996-0.185043  1.438572 -0.774753
997-0.394469 -0.642343  0.011374
998-1.174126  1.857148 -0.774753
999 0.234564  0.517098  0.393534
[1000 rows x 3 columns]
```

### 16.5.1 New syntax to window and resample operations

New in version 0.18.1.

Working with the resample, expanding or rolling operations on the groupby level used to require the application of helper functions. However, now 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.

```
                           'B': np.arange(20)})
In [109]: df_re
Out[109]:
    A  B
0   1  0
1   1  1
2   2  2
3   3  3
4   4  4
5   5  5
6   6  6
7   7  7
8   8  8
9   9  9
10 10 10
11 11 11
12 12 12
13 13 13
```

16.5. Transformation
```python
In [110]: df_re.groupby('A').rolling(4).B.mean()
Out[110]:
    A
0  0.0  NaN
1  0.0  NaN
2  0.0  NaN
3  0.5  2.5
4  2.0  3.5
5  3.5  4.5
... ...
5 13 10.5
14 11.0
15 11.5
16 12.0
17 12.5
18 13.0
19 13.5
Name: B, Length: 20, dtype: float64
```

The `expanding()` method will accumulate a given operation (e.g., sum) for all the members of each particular group.

```python
In [111]: df_re.groupby('A').expanding().sum()
Out[111]:
    A  B
   --
A
0  0.0  0.0
1  0.0  1.0
2  0.0  3.0
3  0.0  6.0
4  1.0  10.0
5  2.0  15.0
6  3.0  21.0
... ...
5 13 20.0  40.0
14 25.0  60.0
15 30.0  75.0
16 35.0  91.0
17 40.0 108.0
18 45.0 126.0
19 50.0 145.0
[20 rows x 2 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.
16.6 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 [115]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [116]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[116]:
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 [117]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc'))

In [118]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[118]:
      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 [119]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[119]:
     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

For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

In [120]: dff['C'] = np.arange(8)

In [121]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[121]:
      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.

In [122]: dff.groupby('B').head(2)
Out[122]:
     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
16.7 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:

```
In [123]: grouped = df.groupby('A')
In [124]: grouped.agg(lambda x: x.std())
Out[124]:
    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:

```
In [125]: grouped.std()
Out[125]:
    C    D
A
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:

```
In [126]: tsdf = pd.DataFrame(np.random.randn(1000, 3),
                      index=pd.date_range('1/1/2000', periods=1000),
                      columns=['A', 'B', 'C'])

In [127]: tsdf.iloc[::2] = np.nan
In [128]: grouped = tsdf.groupby(lambda x: x.year)
In [129]: grouped.fillna(method='pad')
Out[129]:
          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
2000-01-06  0.030091  0.186460  -0.680149
2000-01-07  0.030091  0.186460  -0.680149
          ...   ...   ...
2002-09-20  2.310215  0.157482  -0.064476
2002-09-21  2.310215  0.157482  -0.064476
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
```

16.7. Dispatching to instance methods

769
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 [130]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])
In [131]: g = pd.Series(list('abababab'))
In [132]: gb = s.groupby(g)
In [133]: gb.nlargest(3)
Out[133]:
   a    19.0
   0   9.0
   2   7.0
   b    8.0
   1    5.0
   3    3.3
dtype: float64

In [134]: gb.nsmallest(3)
   a    4.2
   2   7.0
   0   9.0
   b    1.0
   7    3.3
   3    5.0
dtype: float64
```

### 16.8 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 [135]: df
   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 [136]: grouped = df.groupby('A')
```
# could also just call \texttt{.describe()}

\begin{Verbatim}
In [137]: grouped['C'].apply(\texttt{lambda x: x.describe()})
\end{Verbatim}

\textbf{Out[137]}:

\begin{verbatim}
A
bar  count  3.000000
  mean  0.130980
  std   0.181231
  min  -0.077118
  25%   0.069390
  50%   0.215897
  75%   0.235029
 ...  
foo  mean  -0.359284
  std   0.912265
  min  -1.143704
  25%  -0.862495
  50%  -0.575247
  75%  -0.408530
  max   1.193555
Name: C, Length: 16, dtype: float64
\end{verbatim}

The dimension of the returned result can also change:

\begin{Verbatim}
In [138]: grouped = df.groupby('A')['C']

In [139]: def f(group):
    .....: return pd.DataFrame({'original': group,
    .....:                        'demeaned': group - group.mean()})
    .....:

In [140]: grouped.apply(f)
\end{Verbatim}

\textbf{Out[140]}:

\begin{verbatim}
demeaned     original
0  -0.215962  -0.575247
1   0.123181   0.254161
2  -0.784420  -1.143704
3   0.084917   0.215897
4   1.552839   1.193555
5  -0.208098  -0.077118
6  -0.049245  -0.408530
7  -0.503211  -0.862495
\end{verbatim}

\texttt{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

\begin{Verbatim}
In [141]: def f(x):
    .....: return pd.Series([x, x**2], index = ['x', 'x^2'])
    .....:

In [142]: s
\end{Verbatim}

\textbf{Out[142]}:

\begin{verbatim}
 0   9.0
 1   8.0
 2   7.0
 3   5.0
 4  19.0
 5   1.0
 6   4.2
\end{verbatim}
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.

Warning: In the current implementation apply calls func twice on the first group to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if func has side-effects, as they will take effect twice for the first group.

16.9 Other useful features

16.9.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:
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 [148]: df.groupby('A').std()
Out[148]:
   C    D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

### 16.9.2 NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. So 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).

### 16.9.3 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:

```
In [149]: data = pd.Series(np.random.randn(100))
In [150]: factor = pd.qcut(data, [0, .25, .5, .75, 1.])
In [151]: data.groupby(factor).mean()
```

```
(2.618, -0.684]  -1.331461
(-0.684, -0.0232]  -0.272816
(-0.0232, 0.541]  0.263607
(0.541, 2.369]  1.166038
```

dtype: float64

### 16.9.4 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.

```
In [152]: import datetime
In [153]: df = pd.DataFrame({
```

16.9. Other useful features

773
Groupby a specific column with the desired frequency. This is like resampling.

**In [155]:**

```python
df.groupby([pd.Grouper(freq='6M', key='Date'), 'Buyer']).sum()
```

**Out[155]:**

```
Date Buyer   Quantity
2013-02-28 Carl    1
          Mark    3
2014-02-28 Carl    9
          Joe    18
```

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

**In [156]:**

```python
df = df.set_index('Date')
```

**In [157]:**

```python
df['Date'] = df.index + pd.offsets.MonthEnd(2)
```

**In [158]:**

```python
df.groupby([pd.Grouper(freq='6M', key='Date'), 'Buyer']).sum()
```

**Out[158]:**

```
Date Buyer   Quantity
2013-02-28 Carl    1
          Mark    3
2014-02-28 Carl    9
          Joe    18
```
### 16.9.5 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 [160]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
In [161]: df
Out[161]:
   A  B
0  1  2
1  1  4
2  5  6

In [162]: g = df.groupby('A')
In [163]: g.head(1)
Out[163]:
   A  B
0  1  2
2  5  6
```

This shows the first or last n rows from each group.

### 16.9.6 Taking the nth row of each group

To select from a DataFrame or Series the nth item, use the nth method. 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 [165]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
In [166]: g = df.groupby('A')
In [167]: g.nth(0)
Out[167]:
   A  B
0  1  NaN
2  5  6

In [168]: g.nth(-1)
Out[168]:
   A  B
1  1  4
2  5  6
```

16.9. Other useful features
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`:

```python
# nth(0) is the same as g.first()
In [170]: g.nth(0, dropna='any')
Out[170]:
   A
0  4.0
5  6.0

In [171]: g.first()
Out[171]:
   A
0  4.0
5  6.0

# nth(-1) is the same as g.last()  
In [172]: g.nth(-1, dropna='any')  # NaNs denote group exhausted when using dropna
Out[172]:
   A
0  4.0
5  6.0

In [173]: g.last()
Out[173]:
   A
0  4.0
5  6.0

In [174]: g.B.nth(0, dropna='all')
Out[174]:
   A
0  4.0
5  6.0
Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```python
In [175]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
```
You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

```python
In [179]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq='B')
In [180]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [181]: df.groupby((df.index.year, df.index.month)).nth([0, 3, -1])
```

```
Out[181]:
         a  b
2014 4   1  1
      4   1  1
      5   1  1
      5   1  1
      6   1  1
      6   1  1
      6   1  1
```

### 16.9.7 Enumerate group items

To see the order in which each row appears within its group, use the `cumcount` method:

```python
In [182]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [183]: dfg
Out [183]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [184]: dfg.groupby('A').cumcount()
```

```
Out[184]:
   0  1  2  3
0  0  1  2  3
1  1
```

16.9. Other useful features 777
16.9.8 Enumerate groups

New in version 0.20.2.

To see the ordering of the groups (as opposed to the order of rows within a group given by cumcount) you can use the ngroup method.

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.
16.9.9 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 [190]: np.random.seed(1234)
In [191]: df = pd.DataFrame(np.random.randn(50, 2))
In [192]: df['g'] = np.random.choice(['A', 'B'], size=50)
In [193]: df.loc[df['g'] == 'B', 1] += 3
```

We can easily visualize this with a boxplot:

```python
In [194]: df.groupby('g').boxplot()
Out[194]:
```

![Boxplot](image)

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](link) for more.

**Warning:** For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See [here](link) for an explanation.

## 16.9. Other useful features
16.9.10 Piping function calls

New in version 0.21.0.

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.

For an example, 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 [195]: import numpy as np

in [196]: n = 1000

In [197]: 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.random(n)*50+10).round(2),
                     'Quantity': np.random.randint(1, 10, size=n)})

In [198]: df.head(2)
```

```
Out[198]:
          Product  Quantity  Revenue  Store
0  Product_1       6     30.35   Store_2
1  Product_3       2     35.69   Store_2
```

Now, to find prices per store/product, we can simply do:

```
in [199]: (df.groupby(['Store', 'Product'])
               .pipe(lambda grp: grp.Revenue.sum()/grp.Quantity.sum())
               .unstack().round(2))
```

```
Out[199]:
          Product  Product_1  Product_2  Product_3
Store
Store_1    6.93        6.82       7.15
Store_2    6.69        6.64       6.77
```

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

```
(df.groupby(['Store', 'Product'])).pipe(report_func)
```

where report_func takes a GroupBy object and creates a report from that.

16.10 Examples

16.10.1 Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.
In [200]: df = pd.DataFrame({"a": [1, 0, 0], "b": [0, 1, 0], "c": [1, 0, 0], "d": [2, 3, 4]})

In [201]: df
Out[201]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4

In [202]: df.groupby(df.sum(), axis=1).sum()
Out[202]:
   1  9
  0  2  2
  1  1  3
  2  0  4

16.10.2 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.)

In [203]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba"))

In [204]: dfg
Out[204]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a

In [205]: dfg.groupby(["A", "B"]).ngroup()
Out[205]:
   0 0
  1 0
  2 1
  3 2
  4 1
dtype: int64

In [206]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
Out[206]:
   0 0
  1 0
  2 1
  3 3
  4 2
dtype: int64
16.10.3 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 [207]: df = pd.DataFrame(np.random.randn(10,2))

In [208]: df
Out[208]:
   0   1
0  0.431670  0.882143
1 -0.026213 -1.941880
2 -1.106825 -0.667835
3  0.210712 -0.530195
4 -0.295191 -0.172722
5  0.638454  1.807622
6  1.008900  0.672822
7  0.770658  1.533002
8  0.576321 -0.819781
9 -1.302052  1.599477

In [209]: df.index // 5
   → Int64Index([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype='int64')

In [210]: df.groupby(df.index // 5).std()
   →
   0   1
0  0.595843  1.015451
1  0.931952  1.084644
```

16.10.4 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:

```
In [211]: df = pd.DataFrame({
    ....:     'a': [0, 0, 0, 0, 1, 1, 1, 1, 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, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1],
    ....: })
```
In [212]:
def compute_metrics(x):
    .....:    result = {'b_sum': x['b'].sum(), 'c_mean': x['c'].mean()}
    .....:    return pd.Series(result, name='metrics')
    .....:

In [213]: result = df.groupby('a').apply(compute_metrics)

In [214]: result
Out[214]:
            metrics     b_sum     c_mean
a
0       b_sum  2.0     c_mean  0.5
1       b_sum  2.0     c_mean  0.5
2       b_sum  2.0     c_mean  0.5
dtype: float64

In [215]: result.stack()
MERGE, JOIN, AND CONCATENATE

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

17.1 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:

In [1]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
...:'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({'A': ['A8', 'A9', 'A10', 'A11'],
...:'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', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, copy=True)
```

- `objs`: a sequence or mapping of Series, DataFrame, or Panel 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.
- `join_axes`: list of Index objects. Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic.
- `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 and example many of these arguments don’t make much sense. Let’s take 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 do stuff like 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
```

It’s not a stretch to see how this can be very useful. More detail on this functionality below.

---

**Note:** It is worth noting however, 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)
```
17.1.1 Set logic on the other axes

When gluing together multiple DataFrames (or Panels or...), for example, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in three ways:

- Take the (sorted) union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.
- Use a specific index (in the case of DataFrame) or indexes (in the case of Panel or future higher dimensional objects), i.e. the `join_axes` argument.

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'],
        'C': ['C2', 'C3', 'C6', 'C7'],
        'D': ['D2', 'D3', 'D6', 'D7'],
        'F': ['F2', 'F3', 'F6', 'F7'],
    }, index=[2, 3, 6, 7])

In [9]: result = pd.concat([df1, df4], axis=1)
```

Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```python
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```

Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:
In [11]: result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])

17.1.2 Concatenating using `append`

A useful shortcut to `concat` are the `append` instance methods on Series and DataFrame. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

In [12]: result = df1.append(df2)

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

In [13]: result = df1.append(df4)
append may take multiple objects to concatenate:

```python
In [14]: result = df1.append([df2, df3])
```

Note: Unlike `list.append` method, which appends to the original list and returns nothing, `append` here does not modify `df1` and returns its copy with `df2` appended.
17.1.3 Ignoring indexes on the concatenation axis

For DataFrames 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:

```
in [15]: result = pd.concat([df1, df4], ignore_index=True)
```

This is also a valid argument to `DataFrame.append`:

```
in [16]: result = df1.append(df4, ignore_index=True)
```
17.1.4 Concatenating with mixed ndims

You can concatenate a mix of Series and DataFrames. The Series will be transformed to DataFrames with the column name as the name of the Series.

```python
In [17]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')

In [18]: result = pd.concat([df1, s1], axis=1)
```

If unnamed Series are passed they will be numbered consecutively.

```python
In [19]: s2 = pd.Series(['_0', '_1', '_2', '_3'])

In [20]: result = pd.concat([df1, s2, s2, s2], axis=1)
```

Passing `ignore_index=True` will drop all name references.

```python
In [21]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```
17.1.5 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 inherits the parent Series’ name, when these existed.

```python
In [22]: s3 = pd.Series([0, 1, 2, 3], name='foo')
In [23]: s4 = pd.Series([0, 1, 2, 3])
In [24]: s5 = pd.Series([0, 1, 4, 5])
In [25]: pd.concat([s3, s4, s5], axis=1)
Out[25]:
       foo
0  0  0  0
1  1  1  1
2  2  2  4
3  3  3  5
```

Through the `keys` argument we can override the existing column names.

```python
In [26]: pd.concat([s3, s4, s5], axis=1, keys=['red','blue','yellow'])
Out[26]:
       red  blue  yellow
0     0     0       0
1     1     1       1
2     2     2       4
3     3     3       5
```

Let’s consider now a variation on the very first example presented:

```python
In [27]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```
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):

```python
In [28]: pieces = {'x': df1, 'y': df2, 'z': df3}
In [29]: result = pd.concat(pieces)
```
The MultiIndex created has levels that are constructed from the passed keys and the index of the DataFrame pieces:
If you wish to specify other levels (as will occasionally be the case), you can do so using the levels argument:

```python
In [32]: result = pd.concat(pieces, keys=['x', 'y', 'z'],
   ....:   levels=[['z', 'y', 'x', 'w']],
   ....:   names=['group_key'])
```

Yes, this is fairly esoteric, but is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

### 17.1.6 Appending rows to a DataFrame

While not especially efficient (since a new object must be created), you can append a single row to a DataFrame by passing a Series or dict to `append`, which returns a new DataFrame as above.

```python
In [34]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])
In [35]: result = df1.append(s2, 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.

You can also pass a list of dicts or Series:

```
In [36]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
   ....:   {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]
   ....:

In [37]: result = df1.append(dicts, ignore_index=True)
```

### 17.2 Database-style DataFrame 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 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 objects:

```python
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=True,
suffixes=('_x', '_y'), copy=True, indicator=False,
validate=None)
```

- **left**: A DataFrame object
- **right**: Another DataFrame object
- **on**: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the DataFrames will be inferred to be the join keys
- **left_on**: Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- **right_on**: Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- **left_index**: If `True`, use the index (row labels) from the left DataFrame as its join key(s). In the case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame
- **right_index**: Same usage as `left_index` for the right DataFrame
- **how**: One of 'left', 'right', 'outer', 'inner'. 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 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, `right_only` for observations whose merge key only appears in 'right' DataFrame, and `both` if the observation’s merge key is found in both.
  New in version 0.17.0.
- **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.
  New in version 0.21.0.

The return type will be the same as `left`. If `left` is a DataFrame and `right` is a subclass of DataFrame, the return type will still be DataFrame.

798 Chapter 17. Merge, join, and concatenate
merge is a function in the pandas namespace, and it is also available as a DataFrame instance method, with the calling DataFrame being implicitly considered the left object in the join.

The related DataFrame.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.

### 17.2.1 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 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:

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

Here is a more complicated example with multiple join keys:

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

17.2. Database-style DataFrame joining/merging
pandas: powerful Python data analysis toolkit, Release 0.21.0

```python
....:
....:
....:
....:
....:

In [42]:
right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                     'key2': ['K0', 'K0', 'K0', 'K0'],
                     'C': ['C0', 'C1', 'C2', 'C3'],
                     'D': ['D0', 'D1', 'D2', 'D3']})

In [43]:
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>Merge 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>
</tbody>
</table>

```python
In [44]:
result = pd.merge(left, right, how='left', on=['key1', 'key2'])
```

```python
In [45]:
result = pd.merge(left, right, how='right', on=['key1', 'key2'])
```
In [46]: result = pd.merge(left, right, how='outer', on=['key1', 'key2'])

Here is another example with duplicate join keys in DataFrames:

In [48]: left = pd.DataFrame({"A" : [1,2], "B" : [2,2]})
In [49]: right = pd.DataFrame({"A" : [4,5,6], "B" : [2,2,2]})
In [50]: result = pd.merge(left, right, on='B', how='outer')

17.2. Database-style DataFrame joining/merging
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.

17.2.2 Checking for duplicate keys

New in version 0.21.0.

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.

```
In [51]: left = pd.DataFrame({'A' : [1,2], 'B' : [1,2]})
In [52]: 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.

```
In [53]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[53]:
    A_x B  A_y
0  1  1  NaN
1  2  2   4.0
2  2  2   5.0
3  2  2   6.0
```

17.2.3 The merge indicator

New in version 0.17.0.

`merge` now accepts the argument `indicator`. If True, a Categorical-type column called `_merge` will be added to the output object that takes on values:
### Observation Origin

<table>
<thead>
<tr>
<th>Merge key only in 'left' frame</th>
<th>left_only</th>
</tr>
</thead>
<tbody>
<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 [54]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})

In [55]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})

In [56]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)

```
Out[56]:
  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
```

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 [57]: pd.merge(df1, df2, on='col1', how='outer', indicator='indicator_column')

```
Out[57]:
  col1  col_left  col_right  indicator_column
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
```

### 17.2.4 Merge Dtypes

New in version 0.19.0.

Merging will preserve the dtype of the join keys.

In [58]: left = pd.DataFrame({'key': [1], 'v1': [10]})

In [59]: left

```
Out[59]:
  key  v1
0   1  10
```

In [60]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [61]: right

```
Out[61]:
  key  v1
0   1  20
1   2  30
```

We are able to preserve the join keys

In [62]: pd.merge(left, right, how='outer')

```
Out[62]:
  key  v1
0   1  10
1   1  20
2   2  30
```
In [63]: pd.merge(left, right, how='outer').dtypes

\n
Out[63]:

   key  int64
   v1   int64
dtype: object

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.

In [64]: pd.merge(left, right, how='outer', on='key')

Out[64]:

   key   v1_x  v1_y
 0   1    10.0   20
 1   2 .NaN    30

In [65]: pd.merge(left, right, how='outer', on='key').dtypes

Out[65]:

   key  int64
   v1_x float64
   v1_y  int64
dtype: object

New in version 0.20.0.
Merging will preserve category dtypes of the mergands. See also the section on categoricals

The left frame.

In [66]: from pandas.api.types import CategoricalDtype

In [67]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [68]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))

In [69]: left = pd.DataFrame({'X': X, 'Y': np.random.choice(['one', 'two', 'three'], size=(10,))})

In [70]: left

Out[70]:

   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 [71]: left.dtypes

X  category
Y  object
dtype: object
The right frame.

```python
In [72]: right = pd.DataFrame(
....:     {'X': pd.Series(['foo', 'bar'],
....:                        dtype=CategoricalDtype(['foo', 'bar'])),
....:     'Z': [1, 2]
....: )
....:
```

```python
In [73]: right
Out[73]:
   X   Z
0  foo  1
1  bar  2
```

```python
In [74]: right.dtypes
Out[74]:
X    category
Z     int64
dtype: object
```

The merged result

```python
In [75]: result = pd.merge(left, right, how='outer')
```

```python
In [76]: result
Out[76]:
   X   Y   Z
0  bar one  2
1  bar three  2
2  bar one  2
3  bar three  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
```

```python
In [77]: result.dtypes
Out[77]:
   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 *object* dtype.

**Note:** Merging on category dtypes that are the same can be quite performant compared to object dtype merging.
17.2.5 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 [78]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                      'B': ['B0', 'B1', 'B2'],
                      index=['K0', 'K1', 'K2'])

In [79]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                         'D': ['D0', 'D2', 'D3'],
                         index=['K0', 'K2', 'K3'])

In [80]: result = left.join(right)
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>A0</td>
<td>B0</td>
<td>D0</td>
</tr>
<tr>
<td>K1</td>
<td>A1</td>
<td>B1</td>
<td>D2</td>
</tr>
<tr>
<td>K2</td>
<td>A2</td>
<td>B2</td>
<td>D3</td>
</tr>
</tbody>
</table>

```
In [81]: result = left.join(right, how='outer')
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>A0</td>
<td>B0</td>
<td>C0</td>
</tr>
<tr>
<td>K1</td>
<td>A1</td>
<td>B1</td>
<td>NaN</td>
</tr>
<tr>
<td>K2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

```
In [82]: result = left.join(right, how='inner')
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>A0</td>
<td>B0</td>
<td>D0</td>
</tr>
<tr>
<td>K2</td>
<td>A2</td>
<td>B2</td>
<td>D2</td>
</tr>
</tbody>
</table>
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:

```python
In [83]: result = pd.merge(left, right, left_index=True, right_index=True, how='outer')
```

17.2.6 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:

```python
In [85]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3'],
                        'key': ['K0', 'K1', 'K0', 'K1']})

In [86]: right = pd.DataFrame({'C': ['C0', 'C1'],
                        'D': ['D0', 'D1']},
                       index=['K0', 'K1'])
```
In [87]: result = left.join(right, on='key')

To join on multiple keys, the passed DataFrame must have a MultiIndex:

In [89]: left = pd.DataFrame({
        'A': ['A0', 'A1', 'A2', 'A3'],
        'B': ['B0', 'B1', 'B2', 'B3'],
        'key1': ['K0', 'K0', 'K1', 'K2'],
        'key2': ['K0', 'K1', 'K0', 'K1']})

In [90]: index = pd.MultiIndex.from_tuples([(K0, K0), (K1, K0), (K2, K0), (K2, K1)])

In [91]: 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 [92]: 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 [93]: result = left.join(right, on=['key1', 'key2'], how='inner')
```

As you can see, this drops any rows where there was no match.

### 17.2.7 Joining a single Index to a Multi-index

You can join a singly-indexed DataFrame with a level of a multi-indexed DataFrame. The level will match on the name of the index of the singly-indexed frame against a level name of the multi-indexed frame.

```
In [94]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                        'B': ['B0', 'B1', 'B2'],
                        index=pd.Index(['K0', 'K1', 'K2'], name='key'))
```

```
In [95]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                                       ('K2', 'Y2'), ('K2', 'Y3')],
                                       names=['key', 'Y'])
```

```
In [96]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                         'D': ['D0', 'D1', 'D2', 'D3'],
                         index=index)
```

```
In [97]: result = left.join(right, how='inner')
```
This is equivalent but less verbose and more memory efficient / faster than this.

```
In [98]: result = pd.merge(left.reset_index(), right.reset_index(),
                      on=['key'], how='inner').set_index(['key', 'Y'])
```

17.2.8 Joining with two multi-indexes

This is not implemented via `join` at-the-moment, however it can be done using the following.

```
In [99]: index = pd.MultiIndex.from_tuples([(‘K0’, ‘X0’), (‘K0’, ‘X1’),
                                       (‘K1’, ‘X2’)],
                                           names=[‘key’, ‘X’])

In [100]: left = pd.DataFrame({‘A’: [‘A0’, ‘A1’, ‘A2’],
                         index=index)

In [101]: result = pd.merge(left.reset_index(), right.reset_index(),
                         on=[‘key’], how=’inner’).set_index([‘key’, ‘X’, ‘Y’])
```
17.2.9 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 [102]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})
In [103]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})
In [104]: result = pd.merge(left, right, on='k')
```

```
<table>
<thead>
<tr>
<th>k</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>1</td>
</tr>
<tr>
<td>K1</td>
<td>2</td>
</tr>
<tr>
<td>K2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>k</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>4</td>
</tr>
<tr>
<td>K0</td>
<td>5</td>
</tr>
<tr>
<td>K3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
Result
<table>
<thead>
<tr>
<th>k</th>
<th>v_l</th>
<th>v_r</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>K0</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
In [105]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```

```
<table>
<thead>
<tr>
<th>k</th>
<th>v_l</th>
<th>v_r</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>K0</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

**DataFrame.join** has lsuffix and rsuffix arguments which behave similarly.

```
In [106]: left = left.set_index('k')
In [107]: right = right.set_index('k')
In [108]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```
### 17.2.10 Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to `DataFrame.join` to join them together on their indexes. The same is true for `Panel.join`.

```python
In [109]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])
In [110]: result = left.join([right, right2])
```

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>right2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
<td></td>
<td>v_l</td>
</tr>
<tr>
<td>K0</td>
<td>1</td>
<td>K0</td>
<td>1</td>
</tr>
<tr>
<td>K0</td>
<td>1</td>
<td>K0</td>
<td>1</td>
</tr>
<tr>
<td>K1</td>
<td>2</td>
<td>K0</td>
<td>2</td>
</tr>
<tr>
<td>K2</td>
<td>3</td>
<td>K3</td>
<td>3</td>
</tr>
</tbody>
</table>

### 17.2.11 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 [111]: df1 = pd.DataFrame([[np.nan, 3., 5.], [-4.6, np.nan, np.nan],
                      [np.nan, 7., np.nan]])
          ...
In [112]: df2 = pd.DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4.]],
                      index=[1, 2])
```

For this, use the `combine_first` method:
In [113]: 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 inplace:

In [114]: df1.update(df2)

17.3 Timeseries friendly merging

17.3.1 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 [115]: left = pd.DataFrame({
        'k': ['K0', 'K1', 'K1', 'K2'],
        'lv': [1, 2, 3, 4],
        's': ['a', 'b', 'c', 'd']})

In [116]: right = pd.DataFrame({
        'k': ['K1', 'K2', 'K4'],
        'rv': [1, 2, 3]})

In [117]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')

Out[117]:

<table>
<thead>
<tr>
<th>k</th>
<th>lv</th>
<th>s</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>1.0</td>
<td>a</td>
<td>NaN</td>
</tr>
<tr>
<td>K1</td>
<td>1.0</td>
<td>a</td>
<td>1.0</td>
</tr>
<tr>
<td>K2</td>
<td>1.0</td>
<td>a</td>
<td>2.0</td>
</tr>
<tr>
<td>K4</td>
<td>1.0</td>
<td>a</td>
<td>3.0</td>
</tr>
</tbody>
</table>
17.3.2 Merging AsOf

New in version 0.19.0.

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.

```python
In [118]: 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'],
    ....:     '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 [119]: 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,
    ....:                720.77, 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 [120]: trades
Out[120]:
<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
</tr>
</tbody>
</table>

In [121]: quotes
\|
<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.97</td>
<td>51.98</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.99</td>
<td>52.00</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>97.99</td>
<td>98.01</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.50</td>
<td>720.88</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>52.01</td>
<td>52.03</td>
</tr>
</tbody>
</table>

By default we are taking the asof of the quotes.

In [122]: pd.merge_asof(trades, quotes,
                   on='time',
                   by='ticker')
Out[122]:
<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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>51.97</td>
<td>51.98</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

We only asof within 2ms between the quote time and the trade time.

In [123]: pd.merge_asof(trades, quotes,
                   on='time',
                   by='ticker',
                   tolerance=pd.Timedelta('2ms'))
Out[123]:
<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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

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.

In [124]: pd.merge_asof(trades, quotes,
                   on='time',
                   by='ticker',
                   tolerance=pd.Timedelta('10ms'),
                   asof='earliest')
Out[124]:
<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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
allow_exact_matches=False)

Out[124]:

<table>
<thead>
<tr>
<th></th>
<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</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>51.97</td>
<td>51.98</td>
</tr>
<tr>
<td>2</td>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
18.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```
In [1]: df
Out[1]:
   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
```

For the curious here is how the above DataFrame was created:

```python
import pandas.util.testing as tm; tm.N = 3

def unpivot(frame):
    N, K = frame.shape
    data = {'value' : frame.values.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'])

df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable A we could do:

```
In [2]: df[df['variable'] == 'A']
Out[2]:
   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, use the `pivot` function:
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 toponmost level indicates the respective value column:

```python
In [4]: df['value2'] = df['value'] * 2
In [5]: pivoted = df.pivot('date', 'variable')
In [6]: pivoted
```
```
Out[6]:
```
<table>
<thead>
<tr>
<th>variable</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>value2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.469112</td>
<td>-1.135632</td>
<td>0.119209</td>
<td>-2.104569</td>
<td>0.938225</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.282863</td>
<td>1.212112</td>
<td>-1.044236</td>
<td>-0.494929</td>
<td>-0.565727</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.509059</td>
<td>-0.173215</td>
<td>-0.861849</td>
<td>1.071804</td>
<td>-3.018117</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>variable</td>
<td></td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.238417</td>
<td>-4.209138</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-2.088472</td>
<td>-0.989859</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.723698</td>
<td>2.143608</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

You of course can then select subsets from the pivoted DataFrame:

```python
In [7]: pivoted['value2']
```
```
Out[7]:
```
<table>
<thead>
<tr>
<th>variable</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.938225</td>
<td>-2.27125</td>
<td>-4.209138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.564727</td>
<td>2.424224</td>
<td>-2.088472</td>
<td>-0.989859</td>
<td></td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-3.018117</td>
<td>-0.346429</td>
<td>-1.723698</td>
<td>2.143608</td>
<td></td>
</tr>
</tbody>
</table>
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

### 18.2 Reshaping by stacking and unstacking

Closely related to the `pivot` function are the related `stack` and `unstack` functions currently available on Series and DataFrame. These functions are designed to work together with MultiIndex objects (see the section on *hierarchical indexing*). Here are essentially what these functions do:

- **stack**: “pivot” a level of the (possibly hierarchical) column labels, returning a DataFrame with an index with a new inner-most level of row labels.
- **unstack**: inverse operation from `stack`: “pivot” a level of the (possibly hierarchical) row index to the column axis, producing a reshaped 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:

```python
In [8]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                        'foo', 'foo', 'qux', 'qux'],
                        ['one', 'two', 'one', 'two',
                        'one', 'two', 'one', 'two']]))

In [9]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [10]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [11]: df2 = df[:4]

In [12]: df2

Out[12]:
   A      B
first  second
bar    one  0.721555 -0.706771
      two -1.039575  0.271860
baz    one -0.424972  0.567020
      two  0.276232 -1.087401
```

The `stack` function “compresses” a level in the DataFrame’s 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:

```python
In [13]: stacked = df2.stack()

In [14]: stacked

Out[14]:
   first  second
bar    one       A  0.721555
      B   -0.706771
      two  -1.039575
      B   0.271860
baz    one       A -0.424972
      B   0.567020
      two  0.276232
      B -1.087401
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:

```python
In [15]: stacked.unstack()

Out[15]:
   A      B
first  second
bar    one  0.721555 -0.706771
      two -1.039575  0.271860
baz    one -0.424972  0.567020
      two  0.276232 -1.087401
```

18.2. Reshaping by stacking and unstacking
If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [18]: stacked.unstack('second')
Out[18]:
      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
```

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:

```
In [19]: index = pd.MultiIndex.from_product([[2,1], ['a', 'b']])
In [20]: df = pd.DataFrame(np.random.randn(4), index=index, columns=['A'])
In [21]: df
Out[21]:
       A
2  a  -0.370647
   b  -1.157892
1  a  -1.344312
   b   0.844885
In [22]: all(df.unstack().stack() == df.sort_index())
Out[22]:
```

```
→True
```

while the above code will raise a `TypeError` if the call to `sort_index` is removed.

### 18.2.1 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.
In [23]: columns = pd.MultiIndex.from_tuples([
    ('A', 'cat', 'long'), ('B', 'cat', 'long'),
    ('A', 'dog', 'short'), ('B', 'dog', 'short')
],
   names=['exp', 'animal', 'hair_length'])

In [24]: df = pd.DataFrame(np.random.randn(4, 4), columns=columns)

In [25]: df
Out[25]:
   exp
animal  A    B   A   B
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 [26]: df.stack(level=['animal', 'hair_length'])
Out[26]:
   exp
animal  A    B
    hair_length
0 cat long 1.075770 -0.109050
  dog long 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).

# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [27]: df.stack(level=[1, 2])
Out[27]:
   exp
animal  A    B
    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

18.2.2 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:
As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```python
In [33]: df2.stack('exp')
Out[33]:
   exp  A   B
animal  A  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  0.875906 -2.211372  0.974466 -2.006747
       foo one -1.413681  1.607920  1.024180  0.569605
          two  0.875906 -2.211372  0.974466 -2.006747
       qux two -1.226825  0.769804 -1.281247 -0.727707

In [34]: df2.stack('animal')
   exp  A   B
animal  A  cat  dog  cat  dog
       bar one  0.895717  2.565646
          two -1.206412  0.805244
       baz one  0.410835 -0.827317
          two  0.132003  0.813850
       foo one -1.413681  0.569605
          two  1.024180  1.607920
       qux two -1.226825 -0.727707
          B  0.769804
```

Chapter 18. Reshaping and Pivot Tables
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.

```
In [35]: df3 = df.iloc[[0, 1, 4, 7], [1, 2]]

In [36]: df3
Out[36]:
exp  B
animal dog cat
first  second
bar one    0.805244 -1.206412
        two    1.340309 -1.170299
foo one    1.607920  1.024180
qux two   -0.727707 -0.769804

In [37]: df3.unstack()

--> exp  B
animal dog cat
second  one  two  one  two
first
bar    0.805244  1.340309 -1.206412 -1.170299
foo    1.607920  NaN     1.024180   NaN
qux   -1.000000e+09  0.769804  NaN  -1.281247

```

Alternatively, unstack takes an optional `fill_value` argument, for specifying the value of missing data.

```
In [38]: df3.unstack(fill_value=-1e9)
Out[38]:
exp  B
animal dog cat
second  one  two  one  two
first
bar   8.052440e-01  1.340309e+00 -1.206412e+00 -1.170299e+00
foo   1.607920e+00 -1.000000e+09  1.024180e+00 -1.000000e+09
qux -1.000000e+09  7.698036e-01 -1.000000e+09 -1.281247e+00

```

### 18.2.3 With a MultiIndex

Unstacking when the columns are a MultiIndex is also careful about doing the right thing:

```
In [39]: df[:3].unstack(0)
Out[39]:
exp A  B  A  \
animal  cat  dog  cat  dog
first  bar  baz  bar  baz  bar
second one  0.895717  0.410835  0.805244  0.81385 -1.206412  0.132003  2.565646
two   1.431256   NaN  1.340309   NaN -1.170299   NaN  -0.226169

```

18.2. Reshaping by stacking and unstacking
18.3 Reshaping by Melt

The top-level `melt()` and `melt()` functions 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 [41]: cheese = pd.DataFrame({'first' : ['John', 'Mary'],
                           'last' : ['Doe', 'Bo'],
                           'height' : [5.5, 6.0],
                           'weight' : [130, 150]})

In [42]: cheese
Out[42]:
          first  height   last  weight
      0     John   5.5   Doe     130
      1    Mary    6.0    Bo     150

In [43]: cheese.melt(id_vars=['first', 'last'])
Out[43]:
         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 [44]: cheese.melt(id_vars=['first', 'last'], var_name='quantity')

→
first last quantity 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

Another way to transform is to use the `wide_to_long` panel data convenience function.

In [45]: dft = 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 [46]: dft["id"] = dft.index

In [47]: dft
Out[47]:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
<td>2.5</td>
<td>3.2</td>
<td>-0.121306</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>1.2</td>
<td>1.3</td>
<td>-0.097883</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>0.7</td>
<td>0.1</td>
<td>0.695775</td>
<td>2</td>
</tr>
</tbody>
</table>

In [48]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")

→

<table>
<thead>
<tr>
<th>id year</th>
<th>X</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1970</td>
<td>-0.121306</td>
<td>a</td>
<td>2.5</td>
</tr>
<tr>
<td>1 1970</td>
<td>-0.097883</td>
<td>b</td>
<td>1.2</td>
</tr>
<tr>
<td>2 1970</td>
<td>0.695775</td>
<td>c</td>
<td>0.7</td>
</tr>
<tr>
<td>0 1980</td>
<td>-0.121306</td>
<td>d</td>
<td>3.2</td>
</tr>
<tr>
<td>1 1980</td>
<td>-0.097883</td>
<td>e</td>
<td>1.3</td>
</tr>
<tr>
<td>2 1980</td>
<td>0.695775</td>
<td>f</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### 18.4 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 [49]: df
Out[49]:

<table>
<thead>
<tr>
<th>exp</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>animal</td>
<td>cat</td>
<td>dog</td>
<td>cat</td>
</tr>
<tr>
<td>first second</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar one</td>
<td>0.895717</td>
<td>0.805244</td>
<td>-1.206412</td>
</tr>
<tr>
<td>two</td>
<td>1.431256</td>
<td>1.340309</td>
<td>-1.170299</td>
</tr>
</tbody>
</table>

18.4. Combining with stats and GroupBy
In [50]: df.stack().mean(1).unstack()

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

# same result, another way
In [51]: df.groupby(level=1, axis=1).mean()

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  -1.254036  0.021048
qux   one  0.067976  -0.648927
two  -1.254036  0.021048

In [52]: df.stack().groupby(level=1).mean()

→ exp       A       B
second
one  0.071448  0.455513
two -0.424186 -0.204486

In [53]: df.mean().unstack()

→ exp       A       B
animal
  cat  0.060843  0.018596
dog  0.413580  0.232430
18.5 Pivot tables

While pivot provides general purpose pivoting of DataFrames with various data types (strings, numerics, etc.), Pandas also provides the pivot_table function for pivoting with aggregation of numeric data.

The function pandas.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:

```
In [54]: import datetime
In [55]: 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, 14)] +
                              [datetime.datetime(2013, i, 15)
                               for i in range(1, 13)])
...
```

```
In [56]: df
Out[56]:
   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
5  one  C  bar -0.732339 -2.182937 2013-06-01
6  two  A  foo  0.687738  0.380396 2013-07-01
7  three  B  foo  1.314232 -0.251905 2013-08-01
8  one  C  bar -0.345352  0.206053 2013-09-01
9  two  A  foo  1.314232 -0.251905 2013-09-15
10 three  B  foo  0.690579 -2.213588 2013-08-15
11 one  C  foo  0.995761  1.063327 2013-09-15
12 one  A  bar  2.396780  1.266143 2013-10-15
13 two  B  bar  0.014871  0.299368 2013-11-15
14 three  C  bar  3.357427 -0.863838 2013-12-15
```

We can produce pivot tables from this data very easily:
In [57]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])

Out[57]:

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>1.120915 -0.514058</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.338421 0.002759</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.538846 0.699535</td>
<td></td>
</tr>
<tr>
<td>three</td>
<td>-1.181568 NaN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NaN 0.433512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.588783 NaN</td>
<td></td>
</tr>
<tr>
<td>two</td>
<td>1.000985</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.158248 NaN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NaN 0.176180</td>
<td></td>
</tr>
</tbody>
</table>

In [58]: pd.pivot_table(df, values='D', index=['B'], columns=['A', 'C'], aggfunc=np.sum)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>one</th>
<th>three</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>bar</td>
<td>foo</td>
<td>bar</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>2.241830 -1.028115 -2.363137 NaN NaN 2.001971</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.676843 0.005518 NaN 0.867024 0.316495 NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.077692 1.399070 1.177566 NaN NaN 0.352360</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In [59]: pd.pivot_table(df, values=['D','E'], index=['B'], columns=['A', 'C'], aggfunc=np.sum)

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>bar</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>2.241830 -1.028115 -2.363137 NaN NaN 2.001971 2.786113</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.676843 0.005518 NaN 0.867024 0.316495 NaN 1.368280</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.077692 1.399070 1.177566 NaN NaN 0.352360 -1.976883</td>
</tr>
</tbody>
</table>

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:

In [60]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])

Out[60]:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>bar</td>
</tr>
<tr>
<td>A B</td>
<td>one</td>
<td>1.120915 -0.514058 1.393057 -0.021605</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.338421 0.002759 0.684140 -0.551692</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.538846 0.699535 -0.988442 0.747859</td>
</tr>
<tr>
<td>three</td>
<td>-1.181568 NaN 0.961289 NaN</td>
<td></td>
</tr>
</tbody>
</table>
Also, you can use `Grouper` for index and columns keywords. For detail of `Grouper`, see *Grouping with a Grouper specification*.

```python
In [61]: pd.pivot_table(df, values='D', index=pd.Grouper(freq='M', key='F'), columns='C')
Out[61]:
+----------------+----------------+----------------+
|                | 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     | 0.158248       | NaN            |
| 2013-12-31     | 0.588783       | NaN            |
```

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```python
In [62]: table = pd.pivot_table(df, index=['A', 'B'], columns=['C'])
In [63]: print(table.to_string(na_rep=''))
```

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bar</td>
</tr>
<tr>
<td>A</td>
<td>one</td>
</tr>
<tr>
<td>B</td>
<td>1.120915</td>
</tr>
<tr>
<td></td>
<td>-0.338421</td>
</tr>
<tr>
<td></td>
<td>-0.338442</td>
</tr>
<tr>
<td></td>
<td>0.588783</td>
</tr>
<tr>
<td></td>
<td>0.158248</td>
</tr>
<tr>
<td></td>
<td>0.588783</td>
</tr>
</tbody>
</table>

Note that `pivot_table` is also available as an instance method on DataFrame.

### 18.5.1 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:

```python
In [64]: df.pivot_table(index=['A', 'B'], columns='C', margins=True, aggfunc=np.std)
Out[64]:
```

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bar</td>
</tr>
<tr>
<td>A</td>
<td>one</td>
</tr>
<tr>
<td>B</td>
<td>1.120915</td>
</tr>
<tr>
<td></td>
<td>-0.338421</td>
</tr>
<tr>
<td></td>
<td>-0.338442</td>
</tr>
<tr>
<td></td>
<td>0.588783</td>
</tr>
<tr>
<td></td>
<td>0.158248</td>
</tr>
<tr>
<td></td>
<td>0.588783</td>
</tr>
</tbody>
</table>
18.6 Cross tabulations

Use the `crosstab` function 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 [65]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [66]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [67]: b = np.array([one, one, two, one, two, one], dtype=object)

In [68]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [69]: pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
```

```
Out[69]:
       b    c
     | dull | shiny|
---|------|------|
   a |       |      |
   bar| 1     | 0     |
   foo| 2     | 1     |
```

If `crosstab` receives only two Series, it will provide a frequency table.
In [70]: df = pd.DataFrame({'A': [1, 2, 2, 2, 2], 'B': [3, 3, 4, 4, 4], 'C': [1, 1, np.nan, 1, 1]})

In [71]: df
Out[71]:
   A  B  C
0  1  3  1.0
1  2  3  1.0
2  2  4  NaN
3  2  4  1.0
4  2  4  1.0

In [72]: pd.crosstab(df.A, df.B)
Out[72]:
   B  3  4
A
1  1  0
2  1  3

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 [73]: foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
In [74]: bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
In [75]: pd.crosstab(foo, bar)
Out[75]:
   col_0 d e f
row_0
a  1  0  0
b  0  1  0
c  0  0  0

18.6.1 Normalization

New in version 0.18.1.

Frequency tables can also be normalized to show percentages rather than counts using the normalize argument:

In [76]: pd.crosstab(df.A, df.B, normalize=True)
Out[76]:
   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 [77]: pd.crosstab(df.A, df.B, normalize='columns')
Out[77]:
   B  3  4
A
1  0.5  0.0
2  0.5  1.0

18.6. Cross tabulations
pandas: powerful Python data analysis toolkit, Release 0.21.0

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:

```python
In [78]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum)
Out[78]:
   B  3  4
A
1  1.0 NaN
2  1.0  2.0
```

### 18.6.2 Adding Margins

Finally, one can also add margins or normalize this output.

```python
In [79]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum, normalize=True,
   ....: margins=True)
   ....:
Out[79]:
   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
```

### 18.7 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:

```python
In [80]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])
In [81]: pd.cut(ages, bins=3)
Out[81]:
[(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]): [(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:

```python
In [82]: c = pd.cut(ages, bins=[0, 18, 35, 70])
In [83]: c
Out[83]:
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]
Categories (3, interval[int64]): [(0, 18] < (18, 35] < (35, 70]
```

New in version 0.20.0.

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)
```
18.8 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:

```python
In [84]: df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})
In [85]: pd.get_dummies(df['key'])
Out[85]:
   a  b  c
0  0  1  0
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 [86]: dummies = pd.get_dummies(df['key'], prefix='key')
In [87]: dummies
Out[87]:
   key_a  key_b  key_c
0      0      1      0
1      0      1      0
2      1      0      0
3      0      0      1
4      1      0      0
5      0      1      0
```

This function is often used along with discretization functions like `cut`:

```python
In [89]: values = np.random.randn(10)
In [90]: values
Out[90]:
array([ 0.4082, -1.0481, -0.0257, -0.9884, 0.0941, 1.2627, 1.29
   0.0824, -0.0558, 0.5366])
In [91]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
In [92]: pd.get_dummies(pd.cut(values, bins))
Out[92]:
   (0.0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1.0]
0      0      0      1      0      0
1      0      0      0      0      0
```
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.

```python
In [93]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'], 'C': [1, 2, 3]})
In [94]: pd.get_dummies(df)
Out[94]:
   C  A_a  A_b  B_b  B_c
0  1    1    0    0    1
1  2    0    1    0    1
2  3    1    0    1    0
```

All non-object columns are included untouched in the output.

You can control the columns that are encoded with the columns keyword.

```python
In [95]: pd.get_dummies(df, columns=['A'])
Out[95]:
   B  C  A_a  A_b
0  c  1    1    0
1  c  2    0    1
2  b  3    1    0
```

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

```python
In [96]: simple = pd.get_dummies(df, prefix='new_prefix')
In [97]: simple
Out[97]:
   C  new_prefix_a  new_prefix_b  new_prefix_b  new_prefix_c
0  1              1              0              0              1
1  2              0              1              0              1
2  3              1              0              1              0
In [98]: from_list = pd.get_dummies(df, prefix=['from_A', 'from_B'])
In [99]: from_list
```
New in version 0.18.0.

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`.

When a column contains only one level, it will be omitted in the result.
18.9 Factorizing values

To encode 1-d values as an enumerated type use `factorize`:

```
In [108]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])

In [109]: x
Out[109]:
0    A
1    A
2  NaN
3    B
4  3.14
5    inf
dtype: object

In [110]: labels, uniques = pd.factorize(x)

In [111]: labels
Out[111]:
array([ 0, 0, -1, 1, 2, 3])

In [112]: uniques
Out[112]: 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 [2]: pd.factorize(x, sort=True)
Out[2]:
(array([ 2, 2, -1, 3, 0, 1]),
     Index([3.14, inf, u'A', u'B'], dtype='object'))

In [3]: np.unique(x, return_inverse=True)[::-1]
Out[3]: (array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
```

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`. 

836 Chapter 18. Reshaping and Pivot Tables
pandas has proven very successful as a tool for working with time series data, especially in the financial data analysis space. Using the NumPy datetime64 and timedelta64 dtypes, we have 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.

In working with time series data, we will frequently seek to:

• generate sequences of fixed-frequency dates and time spans
• conform or convert time series to a particular frequency
• compute “relative” dates based on various non-standard time increments (e.g. 5 business days before the last business day of the year), or “roll” dates forward or backward

pandas provides a relatively compact and self-contained set of tools for performing the above tasks.

Create a range of dates:

```python
# 72 hours starting with midnight Jan 1st, 2011
In [1]: rng = pd.date_range('1/1/2011', periods=72, freq='H')
In [2]: rng[:5]
Out[2]:
 DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 01:00:00',
               '2011-01-01 02:00:00', '2011-01-01 03:00:00',
               '2011-01-01 04:00:00'],
               dtype='datetime64[ns]', freq='H')
```

Index pandas objects with dates:

```python
In [3]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [4]: ts.head()
Out[4]:
   2011-01-01 00:00:00    0.469112
   2011-01-01 01:00:00   -0.282863
   2011-01-01 02:00:00   -1.509059
   2011-01-01 03:00:00   -1.135632
   2011-01-01 04:00:00    1.212112
Freq: H, dtype: float64
```

Change frequency and fill gaps:

```python
# to 45 minute frequency and forward fill
In [5]: converted = ts.asfreq('45Min', method='pad')
```

```
In [6]: converted.head()
Out[6]:
2011-01-01 00:00:00 0.469112
2011-01-01 00:45:00 0.469112
2011-01-01 01:30:00 -0.282863
2011-01-01 02:15:00 -1.509059
2011-01-01 03:00:00 -1.135632
Freq: 45T, dtype: float64

Resample:
# Daily means
In [7]: ts.resample('D').mean()
Out[7]:
2011-01-01  -0.319569
2011-01-02  -0.337703
2011-01-03   0.117258
Freq: D, dtype: float64

19.1 Overview

Following table shows the type of time-related classes pandas can handle and how to create them.

<table>
<thead>
<tr>
<th>Class</th>
<th>Remarks</th>
<th>How to create</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp</td>
<td>Represents a single timestamp</td>
<td>to_datetime, Timestamp</td>
</tr>
<tr>
<td>DatetimeIndex</td>
<td>Index of Timestamp</td>
<td>to_datetime, date_range, bdate_range, DatetimeIndex</td>
</tr>
<tr>
<td>Period</td>
<td>Represents a single time span</td>
<td>Period</td>
</tr>
<tr>
<td>PeriodIndex</td>
<td>Index of Period</td>
<td>period_range, PeriodIndex</td>
</tr>
</tbody>
</table>

19.2 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.

However, in many cases it is more natural to associate things like change variables with a time span instead. The span represented by Period can be specified explicitly, or inferred from datetime string format.

For example:
In [11]: pd.Period('2011-01')
Out[11]: Period('2011-01', 'M')

In [12]: pd.Period('2012-05', freq='D')
Out[12]: Period('2012-05-01', 'D')

Timestamp and Period can be the index. Lists of Timestamp and Period are automatically coerced to DatetimeIndex and PeriodIndex respectively.

In [14]: ts = pd.Series(np.random.randn(3), dates)
In [15]: type(ts.index)
Out[15]: pandas.core.indexes.datetimes.DatetimeIndex
In [16]: ts.index
Out[16]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)
In [17]: ts
Out[17]: 2012-05-01 -0.410001
2012-05-02 -0.078638
2012-05-03 0.545952
dtype: float64

In [18]: periods = [pd.Period('2012-01'), pd.Period('2012-02'), pd.Period('2012-03')]
In [19]: ts = pd.Series(np.random.randn(3), periods)
In [20]: type(ts.index)
Out[20]: pandas.core.indexes.period.PeriodIndex
In [21]: ts.index
Out[21]: PeriodIndex(['2012-01', '2012-02', '2012-03'], dtype='period[M]', freq='M')
In [22]: ts
Out[22]: 2012-01 -1.219217
2012-02 -1.226825
2012-03 0.769804
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.
19.3 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:

```python
In [23]: pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))
Out[23]:
0  2009-07-31
1  2010-01-10
2      NaT
dtype: datetime64[ns]
```

```python
In [24]: pd.to_datetime(['2005/11/23', '2010.12.31'])
Out[24]:

→ DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None)
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```python
In [25]: pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)
Out[25]: DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=None)
```

```python
In [26]: pd.to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)

→ DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]', freq=None)
```

**Warning:** You see in the above example that `dayfirst` isn’t strict, so if a date can’t be parsed with the day being first it will be parsed as if `dayfirst` were False.

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.

```python
In [27]: pd.to_datetime('2010/11/12')
Out[27]: Timestamp('2010-11-12 00:00:00')
```

```python
In [28]: pd.Timestamp('2010/11/12')

→ Timestamp('2010-11-12 00:00:00')
```

19.3.1 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 [29]: pd.to_datetime('2010/11/12', format='%Y/%m/%d')
Out[29]: Timestamp('2010-11-12 00:00:00')
```

```python
In [30]: pd.to_datetime('12-11-2010 00:00', format='%d-%m-%Y %H:%M')

→ Timestamp('2010-11-12 00:00:00')
```

For more information on how to specify the format options, see https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.
19.3.2 Assembling Datetime from Multiple DataFrame Columns

New in version 0.18.1.

You can also pass a DataFrame of integer or string columns to assemble into a Series of Timestamps.

```python
In [31]: df = pd.DataFrame({'year': [2015, 2016],
                          'month': [2, 3],
                          'day': [4, 5],
                          'hour': [2, 3]})

In [32]: pd.to_datetime(df)
Out[32]:
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 [33]: pd.to_datetime(df[['year', 'month', 'day']])
Out[33]:
0 2015-02-04
1 2016-03-05

dtype: datetime64[ns]
```

d.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

19.3.3 Invalid Data

**Note:** In version 0.17.0, the default for to_datetime is now errors='raise', rather than errors='ignore'. This means that invalid parsing will raise rather that return the original input as in previous versions.

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 [34]: pd.to_datetime(['2009/07/31', 'asd'], errors='ignore')
Out[34]: array(['2009/07/31', 'asd'], dtype=object)
```

Pass errors='coerce' to convert unparsable data to NaT (not a time):

```python
In [35]: pd.to_datetime(['2009/07/31', 'asd'], errors='coerce')
Out[35]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```
19.3.4 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.

```
In [36]: pd.to_datetime([1349720105, 1349806505, 1349892905,
                     ....: 1349979305, 1350065705], unit='s')
Out[36]:
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)

In [37]: pd.to_datetime([1349720105100, 1349720105200, 1349720105300,
                     ....: 1349720105400, 1349720105500], unit='ms')
Out[37]:
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: Epoch times will be rounded to the nearest nanosecond.

```
In [38]: pd.to_datetime([1490195805.433, 1490195805.433502912], unit='s')
Out[38]:
DatetimeIndex(['2017-03-22 15:16:45.433000', '2017-03-22 15:16:45.433503'],
               dtype='datetime64[ns]', freq=None)

In [39]: pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

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).

```
In [38]: pd.to_datetime([1490195805.433, 1490195805.433502912], unit='s')
Out[38]:
DatetimeIndex(['2017-03-22 15:16:45.433000', '2017-03-22 15:16:45.433503'],
               dtype='datetime64[ns]', freq=None)

In [39]: pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

See also:
Using the origin Parameter

19.3.5 From Timestamps to Epoch

To invert the operation from above, namely, to convert from a Timestamp to a ‘unix’ epoch:

```
In [40]: stamps = pd.date_range('2012-10-08 18:15:05', periods=4, freq='D')
In [41]: stamps
Out[41]:
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 convert the DatetimeIndex to an int64 array, then divide by the conversion unit.

```
In [42]: stamps.view('int64') // pd.Timedelta(1, unit='s')
Out[42]: array([1349720105, 1349806505, 1349892905, 1349979305])
```

19.3.6 Using the origin Parameter

New in version 0.20.0.

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:

```
In [43]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[43]: 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.

```
In [44]: pd.to_datetime([1, 2, 3], unit='D')
Out[44]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype='datetime64[ns]', freq=None)
```

19.4 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 [45]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]
# Note the frequency information
In [46]: index = pd.DatetimeIndex(dates)
In [47]: index
Out[47]: 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 [50]: start = datetime(2011, 1, 1)
In [51]: end = datetime(2012, 1, 1)
In [52]: index = pd.date_range(start, end)
In [53]: index
Out[53]:
              '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 [54]: index = pd.bdate_range(start, end)
In [55]: index
Out[55]:
              '2011-01-13', '2011-01-14',
              ... '2011-12-19', '2011-12-20', '2011-12-21', '2011-12-22',
              '2011-12-23', '2011-12-26', '2011-12-27', '2011-12-28',
              '2011-12-29', '2011-12-30'],
dtype='datetime64[ns]', length=260, freq='B')

Convenience functions like date_range and bdate_range can utilize a variety of frequency aliases:

In [56]: pd.date_range(start, periods=1000, freq='M')
Out[56]:
              '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')

In [57]: pd.bdate_range(start, periods=250, freq='BQS')
Out[57]:
              '2011-01-13', '2011-01-14',
              ... '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:
In [58]: pd.date_range(start, end, freq='BM')
Out[58]:
            dtype='datetime64[ns]', freq='BM')

In [59]: pd.date_range(start, end, freq='W')
Out[59]:
              '2011-02-27', '2011-03-03', '2011-03-10', '2011-03-17',
              '2011-08-14', '2011-08-21', '2011-08-28', '2011-09-04',
              '2011-12-04', '2011-12-11', '2011-12-18', '2011-12-25',
              '2012-01-01'],
            dtype='datetime64[ns]', freq='W-SUN')

In [60]: pd.bdate_range(end=end, periods=20)
Out[60]:
DatetimeIndex(['2011-12-05', '2011-12-06', '2011-12-07', '2011-12-08',
              '2011-12-09', '2011-12-10', '2011-12-11', '2011-12-12',
              '2011-12-13', '2011-12-14', '2011-12-15', '2011-12-16',
              '2011-12-17', '2011-12-18', '2011-12-19', '2011-12-20',
              '2011-12-21', '2011-12-22', '2011-12-23', '2011-12-24',
              '2011-12-25', '2011-12-26', '2011-12-27', '2011-12-28',
              '2011-12-29', '2011-12-30'],
            dtype='datetime64[ns]', freq='B')

In [61]: pd.bdate_range(start=start, periods=20)
Out[61]:
              '2011-01-27', '2011-01-28'],
            dtype='datetime64[ns]', freq='B')

### 19.4.1 Custom Frequency Ranges

**Warning:** This functionality was originally exclusive to `cdate_range`, which is deprecated as of version 0.21.0 in favor of `bdate_range`. Note that `cdate_range` only utilizes the `weekmask` and `holidays` parameters when custom business day, ‘C’, is passed as the frequency string. Support has been expanded with `bdate_range` to work with any custom frequency string.

New in version 0.21.0.
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.

```python
In [62]: weekmask = 'Mon Wed Fri'

In [63]: holidays = [datetime(2011, 1, 5), datetime(2011, 3, 14)]

In [64]: pd.bdate_range(start, end, freq='C', weekmask=weekmask, holidays=holidays)
Out[64]:
               '2011-01-24', '2011-01-26',
               '2011-12-09', '2011-12-12', '2011-12-14', '2011-12-16',
               '2011-12-19', '2011-12-21', '2011-12-23', '2011-12-26',
               '2011-12-28', '2011-12-30'],
dtype='datetime64[ns]', length=154, freq='C')

In [65]: pd.bdate_range(start, end, freq='CBMS', weekmask=weekmask)
```

See also:

*Custom Business Days*

### 19.5 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:

```python
In [66]: pd.Timestamp.min
Out[66]: Timestamp('1677-09-21 00:12:43.145225')

In [67]: pd.Timestamp.max
```

See also:

*Representing Out-of-Bounds Spans*

### 19.6 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` and `tshift` 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

`Datetimerange` 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.

`Datetimerange` can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```python
In [68]: rng = pd.date_range(start, end, freq='BM')

In [69]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [70]: ts.index
                  freq='BM', dtype='datetime64[ns]')

In [71]: ts[:5].index
                   '2011-05-31'],
                  dtype='datetime64[ns]', freq='BM')

In [72]: ts[::2].index
                   '2011-09-30', '2011-11-30'],
                  dtype='datetime64[ns]', freq='2BM')
```

### 19.6.1 Partial String Indexing

Dates and strings that parse to timestamps can be passed as indexing parameters:

```python
In [73]: ts['1/31/2011']
Out[73]: -1.2812473076599531

In [74]: ts[datetime(2011, 12, 25):]
Out[74]:
2011-12-30  0.687738
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 [76]: ts['2011']
Out[76]:
2011-01-31 -1.281247
2011-02-28  0.727707
2011-03-31 -0.121306
2011-04-29  0.097883
2011-05-31  0.695775
2011-06-30  0.341734
2011-07-29  0.959726
2011-08-31 -1.110336
2011-09-30  0.619976
2011-10-31  0.149748
2011-11-30 -0.732339
2011-12-30  0.687738
Freq: BM, dtype: float64
```

```
In [77]: ts['2011-6']
...
2011-06-30  0.341734
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:

```
In [78]: dft = pd.DataFrame(randn(100000,1),
   ...:     columns=['A'],
   ...:     index=pd.date_range('20130101',periods=100000,freq='T'))
   ...
In [79]: dft
Out[79]:
   A
2013-01-01  00:00:00  0.176444
2013-01-01  00:01:00  0.403310
2013-01-01  00:02:00 -0.154951
2013-01-01  00:03:00  0.301624
2013-01-01  00:04:00 -2.179861
2013-01-01  00:05:00 -1.369748
2013-01-01  00:06:00 -0.954208
...     ...
2013-03-11  10:33:00 -0.293083
2013-03-11  10:34:00 -0.059881
2013-03-11  10:35:00  1.252450
2013-03-11  10:36:00  0.046611
2013-03-11  10:37:00  0.059478
2013-03-11  10:38:00 -0.286539
2013-03-11  10:39:00  0.841669
```
In [80]: dft['2013']

Out[80]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>0.176444</td>
</tr>
<tr>
<td>2013-01-01 00:01:00</td>
<td>0.403310</td>
</tr>
<tr>
<td>2013-01-01 00:02:00</td>
<td>-0.154951</td>
</tr>
<tr>
<td>2013-01-01 00:03:00</td>
<td>0.301624</td>
</tr>
<tr>
<td>2013-01-01 00:04:00</td>
<td>-2.179861</td>
</tr>
<tr>
<td>2013-01-01 00:05:00</td>
<td>-1.369849</td>
</tr>
<tr>
<td>2013-01-01 00:06:00</td>
<td>-0.954208</td>
</tr>
</tbody>
</table>

... ...

| 2013-03-11 10:33:00 | -0.293083 |
| 2013-03-11 10:34:00 | -0.059881 |
| 2013-03-11 10:35:00 | 1.252450 |
| 2013-03-11 10:36:00 | 0.046611 |
| 2013-03-11 10:37:00 | 0.059478 |
| 2013-03-11 10:38:00 | -0.286539 |
| 2013-03-11 10:39:00 | 0.841669 |

[100000 rows x 1 columns]

This starts on the very first time in the month, and includes the last date & time for the month

In [81]: dft['2013-1':'2013-2']

Out[81]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>0.176444</td>
</tr>
<tr>
<td>2013-01-01 00:01:00</td>
<td>0.403310</td>
</tr>
<tr>
<td>2013-01-01 00:02:00</td>
<td>-0.154951</td>
</tr>
<tr>
<td>2013-01-01 00:03:00</td>
<td>0.301624</td>
</tr>
<tr>
<td>2013-01-01 00:04:00</td>
<td>-2.179861</td>
</tr>
<tr>
<td>2013-01-01 00:05:00</td>
<td>-1.369849</td>
</tr>
<tr>
<td>2013-01-01 00:06:00</td>
<td>-0.954208</td>
</tr>
</tbody>
</table>

... ...

| 2013-02-28 23:53:00 | 0.103114 |
| 2013-02-28 23:54:00 | -1.303422 |
| 2013-02-28 23:55:00 | 0.451943 |
| 2013-02-28 23:56:00 | 0.220534 |
| 2013-02-28 23:57:00 | -1.624220 |
| 2013-02-28 23:58:00 | 0.093915 |
| 2013-02-28 23:59:00 | -1.087454 |

[84960 rows x 1 columns]

This specifies a stop time that includes all of the times on the last day

In [82]: dft['2013-1':'2013-2-28']

Out[82]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>0.176444</td>
</tr>
<tr>
<td>2013-01-01 00:01:00</td>
<td>0.403310</td>
</tr>
<tr>
<td>2013-01-01 00:02:00</td>
<td>-0.154951</td>
</tr>
<tr>
<td>2013-01-01 00:03:00</td>
<td>0.301624</td>
</tr>
<tr>
<td>2013-01-01 00:04:00</td>
<td>-2.179861</td>
</tr>
<tr>
<td>2013-01-01 00:05:00</td>
<td>-1.369849</td>
</tr>
</tbody>
</table>

[100000 rows x 1 columns]

19.6. Indexing
This specifies an exact stop time (and is not the same as the above)

```python
In [83]: dft['2013-1':'2013-2-28 00:00:00']
Out[83]:
            A
2013-01-01 00:00:00  0.176444
2013-01-01 00:01:00  0.403310
2013-01-01 00:02:00 -0.154951
2013-01-01 00:03:00  0.301624
2013-01-01 00:04:00 -2.179861
2013-01-01 00:05:00 -1.369849
2013-01-01 00:06:00 -0.954208
... ... 
2013-02-27 23:54:00  0.897051
2013-02-27 23:55:00 -0.309230
2013-02-27 23:56:00  1.944713
2013-02-27 23:57:00  0.369265
2013-02-27 23:58:00  0.053071
2013-02-27 23:59:00 -0.019734
2013-02-28 00:00:00  1.388189

[83521 rows x 1 columns]
```

We are stopping on the included end-point as it is part of the index

```python
In [84]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[84]:
            A
2013-01-15 00:00:00  0.501288
2013-01-15 00:01:00 -0.605198
2013-01-15 00:02:00  0.215146
2013-01-15 00:03:00  0.924732
2013-01-15 00:04:00 -2.228519
2013-01-15 00:05:00  1.517331
2013-01-15 00:06:00 -1.188774
... ... 
2013-01-15 12:24:00  1.358314
2013-01-15 12:25:00 -0.737727
2013-01-15 12:26:00  1.838323
2013-01-15 12:27:00 -0.774090
2013-01-15 12:28:00  0.622261
2013-01-15 12:29:00 -0.631649
2013-01-15 12:30:00  0.193284

[751 rows x 1 columns]
```
New in version 0.18.0.

```
In [85]: 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 [86]: dft2
```

```
Out[86]:
          A
2013-01-01 00:00:00 a  -0.659574
     b      1.494522
2013-01-01 12:00:00 a  -0.778425
     b      0.253355
2013-01-02 00:00:00 a  -2.816159
     b     -1.210929
2013-01-02 12:00:00 a   0.144669
     b     -0.607471
... ...
2013-01-04 00:00:00 b   1.624463
2013-01-04 12:00:00 a   0.056912
     b     0.149867
2013-01-05 00:00:00 a  -1.256173
     b     2.324544
2013-01-05 12:00:00 a  -1.067396
     b     0.660996

[20 rows x 1 columns]
```

```
In [87]: dft2.loc['2013-01-05']
```

```
     A
2013-01-05 00:00:00 a  -1.256173
     b     2.324544
2013-01-05 12:00:00 a  -1.067396
     b     0.660996
```

```
In [88]: idx = pd.IndexSlice

In [89]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [90]: dft2.loc[idx[:, '2013-01-05'], :]
```

```
Out[90]:
          A
     a 2013-01-05 00:00:00  -1.256173
          b 2013-01-05 12:00:00  -1.067396
     b 2013-01-05 00:00:00   2.324544
          b 2013-01-05 12:00:00  -0.660996
```

19.6. Indexing
19.6.2 Slice vs. Exact Match

Changed in version 0.20.0.

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:

```python
In [91]: 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 [92]: series_minute.index.resolution
Out[92]: 'minute'

```

A timestamp string less accurate than a minute gives a `Series` object.

```python
In [93]: series_minute['2011-12-31 23']
Out[93]:
2011-12-31 23:59:00    1
dtype: int64
```

A timestamp string with minute resolution (or more accurate), gives a scalar instead, i.e. it is not casted to a slice.

```python
In [94]: series_minute['2011-12-31 23:59']
Out[94]: 1

In [95]: series_minute['2011-12-31 23:59:00']
Out[95]: 1
```

If index resolution is second, then, the minute-accurate timestamp gives a `Series`.

```python
In [96]: 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 [97]: series_second.index.resolution
Out[97]: 'second'

In [98]: series_second['2011-12-31 23:59']
Out[98]:
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 [] as well.

```python
In [99]: dft_minute = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]},
                                index=series_minute.index)

In [100]: dft_minute['2011-12-31 23']
Out[100]:
```

852 Chapter 19. Time Series / Date functionality
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 [101]: dft_minute.loc['2011-12-31 23:59']
Out[101]:
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 [102]: series_monthly = pd.Series([1, 2, 3],
                           pd.DatetimeIndex(['2011-12',
                                            '2012-01',
                                            '2012-02']))

In [103]: series_monthly.index.resolution
Out[103]: 'day'

In [104]: series_monthly['2011-12'] # returns Series

Out[104]:
2011-12-01    1
Name: 2011-12-01, dtype: int64
```

19.6.3 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 [105]: dft[datetime(2013, 1, 1):datetime(2013,2,28)]
Out[105]:
          A
2013-01-01  00:00:00  0.176444
2013-01-01  00:01:00  0.403310
2013-01-01  00:02:00  0.154951
2013-01-01  00:03:00  0.301624
2013-01-01  00:04:00 -2.179861
2013-01-01  00:05:00 -0.954208
2013-01-01  00:06:00 -0.954208
        ...   ...
2013-02-27  23:54:00  0.964820
```

19.6. Indexing
With no defaults.

```
In [106]: dft[datetime(2013, 1, 1, 10, 12, 0):datetime(2013, 2, 28, 10, 12, 0)]
```

```
Out[106]:
          A
2013-01-01 10:12:00 -0.246733  
2013-01-01 10:13:00 -1.429225  
2013-01-01 10:14:00 -1.265339  
2013-01-01 10:15:00  0.710986   
2013-01-01 10:16:00  0.818200   
2013-01-01 10:17:00  0.543542   
2013-01-01 10:18:00  1.577713   
... ...                                                     
2013-02-28 10:06:00  0.311249   
2013-02-28 10:07:00  2.366080   
2013-02-28 10:08:00 -0.490372   
2013-02-28 10:09:00  0.373340   
2013-02-28 10:10:00  0.638442   
2013-02-28 10:11:00  1.330135   
2013-02-28 10:12:00 -0.945450   

[83521 rows x 1 columns]
```

### 19.6.4 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:

```
In [107]: rng2 = pd.date_range('2011-01-01', '2012-01-01', freq='W')

In [108]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)

In [109]: ts2.truncate(before='2011-11', after='2011-12')
```

```
Out[109]:
          2011-11-06  -0.773743
2011-11-13   0.247216
2011-11-20   0.591308
2011-11-27   2.228500
Freq: W-SUN, dtype: float64
```

```
In [110]: ts2['2011-11':'2011-12']
```

```
  →
2011-11-06  -0.773743
2011-11-13   0.247216
2011-11-20   0.591308
```

854 Chapter 19. Time Series / Date functionality
Even complicated fancy indexing that breaks the DatetimeIndex frequency regularity will result in a DatetimeIndex, although frequency is lost:

```python
In [111]: ts2[[0, 2, 6]].index
Out[111]: DatetimeIndex(['2011-01-02', '2011-01-16', '2011-02-13'], dtype='datetime64[ns]', freq=None)
```

### 19.7 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>dayofyear</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>weekday</td>
<td>The number of the day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>weekday_name</td>
<td>The name of the day in a week (ex: Friday)</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, see the docs.
19.8 DateOffset Objects

In the preceding examples, we created DatetimeIndex objects at various frequencies by passing in frequency strings like ‘M’, ‘W’, and ‘BM to the freq keyword. Under the hood, these frequency strings are being translated into an instance of pandas DateOffset, which represents a regular frequency increment. Specific offset logic like “month”, “business day”, or “one hour” is represented in its various subclasses.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateOffset</td>
<td>Generic offset class, defaults to 1 calendar day</td>
</tr>
<tr>
<td>BDay</td>
<td>business day (weekday)</td>
</tr>
<tr>
<td>CDay</td>
<td>custom business day</td>
</tr>
<tr>
<td>Week</td>
<td>one week, optionally anchored on a day of the week</td>
</tr>
<tr>
<td>WeekOfMonth</td>
<td>the x-th day of the y-th week of each month</td>
</tr>
<tr>
<td>LastWeekOfMonth</td>
<td>the x-th day of the last week of each month</td>
</tr>
<tr>
<td>MonthEnd</td>
<td>calendar month end</td>
</tr>
<tr>
<td>MonthBegin</td>
<td>calendar month begin</td>
</tr>
<tr>
<td>BMonthEnd</td>
<td>business month end</td>
</tr>
<tr>
<td>BMonthBegin</td>
<td>business month begin</td>
</tr>
<tr>
<td>CMonthEnd</td>
<td>custom business month end</td>
</tr>
<tr>
<td>CMonthBegin</td>
<td>custom business month begin</td>
</tr>
<tr>
<td>SemiMonthEnd</td>
<td>15th (or other day_of_month) and calendar month end</td>
</tr>
<tr>
<td>SemiMonthBegin</td>
<td>15th (or other day_of_month) and calendar month begin</td>
</tr>
<tr>
<td>QuarterEnd</td>
<td>calendar quarter end</td>
</tr>
<tr>
<td>QuarterBegin</td>
<td>calendar quarter begin</td>
</tr>
<tr>
<td>BQuarterEnd</td>
<td>business quarter end</td>
</tr>
<tr>
<td>BQuarterBegin</td>
<td>business quarter begin</td>
</tr>
<tr>
<td>FY5253Quarter</td>
<td>retail (aka 52-53 week) quarter</td>
</tr>
<tr>
<td>YearEnd</td>
<td>calendar year end</td>
</tr>
<tr>
<td>YearBegin</td>
<td>calendar year begin</td>
</tr>
<tr>
<td>BYearEnd</td>
<td>business year end</td>
</tr>
<tr>
<td>BYearBegin</td>
<td>business year begin</td>
</tr>
<tr>
<td>FY5253</td>
<td>retail (aka 52-53 week) year</td>
</tr>
<tr>
<td>BusinessHour</td>
<td>business hour</td>
</tr>
<tr>
<td>CustomBusinessHour</td>
<td>custom business hour</td>
</tr>
<tr>
<td>Hour</td>
<td>one hour</td>
</tr>
<tr>
<td>Minute</td>
<td>one minute</td>
</tr>
<tr>
<td>Second</td>
<td>one second</td>
</tr>
<tr>
<td>Milli</td>
<td>one millisecond</td>
</tr>
<tr>
<td>Micro</td>
<td>one microsecond</td>
</tr>
<tr>
<td>Nano</td>
<td>one nanosecond</td>
</tr>
</tbody>
</table>

The basic DateOffset takes the same arguments as dateutil.relativedelta, which works like:

```
In [112]: d = datetime(2008, 8, 18, 9, 0)

In [113]: d + relativedelta(months=4, days=5)  
Out[113]: datetime.datetime(2008, 12, 23, 9, 0)
```

We could have done the same thing with DateOffset:

```
In [114]: from pandas.tseries.offsets import *
```
The key features of a `DateOffset` object are:

- it can be added / subtracted to/from a datetime object to obtain a shifted date
- it can be multiplied by an integer (positive or negative) so that the increment will be applied multiple times
- it has `rollforward` and `rollback` methods for moving a date forward or backward to the next or previous “offset date”

Subclasses of `DateOffset` define the `apply` function which dictates custom date increment logic, such as adding business days:

```python
class BDay(DateOffset):
    """DateOffset increments between business days""
    def apply(self, other):
        ...
```

The `rollforward` and `rollback` methods do exactly what you would expect:

```python
In [116]: d - 5 * BDay()
Out[116]: Timestamp('2008-08-11 09:00:00')
In [117]: d + BMonthEnd()
Out[117]: Timestamp('2008-08-29 09:00:00')
```

It’s definitely worth exploring the `pandas.tseries.offsets` module and the various docstrings for the classes. These operations (`apply`, `rollforward` and `rollback`) preserves time (hour, minute, etc) information by default. To reset time, use `normalize=True` keyword when creating the offset instance. If `normalize=True`, result is normalized after the function is applied.

```python
In [122]: day = Day()
In [123]: day.apply(pd.Timestamp('2014-01-01 09:00'))
Out[123]: Timestamp('2014-01-02 09:00:00')
In [124]: day = Day(normalize=True)
In [125]: day.apply(pd.Timestamp('2014-01-01 09:00'))
Out[125]: Timestamp('2014-01-02 00:00:00')
In [126]: hour = Hour()
In [127]: hour.apply(pd.Timestamp('2014-01-01 22:00'))
Out[127]: Timestamp('2014-01-01 23:00:00')
```
In [128]: hour = Hour(normalize=True)

In [129]: hour.apply(pd.Timestamp('2014-01-01 22:00'))
Out[129]: Timestamp('2014-01-01 00:00:00')

In [130]: hour.apply(pd.Timestamp('2014-01-01 23:00'))
Out[130]: Timestamp('2014-01-02 00:00:00')

19.8.1 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 [131]: d
Out[131]: datetime.datetime(2008, 8, 18, 9, 0)

In [132]: d + Week()
Out[132]: Timestamp('2008-08-25 09:00:00')

In [133]: d + Week(weekday=4)
Out[133]: Timestamp('2008-08-22 09:00:00')

In [134]: (d + Week(weekday=4)).weekday()
Out[134]: 4

In [135]: d - Week()
Out[135]: Timestamp('2008-08-11 09:00:00')

normalize option will be effective for addition and subtraction.

In [136]: d + Week(normalize=True)
Out[136]: Timestamp('2008-08-25 00:00:00')

In [137]: d - Week(normalize=True)
Out[137]: Timestamp('2008-08-11 00:00:00')

Another example is parameterizing YearEnd with the specific ending month:

In [138]: d + YearEnd()
Out[138]: Timestamp('2008-12-31 09:00:00')

In [139]: d + YearEnd(month=6)
Out[139]: Timestamp('2009-06-30 09:00:00')

19.8.2 Using Offsets with Series / DatetimeIndex

Offsets can be used with either a Series or DatetimeIndex to apply the offset to each element.
In [140]: `rng = pd.date_range('2012-01-01', '2012-01-03')`

In [141]: `s = pd.Series(rng)`

In [142]: `rng`

Out[142]: DatetimeIndex(['2012-01-01', '2012-01-02', '2012-01-03'], dtype='datetime64[ns]', freq='D')

In [143]: `rng + DateOffset(months=2)`

Out[143]: DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype='datetime64[ns]', freq='D')

In [144]: `s + DateOffset(months=2)`

Out[144]:
0 2012-03-01
1 2012-03-02
2 2012-03-03
dtype: datetime64[ns]

In [145]: `s - DateOffset(months=2)`

Out[145]:
0 2011-11-01
1 2011-11-02
2 2011-11-03
dtype: datetime64[ns]

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 [146]: `s - Day(2)`

Out[146]:
0 2011-12-30
1 2011-12-31
2 2012-01-01
dtype: datetime64[ns]

In [147]: `td = s - pd.Series(pd.date_range('2011-12-29', '2011-12-31'))`

In [148]: `td`

Out[148]:
0 3 days
1 3 days
2 3 days
dtype: timedelta64[ns]

In [149]: `td + Minute(15)`

Out[149]:
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

19.8. DateOffset Objects
In [150]: rng + BQuarterEnd()
Out[150]: DatetimeIndex(['2012-03-30', '2012-03-30', '2012-03-30'], dtype='datetime64[ns]', freq=None)

19.8.3 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 [151]: from pandas.tseries.offsets import CustomBusinessDay
In [152]: weekmask_egypt = 'Sun Mon Tue Wed Thu'
    # They also observe International Workers' Day so let's
    # add that for a couple of years
In [153]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]
In [154]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)
In [155]: dt = datetime(2013, 4, 30)
In [156]: dt + 2 * bday_egypt
Out[156]: Timestamp('2013-05-05 00:00:00')

Let’s map to the weekday names

In [157]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)
In [158]: pd.Series(dts.weekday, dts).map(pd.Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
Out[158]:
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 [159]: from pandas.tseries.holiday import USFederalHolidayCalendar
In [160]: bday_us = CustomBusinessDay(calendar=USFederalHolidayCalendar())
    # Friday before MLK Day
In [161]: dt = datetime(2014, 1, 17)
    # Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [162]: dt + bday_us
Out[162]: Timestamp('2014-01-21 00:00:00')

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.
In [163]: from pandas.tseries.offsets import CustomBusinessMonthBegin

In [164]: bmth_us = CustomBusinessMonthBegin(calendar=USFederalHolidayCalendar())

# Skip new years
In [165]: dt = datetime(2013, 12, 17)

In [166]: dt + bmth_us
Out[166]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [167]: pd.DatetimeIndex(start='20100101', end='20120101', freq=bmth_us)

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.

19.8.4 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. 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, remaining is added to the next business day.

In [168]: bh = BusinessHour()

In [169]: bh
Out[169]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [170]: pd.Timestamp('2014-08-01 10:00').weekday()
Out[170]: 4

In [171]: pd.Timestamp('2014-08-01 10:00') + bh
Out[171]: Timestamp('2014-08-01 11:00:00')

# Below example is the same as: pd.Timestamp('2014-08-01 09:00') + bh
In [172]: pd.Timestamp('2014-08-01 08:00') + bh
Out[172]: Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
Also, you can specify start and end time by keywords. Argument must be str which has hour:minute representation or datetime.time instance. Specifying seconds, microseconds and nanoseconds as business hour results in ValueError.

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.
Although 2014-08-04 is Monday, it is out of business hours because it starts from 08-03 (Sunday).

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.

BusinessHour regards Saturday and Sunday as holidays. To use arbitrary holidays, you can use `CustomBusinessHour` offset, see **Custom Business Hour**.

### 19.8.5 Custom Business Hour

New in version 0.18.1.

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.

```python
from pandas.tseries.holiday import USFederalHolidayCalendar
bhour_us = CustomBusinessHour(calendar=USFederalHolidayCalendar())
# Friday before MLK Day
dt = datetime(2014, 1, 17, 15)
```
You can use keyword arguments supported by either `BusinessHour` and `CustomBusinessDay`.

```python
In [198]: bhour_mon = CustomBusinessHour(start='10:00', weekmask='Tue Wed Thu Fri')
# Monday is skipped because it's a holiday, business hour starts from 10:00
In [199]: dt + bhour_mon * 2
Out[199]: Timestamp('2014-01-21 10:00:00')
```

### 19.8.6 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>milliseconds</td>
</tr>
<tr>
<td>U, us</td>
<td>microseconds</td>
</tr>
<tr>
<td>N</td>
<td>nanoseconds</td>
</tr>
</tbody>
</table>

### 19.8.7 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:
In [200]: pd.date_range(start, periods=5, freq='B')
Out[200]:
              '2011-01-07'],
       dtype='datetime64[ns]', freq='B')

In [201]: pd.date_range(start, periods=5, freq=BDay())
Out[201]:
              '2011-01-07'],
       dtype='datetime64[ns]', freq='B')

You can combine together day and intraday offsets:

In [202]: pd.date_range(start, periods=10, freq='2h20min')
Out[202]:
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 [203]: pd.date_range(start, periods=10, freq='1D10U')
Out[203]:
DatetimeIndex(['2011-01-01 00:00:00.000010', '2011-01-02 00:00:00.000020',
               '2011-01-03 00:00:00.000030', '2011-01-04 00:00:00.000040',
               '2011-01-05 00:00:00.000050', '2011-01-06 00:00:00.000060',
               '2011-01-07 00:00:00.000070', '2011-01-08 00:00:00.000080',
               '2011-01-09 00:00:00.000090'],
       dtype='datetime64[ns]', freq='86400000010U')

19.8.8 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>
</tbody>
</table>

Continued on next page
Table 19.2 – continued from previous page

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<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.

### 19.8.9 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.

```python
In [204]: pd.Timestamp('2014-01-02') + MonthBegin(n=1)  
Out[204]: Timestamp('2014-02-01 00:00:00')

In [205]: pd.Timestamp('2014-01-02') + MonthEnd(n=1)  
\n\n\n\nOut[205]: Timestamp('2014-01-31 00:00:00')

In [206]: pd.Timestamp('2014-01-02') - MonthBegin(n=1)  
\n\n\n\nOut[206]: Timestamp('2014-01-01 00:00:00')

In [207]: pd.Timestamp('2014-01-02') - MonthEnd(n=1)  
\n\n\n\nOut[207]: Timestamp('2013-12-31 00:00:00')

In [208]: pd.Timestamp('2014-01-02') + MonthBegin(n=4)  
\n\n\n\nOut[208]: Timestamp('2014-05-01 00:00:00')

In [209]: pd.Timestamp('2014-01-02') - MonthBegin(n=4)  
\n\n\n\nOut[209]: Timestamp('2013-01-01 00:00:00')
```

If the given date is on an anchor point, it is moved \(|n|\) points forwards or backwards.
In [210]: pd.Timestamp('2014-01-01') + MonthBegin(n=1)
Out[210]: Timestamp('2014-02-01 00:00:00')

In [211]: pd.Timestamp('2014-01-31') + MonthEnd(n=1)
Out[211]: Timestamp('2014-02-28 00:00:00')

In [212]: pd.Timestamp('2014-01-01') - MonthBegin(n=1)
Out[212]: Timestamp('2013-12-01 00:00:00')

In [213]: pd.Timestamp('2014-01-31') - MonthEnd(n=1)
Out[213]: Timestamp('2013-12-31 00:00:00')

In [214]: pd.Timestamp('2014-01-01') + MonthBegin(n=4)
Out[214]: Timestamp('2014-05-01 00:00:00')

In [215]: pd.Timestamp('2014-01-31') + MonthEnd(n=4)
Out[215]: Timestamp('2014-01-31 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 [216]: pd.Timestamp('2014-01-02') + MonthBegin(n=0)
Out[216]: Timestamp('2014-02-01 00:00:00')

In [217]: pd.Timestamp('2014-01-02') + MonthEnd(n=0)
Out[217]: Timestamp('2014-01-31 00:00:00')

In [218]: pd.Timestamp('2014-01-01') + MonthBegin(n=0)
Out[218]: Timestamp('2014-01-01 00:00:00')

In [219]: pd.Timestamp('2014-01-31') + MonthEnd(n=0)
Out[219]: Timestamp('2014-01-31 00:00:00')

19.8.10 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. Further, 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:
pandas: powerful Python data analysis toolkit, Release 0.21.0

<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>move 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:

```python
In [220]: from pandas.tseries.holiday import Holiday, USMemorialDay, AbstractHolidayCalendar, nearest_workday, MO

In [221]: class ExampleCalendar(AbstractHolidayCalendar):
    ...:     rules = [
    ...:         USMemorialDay,
    ...:         Holiday('July 4th', month=7, day=4, observance=nearest_workday),
    ...:         Holiday('Columbus Day', month=10, day=1,
    ...:              offset=DateOffset(weekday=MO(2))),  # same as 2*Week(weekday=2)
    ...:     ]

In [222]: cal = ExampleCalendar()

In [223]: cal.holidays(datetime(2012, 1, 1), datetime(2012, 12, 31))
Out[223]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype='datetime64[ns]', freq=None)
```

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.

```python
In [224]: from pandas.tseries.offsets import CDay

In [225]: pd.DatetimeIndex(start='7/1/2012', end='7/10/2012', freq=CDay(calendar=cal)).to_pydatetime()
Out[225]: array([datetime.datetime(2012, 7, 2, 0, 0),
                      datetime.datetime(2012, 7, 3, 0, 0),
                      datetime.datetime(2012, 7, 5, 0, 0),
                      datetime.datetime(2012, 7, 6, 0, 0),
                      datetime.datetime(2012, 7, 9, 0, 0),
                      datetime.datetime(2012, 7, 10, 0, 0)], dtype=object)

In [226]: offset = CustomBusinessDay(calendar=cal)

In [227]: datetime(2012, 5, 25) + offset
Out[227]: Timestamp('2012-05-29 00:00:00')

In [228]: datetime(2012, 7, 3) + offset
Out[228]: Timestamp('2012-07-05 00:00:00')

In [229]: datetime(2012, 7, 3) + 2 * offset
Out[229]: Timestamp('2012-07-06 00:00:00')

In [230]: datetime(2012, 7, 6) + offset
```
Ranges are defined by the `start_date` and `end_date` class attributes of `AbstractHolidayCalendar`. The defaults are below.

```python
In [231]: AbstractHolidayCalendar.start_date
Out[231]: Timestamp('1970-01-01 00:00:00')

In [232]: AbstractHolidayCalendar.end_date
Out[232]: Timestamp('2030-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as datetime/Timestamp/string.

```python
In [233]: AbstractHolidayCalendar.start_date = datetime(2012, 1, 1)
In [234]: AbstractHolidayCalendar.end_date = datetime(2012, 12, 31)
```

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 [236]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory,

In [237]: cal = get_calendar('ExampleCalendar')
```

```python
In [238]: cal.rules
Out[238]: [Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: kwds={'weekday': MO(-1)}>),
Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x131922e18>),
Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: kwds={'weekday': MO(+2)}>)]
```

```python
In [239]: new_cal = HolidayCalendarFactory('NewExampleCalendar', cal, USLaborDay)
```

```python
In [240]: new_cal.rules
Out[240]: [Holiday: Labor Day (month=9, day=1, offset=<DateOffset: kwds={'weekday': MO(+1)}>),
Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: kwds={'weekday': MO(-1)}>),
Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x131922e18>),
Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: kwds={'weekday': MO(+2)}>)]
```
19.9 Time Series-Related Instance Methods

19.9.1 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.

```
In [241]: ts = ts[:5]
In [242]: ts.shift(1)
Out[242]:
2011-01-31     NaN
2011-02-28  -1.281247
2011-03-31  -0.727707
2011-04-29  -0.121306
2011-05-31  -0.097883
Freq: BM, dtype: float64
```

The shift method accepts an freq argument which can accept a DateOffset class or other timedelta-like object or also a offset alias:

```
In [243]: ts.shift(5, freq=offsets.BDay())
Out[243]:
2011-02-07  -1.281247
2011-03-07  -0.727707
2011-04-07  -0.121306
2011-05-06  -0.097883
2011-06-07   0.695775
dtype: float64
```

```
In [244]: ts.shift(5, freq='BM')
Out[244]:
2011-06-30  -1.281247
2011-07-29  -0.727707
2011-08-31  -0.121306
2011-09-30  -0.097883
2011-10-31   0.695775
Freq: BM, dtype: float64
```

Rather than changing the alignment of the data and the index, DataFrame and Series objects also have a tshift convenience method that changes all the dates in the index by a specified number of offsets:

```
In [245]: ts.tshift(5, freq='D')
Out[245]:
2011-02-05  -1.281247
2011-03-05  -0.727707
2011-04-05  -0.121306
2011-05-04  -0.097883
2011-06-05   0.695775
dtype: float64
```

Note that with tshift, the leading entry is no longer NaN because the data is not being realigned.
19.9.2 Frequency Conversion

The primary function for changing frequencies is the `asfreq` function. For a `DatetimeIndex`, this is basically just a thin, but convenient wrapper around `reindex` which generates a `date_range` and calls `reindex`.

```
In [246]: dr = pd.date_range('1/1/2010', periods=3, freq=3 * offsets.BDay())
In [247]: ts = pd.Series(randn(3), index=dr)
In [248]: ts
Out[248]:
2010-01-01  0.155932
2010-01-06  1.486218
2010-01-11 -2.148675
Freq: 3B, dtype: float64
```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion.

```
In [249]: ts.asfreq(BDay())
Out[249]:
   2010-01-01  0.155932
   2010-01-04   NaN
   2010-01-05   NaN
   2010-01-06  1.486218
   2010-01-07   NaN
   2010-01-08   NaN
   2010-01-11 -2.148675
Freq: B, dtype: float64
```

```
In [250]: ts.asfreq(BDay(), method='pad')
Out[250]:
   2010-01-01  0.155932
   2010-01-04  0.155932
   2010-01-05  0.155932
   2010-01-06  1.486218
   2010-01-07  1.486218
   2010-01-08  1.486218
   2010-01-11 -2.148675
Freq: B, dtype: float64
```

19.9.3 Filling Forward / Backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the `missing data section`.

19.9.4 Converting to Python Datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.
19.10 Resampling

**Warning:** The interface to `.resample` has changed in 0.18.0 to be more groupby-like and hence more flexible. See the *whatsnew docs* for a comparison with prior versions.

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.

Starting in version 0.18.1, the `resample()` function can be used directly from `DataFrameGroupBy` objects, see the *groupby docs*.

**Note:** `.resample()` is similar to using a `.rolling()` operation with a time-based offset, see a discussion *here*.

### 19.10.1 Basics

```python
In [251]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [252]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [253]: ts.resample('5Min').sum()
Out[253]:
2012-01-01    25653
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 [254]: ts.resample('5Min').mean()
Out[254]:
2012-01-01    256.53
Freq: 5T, dtype: float64
```

```python
In [255]: ts.resample('5Min').ohlc()
Out[255]:
open  high  low  close
2012-01-01  296  496   6  449
```

```python
In [256]: ts.resample('5Min').max()
```

```
2012-01-01    496
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` and `loffset` are used to manipulate the resulting labels. `label` specifies whether the result is labeled with the beginning or the end of the interval. `loffset` performs a time adjustment on the output labels.

```python
In [259]: ts.resample('5Min').mean()  # by default label='left'
Out[259]:
2012-01-01 256.53
Freq: 5T, dtype: float64
```

```python
In [261]: ts.resample('5Min', label='left', loffset='1s').mean()
Out[261]:
2012-01-01 00:00:01 256.53
dtype: float64
```

Note: 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’.

```python
In [262]: rng2 = pd.date_range('1/1/2012', end='3/31/2012', freq='D')
In [263]: ts2 = pd.Series(range(len(rng2)), index=rng2)
```

```python
# default: label='right', closed='right'
In [264]: ts2.resample('M').max()
Out[264]:
2012-01-31 30
2012-02-29 59
2012-03-31 90
Freq: M, dtype: int64
```

```python
# default: label='left', closed='left'
In [265]: ts2.resample('SM').max()
Out[265]:
2011-12-31 13
2012-01-15 29
2012-01-31 44
2012-02-15 58
2012-02-29 73
2012-03-15 89
2012-03-31 90
```

19.10. Resampling
The `axis` parameter can be set to 0 or 1 and allows you to resample the specified axis for a DataFrame. The `kind` parameter 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. The `convention` parameter 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.

### 19.10.2 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 [267]: ts[:2].resample('250L').asfreq()
Out[267]:
2012-01-01 00:00:00.000   296.0
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   199.0
Freq: 250L, dtype: float64
```

```
In [268]: ts[:2].resample('250L').ffill()
Out[268]:
2012-01-01 00:00:00.000   296
2012-01-01 00:00:00.250   296
2012-01-01 00:00:00.500   296
2012-01-01 00:00:00.750   296
2012-01-01 00:00:01.000   199
Freq: 250L, dtype: int64
```

```
In [269]: ts[:2].resample('250L').ffill(limit=2)
Out[269]:
2012-01-01 00:00:00.000   296.0
2012-01-01 00:00:00.250   296.0
2012-01-01 00:00:00.500   296.0
2012-01-01 00:00:00.750       NaN
2012-01-01 00:00:01.000   199.0
Freq: 250L, dtype: float64
```
19.10.3 Sparse Resampling

Sparse timeseries are 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.

```python
In [270]: rng = pd.date_range('2014-1-1', periods=100, freq='D') + pd.Timedelta('1s')
In [271]: ts = pd.Series(range(100), index=rng)

If we want to resample to the full range of the series

```python
In [272]: ts.resample('3T').sum()
Out[272]:
2014-01-01 00:00:00    0.0
2014-01-01 00:03:00    NaN
2014-01-01 00:06:00    NaN
2014-01-01 00:09:00    NaN
2014-01-01 00:12:00    NaN
2014-01-01 00:15:00    NaN
2014-01-01 00:18:00    NaN
   ...               ...
2014-04-09 23:42:00    NaN
2014-04-09 23:45:00    NaN
2014-04-09 23:48:00    NaN
2014-04-09 23:51:00    NaN
2014-04-09 23:54:00    NaN
2014-04-09 23:57:00    NaN
2014-04-10 00:00:00    99.0
Freq: 3T, Length: 47521, dtype: float64
```

We can instead only resample those groups where we have points as follows:

```python
In [274]: from functools import partial

In [275]: from pandas.tseries.frequencies import to_offset

In [276]: ts.groupby(partial(round, freq='3T')).sum()
Out[276]:
2014-01-01 0 0
2014-01-02 1 1
2014-01-03 2 2
2014-01-04 3 3
2014-01-05 4 4
2014-01-06 5 5
2014-01-07 6 6
   ...   ...
2014-04-04 93 93
2014-04-05 94 94
2014-04-06 95 95
```

19.10. Resampling
19.10.4 Aggregation

Similar to the aggregating API, groupby API, and the window functions API, a Resampler can be selectively resampled.

Resampling a DataFrame, the default will be to act on all columns with the same function.

In [277]: df = pd.DataFrame(np.random.randn(1000, 3),
                       index=pd.date_range('1/1/2012', freq='S', periods=1000),
                       columns=['A', 'B', 'C'])

In [278]: r = df.resample('3T')

In [279]: r.mean()

Out[279]:
          A         B         C
2012-01-01 00:00:00 -0.038580 -0.085117 -0.024750
2012-01-01 00:03:00  0.052387 -0.061477  0.029548
2012-01-01 00:06:00  0.121377 -0.010630 -0.043691
2012-01-01 00:09:00 -0.106814 -0.053819  0.097222
2012-01-01 00:12:00  0.032560  0.080543  0.167380
2012-01-01 00:15:00  0.060486 -0.057602 -0.106213

We can select a specific column or columns using standard getitem.

In [280]: r['A'].mean()

Out[280]:
           A
2012-01-01 00:00:00 -0.038580
2012-01-01 00:03:00  0.052387
2012-01-01 00:06:00  0.121377
2012-01-01 00:09:00 -0.106814
2012-01-01 00:12:00  0.032560
2012-01-01 00:15:00  0.060486
Freq: 3T, Name: A, dtype: float64

In [281]: r[['A', 'B']].mean()

    A         B
2012-01-01 00:00:00 -0.038580 -0.085117
2012-01-01 00:03:00  0.052387 -0.061477
2012-01-01 00:06:00  0.121377 -0.010630
2012-01-01 00:09:00 -0.106814 -0.053819
2012-01-01 00:12:00  0.032560  0.080543
2012-01-01 00:15:00  0.060486 -0.057602

You can pass a list or dict of functions to do aggregation with, outputting a DataFrame:
In[282]: r['A'].agg([np.sum, np.mean, np.std])
Out[282]:
                 sum       mean      std
2012-01-01 00:00:00 -6.944481 -0.038580  0.985150
2012-01-01 00:03:00  9.429707  0.052387  1.078022
2012-01-01 00:06:00 21.847876  0.121377  0.996365
2012-01-01 00:09:00 -19.226593 -0.106814  0.914070
2012-01-01 00:12:00  5.860874  0.032560  1.100055
2012-01-01 00:15:00  6.048588  0.060486  1.001532

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:

In[283]: r.agg([np.sum, np.mean])
Out[283]:
         A          B          C
       sum      mean      sum      mean      sum
2012-01-01 00:00:00 -6.944481 -0.038580 -15.320993 -0.085117 -4.454941
2012-01-01 00:03:00  9.429707  0.052387  11.065916 -0.061477  5.318688
2012-01-01 00:06:00 21.847876  0.121377 -1.913420 -0.010630 -7.864429
2012-01-01 00:09:00 -19.226593 -0.106814 -9.687468 -0.053819  17.499920
2012-01-01 00:12:00  5.860874  0.032560 14.497725  0.080543  30.128432
2012-01-01 00:15:00  6.048588  0.060486 -5.760208 -0.057602 -10.621260
         mean
2012-01-01 00:00:00 -0.024750
2012-01-01 00:03:00  0.029548
2012-01-01 00:06:00 -0.043691
2012-01-01 00:09:00  0.097222
2012-01-01 00:12:00  0.167380
2012-01-01 00:15:00 -0.106213

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

In[284]: r.agg({'A' : np.sum,
             'B' : lambda x: np.std(x, ddof=1)})
Out[284]:
         A          B
2012-01-01 00:00:00 -6.944481  1.087752
2012-01-01 00:03:00  9.429707  1.014552
2012-01-01 00:06:00 21.847876  0.954588
2012-01-01 00:09:00 -19.226593  1.027990
2012-01-01 00:12:00  5.860874  1.021503
2012-01-01 00:15:00  6.048588  1.004984

The function names can also be strings. In order for a string to be valid it must be implemented on the Resampled object:

In[285]: r.agg({'A' : 'sum', 'B' : 'std'})
Out[285]:
         A          B
2012-01-01 00:00:00 -6.944481  1.087752
2012-01-01 00:03:00  9.429707  1.014552
2012-01-01 00:06:00 21.847876  0.954588
2012-01-01 00:09:00 -19.226593  1.027990
2012-01-01 00:12:00  5.860874  1.021503
2012-01-01 00:15:00  6.048588  1.004984
Furthermore, you can also specify multiple aggregation functions for each column separately.

```python
In [286]: r.agg({'A' : ['sum','std'], 'B' : ['mean','std'] })
```

```
Out[286]:
```
```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>std</td>
<td>mean std</td>
</tr>
<tr>
<td>2012-01-01 00:00:00</td>
<td>-6.944481</td>
<td>0.985150</td>
<td>-0.085117 1.087752</td>
</tr>
<tr>
<td>2012-01-01 00:03:00</td>
<td>9.429707</td>
<td>1.078022</td>
<td>-0.061477 1.014552</td>
</tr>
<tr>
<td>2012-01-01 00:06:00</td>
<td>21.847876</td>
<td>0.996365</td>
<td>-0.010630 0.954588</td>
</tr>
<tr>
<td>2012-01-01 00:09:00</td>
<td>-19.226593</td>
<td>0.914070</td>
<td>-0.053819 1.027990</td>
</tr>
<tr>
<td>2012-01-01 00:12:00</td>
<td>5.860874</td>
<td>1.100055</td>
<td>0.080543 1.021503</td>
</tr>
<tr>
<td>2012-01-01 00:15:00</td>
<td>6.048588</td>
<td>1.001532</td>
<td>-0.057602 1.004984</td>
</tr>
</tbody>
</table>
```

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.

```python
In [287]: 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 [288]: df
```

```
Out[288]:
```
```
a   date
v  d
1 2015-01-04 0 2015-01-04
2 2015-01-11 1 2015-01-11
3 2015-01-18 2 2015-01-18
4 2015-01-25 3 2015-01-25
5 2015-02-01 4 2015-02-01
```

```python
In [289]: df.resample('M', on='date').sum()
```

```
Out[289]:
```
```
a   date
15-01-31 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.

```python
In [290]: df.resample('M', level='d').sum()
```

```
Out[290]:
```
```
a   d
2015-01-31 6
2015-02-28 4
```
19.11 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`.

19.11.1 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 [291]: pd.Period('2012', freq='A-DEC')
Out[291]: Period('2012', 'A-DEC')

In [292]: pd.Period('2012-1-1', freq='D')
Out[292]: Period('2012-01-01', 'D')

In [293]: pd.Period('2012-1-1 19:00', freq='H')
Out[293]: Period('2012-01-01 19:00', 'H')

In [294]: pd.Period('2012-1-1 19:00', freq='5H')
Out[294]: 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 [295]: p = pd.Period('2012', freq='A-DEC')

In [296]: p + 1
Out[296]: Period('2013', 'A-DEC')

In [297]: p - 3
Out[297]: Period('2009', 'A-DEC')

In [298]: p = pd.Period('2012-01', freq='2M')

In [299]: p + 2
Out[299]: Period('2012-05', '2M')

In [300]: p - 1
Out[300]: Period('2011-11', '2M')

In [301]: p == pd.Period('2012-01', freq='3M')

--------------------------------------------------------------------------
| IncompatibleFrequency Traceback (most recent call last)                 |
| <ipython-input-301-ff54ce3238f5> in <module>()                         |
| ----- 1 p == pd.Period('2012-01', freq='3M')                           |
| ~/Envs/pandas-dev/lib/python3.6/site-packages/pandas/pandas/_libs/period.pyx in _pandas._libs.period._Period.__richcmp__() |  |
| IncompatibleFrequency: Input has different freq=3M from Period(freq=2M) |
```

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.

```python
In [302]: p = pd.Period('2014-07-01 09:00', freq='H')
In [303]: p + Hour(2)
Out[303]: Period('2014-07-01 11:00', 'H')
In [304]: p + timedelta(minutes=120)
Out[304]: Period('2014-07-01 11:00', 'H')
In [305]: p + np.timedelta64(7200, 's')
Out[305]: Period('2014-07-01 11:00', 'H')
In [1]: p + Minute(5)
Traceback...
ValueError: Input has different freq from Period(freq=H)
```

If `Period` has other freqs, only the same offsets can be added. Otherwise, `ValueError` will be raised.

```python
In [306]: p = pd.Period('2014-07', freq='M')
In [307]: p + MonthEnd(3)
Out[307]: Period('2014-10', 'M')
In [1]: p + 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:

```python
Out[308]: 10
```

### 19.11.2 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:

```python
In [309]: prng = pd.period_range('1/1/2011', '1/1/2012', freq='M')
In [310]: prng
```

The `PeriodIndex` constructor can also be used directly:

```python
In [311]: pd.PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[311]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```
Passing multiplied frequency outputs a sequence of Period which has multiplied span.

```
In [312]: pd.PeriodIndex(start='2014-01', freq='3M', periods=4)
                   freq='3M')
```

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 [333]: start = pd.Period('2017Q1', freq='Q')
     end = pd.Period('2017Q2', freq='Q'), freq='M')
Out[333]: PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'], dtype='period[M]',
                     freq='M')
```

Just like DatetimeIndex, a PeriodIndex can also be used to index pandas objects:

```
In [314]: ps = pd.Series(np.random.randn(len(prng)), prng)
In [315]: ps
Out[315]:
2011-01  0.258318
2011-02 -2.503700
2011-03 -0.303053
2011-04  0.270509
2011-05  1.004841
2011-06 -0.129044
2011-07 -1.406335
2011-08 -1.310412
2011-09  0.769439
2011-10 -0.129044
2011-11  2.010541
2011-12  1.001558
2012-01 -0.087453
Freq: M, dtype: float64
```

PeriodIndex supports addition and subtraction with the same rule as Period.

```
In [316]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [317]: idx
Out[317]:
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]', freq='H')

In [318]: idx + Hour(2)
   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]', freq='H')

In [319]: idx = pd.period_range('2014-07', periods=5, freq='M')

In [320]: idx
Out[320]:
             dtype='period[M]', freq='M')
```
PeriodIndex has its own dtype named period, refer to Period Dtypes.

### 19.11.3 Period Dtypes

New in version 0.19.0.

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 [322]: pi = pd.period_range('2016-01-01', periods=3, freq='M')
In [323]: pi
Out[323]: PeriodIndex(['2016-01', '2016-02', '2016-03'], dtype='period[M]', freq='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 [325]: pi.astype('period[D]')
Out[325]: PeriodIndex(['2016-01-31', '2016-02-29', '2016-03-31'], dtype='period[D]',
                     freq='D')

# convert to DatetimeIndex
In [326]: pi.astype('datetime64[ns]')
Out[326]: DatetimeIndex(['2016-01-01', '2016-02-01', '2016-03-01'], dtype='datetime64[ns]',
                       freq='MS')

# convert to PeriodIndex
In [327]: dti = pd.date_range('2011-01-01', freq='M', periods=3)
In [328]: dti
Out[328]: DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31'], dtype=
                      'datetime64[ns]', freq='M')
In [329]: dti.astype('period[M]')
Out[329]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```

### 19.11.4 PeriodIndex Partial String Indexing

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 [330]: ps['2011-01']
Out[330]: 0.25831819727391592

In [331]: ps[datetime(2011, 12, 25):]
Out[331]:
2011-12  1.001558
2012-01  0.087453
Freq: M, dtype: float64

In [332]: ps['10/31/2011':'12/31/2011']
→ 2011-10 -0.542325
   2011-11  2.010541
   2011-12  1.001558
Freq: M, dtype: float64

Passing a string representing a lower frequency than PeriodIndex returns partial sliced data.

In [333]: ps['2011']
Out[333]:
2011-01  0.258318
2011-02  2.503700
2011-03 -0.303053
2011-04  0.270509
2011-05  1.004841
2011-06 -0.129044
2011-07 -1.406335
2011-08 -1.310412
2011-09  0.769439
2011-10 -0.542325
2011-11  2.010541
2011-12  1.001558
Freq: M, dtype: float64

In [334]: dfp = pd.DataFrame(np.random.randn(600, 1),
                   columns=['A'],
                   index=pd.period_range('2013-01-01 9:00', periods=600, freq='T'))

In [335]: dfp
Out[335]:
   A          
2013-01-01 09:00  0.005210
2013-01-01 09:01 -0.014385
2013-01-01 09:02 -0.212404
2013-01-01 09:03 -1.227760
2013-01-01 09:04 -0.809722
2013-01-01 09:05 -1.719723
2013-01-01 09:06 -0.808486
   ...   ...
2013-01-01 18:53 -0.783098
2013-01-01 18:54  0.755005
2013-01-01 18:55 -1.116732
2013-01-01 18:56 -0.940692
2013-01-01 18:57  0.228536
2013-01-01 18:58  0.109472

19.11. Time Span Representation
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 [337]: dfp['2013-01-01 10H':'2013-01-01 11H']
Out[337]:
      A
2013-01-01 10:00 -0.148998
2013-01-01 10:01  2.154810
2013-01-01 10:02 -1.605646
2013-01-01 10:03  0.021024
2013-01-01 10:04 -0.623737
2013-01-01 10:05  1.451612
2013-01-01 10:06  1.062463
    ...
2013-01-01 11:53  0.273119
2013-01-01 11:54 -0.994071
2013-01-01 11:55 -1.222179
2013-01-01 11:56 -1.167118
2013-01-01 11:57  0.262822
2013-01-01 11:58 -0.283786
2013-01-01 11:59  1.190726
[120 rows x 1 columns]
```

19.11.5 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:
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 [340]: p.asfreq('M', how='start')
Out[340]: Period('2011-01', 'M')
In [341]: p.asfreq('M', how='end')
Out[341]: Period('2011-12', 'M')
```

The shorthands 's' and 'e' are provided for convenience:

```python
In [342]: p.asfreq('M', 's')
Out[342]: Period('2011-01', 'M')
In [343]: p.asfreq('M', 'e')
Out[343]: 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 [344]: p = pd.Period('2011-12', freq='M')
In [345]: p.asfreq('A-NOV')
Out[345]: 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 define regular calendar quarters:

```python
In [346]: p = pd.Period('2012Q1', freq='Q-DEC')
In [347]: p.asfreq('D', 's')
Out[347]: Period('2012-01-01', 'D')
In [348]: p.asfreq('D', 'e')
Out[348]: Period('2012-03-31', 'D')
```

Q-MAR defines fiscal year end in March:

```python
In [349]: p = pd.Period('2011Q4', freq='Q-MAR')
In [350]: p.asfreq('D', 's')
Out[350]: Period('2011-01-01', 'D')
In [351]: p.asfreq('D', 'e')
Out[351]: Period('2011-03-31', 'D')
```
19.12 Converting Between Representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```python
In [352]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [353]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [354]: ts
Out[354]:
2012-01-31 -0.898547
2012-02-29 -1.332247
2012-03-31 -0.741645
2012-04-30 0.094321
2012-05-31 -0.438813
Freq: M, dtype: float64

In [355]: ps = ts.to_period()

In [356]: ps
Out[356]:
2012-01 -0.898547
2012-02 -1.332247
2012-03 -0.741645
2012-04 0.094321
2012-05 -0.438813
Freq: M, dtype: float64

In [357]: ps.to_timestamp()
Out[357]:
2012-01-01 -0.898547
2012-02-01 -1.332247
2012-03-01 -0.741645
2012-04-01 0.094321
2012-05-01 -0.438813
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 [358]: ps.to_timestamp('D', how='s')
Out[358]:
2012-01-01 -0.898547
2012-02-01 -1.332247
2012-03-01 -0.741645
2012-04-01 0.094321
2012-05-01 -0.438813
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:

```python
In [359]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [360]: ts = pd.Series(np.random.randn(len(prng)), prng)
```

886 Chapter 19. Time Series / Date functionality
In [361]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [362]: ts.head()
Out[362]:
1990-03-01 09:00 -0.564874
1990-06-01 09:00 -1.426510
1990-09-01 09:00 1.295437
1990-12-01 09:00 1.124017
1991-03-01 09:00 0.840428
Freq: H, dtype: float64

19.13 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.

In [363]: span = pd.period_range('1215-01-01', '1381-01-01', freq='D')
In [364]: span
Out[364]:
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, freq='D')

To convert from an int64 based YYYYMMDD representation.

In [365]: s = pd.Series([20121231, 20141130, 99991231])
In [366]: s
Out[366]:
0  20121231
1  20141130
2  99991231
dtype: int64

In [367]: def conv(x):
   ....:     return pd.Period(year = x // 10000, month = x//100 % 100,
   ....:          day = x%100, freq='D')
   ....:

In [368]: s.apply(conv)
Out[368]:
0  2012-12-31
1  2014-11-30
2  9999-12-31
dtype: object

In [369]: s.apply(conv)[2]
   "Period('9999-12-31', 'D')"
These can easily be converted to a `PeriodIndex`

```python
In [370]: span = pd.PeriodIndex(s.apply(conv))
```

```python
In [371]: span
Out[371]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='period[D]', freq='D')
```

## 19.14 Time Zone Handling

Pandas provides rich support for working with timestamps in different time zones using `pytz` and `dateutil` libraries. `dateutil` currently is only supported for fixed offset and tzfile zones. The default library is `pytz`. Support for `dateutil` is provided for compatibility with other applications e.g. if you use `dateutil` in other python packages.

### 19.14.1 Working with Time Zones

By default, pandas objects are time zone unaware:

```python
In [372]: rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')
```

```python
In [373]: rng.tz is None
Out[373]: True
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions. Dateutil time zone strings are distinguished from `pytz` time zones by starting with `dateutil/`.

- 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 timezones so there isn’t a fixed list available. For common zones, the names are the same as `pytz`.

```python
# pytz
In [374]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D', tz='Europe/London')
```

```python
# dateutil
In [375]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D', tz='dateutil/Europe/London')
```

```python
# dateutil - utc special case
In [376]: rng_utc = pd.date_range('3/6/2012 00:00', periods=10, freq='D', tz=dateutil.tz.tzutc())
```
Note that the UTC timezone is a special case in dateutil and should be constructed explicitly as an instance of dateutil.tz.tzutc. You can also construct other timezones explicitly first, which gives you more control over which time zone is used:

```python
# pytz
In [380]: tz_pytz = pytz.timezone('Europe/London')

In [381]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D', tz=tz_pytz)

In [382]: rng_pytz.tz == tz_pytz
Out[382]: True

# dateutil
In [383]: tz_dateutil = dateutil.tz.gettz('Europe/London')

In [384]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D', tz=tz_dateutil)

In [385]: rng_dateutil.tz == tz_dateutil
Out[385]: True
```

Timestamps, like Python's `datetime.datetime` object can be either time zone naive or time zone aware. Naive time series and `DatetimeIndex` objects can be localized using `tz_localize`:

```python
In [386]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [387]: ts_utc = ts.tz_localize('UTC')

In [388]: ts_utc
Out[388]:
2012-03-06 00:00:00+00:00    0.037206
2012-03-07 00:00:00+00:00    2.313998
2012-03-08 00:00:00+00:00    1.458296
2012-03-09 00:00:00+00:00   -0.620431
2012-03-10 00:00:00+00:00   -0.000111
2012-03-11 00:00:00+00:00   -0.342783
2012-03-12 00:00:00+00:00   -0.664322
2012-03-13 00:00:00+00:00    0.654814
2012-03-14 00:00:00+00:00    1.550680
2012-03-15 00:00:00+00:00    0.174511
2012-03-16 00:00:00+00:00    1.360491
2012-03-17 00:00:00+00:00    0.799737
2012-03-18 00:00:00+00:00    0.449149
2012-03-19 00:00:00+00:00    0.113146
2012-03-20 00:00:00+00:00   -0.435531
Freq: D, dtype: float64
```

Again, you can explicitly construct the timezone object first. You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```python
```
```python
In [389]: ts_utc.tz_convert('US/Eastern')
Out[389]:
2012-03-05 19:00:00-05:00    0.037206
2012-03-06 19:00:00-05:00    2.313998
2012-03-07 19:00:00-05:00    1.458296
2012-03-08 19:00:00-05:00   -0.620431
2012-03-09 19:00:00-05:00   -0.000111
2012-03-10 19:00:00-05:00   -0.342783
2012-03-11 20:00:00-04:00   -0.664322
2012-03-12 20:00:00-04:00    0.654814
2012-03-13 20:00:00-04:00    1.550680
2012-03-14 20:00:00-04:00    0.174511
2012-03-15 20:00:00-04:00    1.360491
2012-03-16 20:00:00-04:00    0.799737
2012-03-17 20:00:00-04:00    0.449149
2012-03-18 20:00:00-04:00    0.111346
2012-03-19 20:00:00-04:00   -0.435531
Freq: D, dtype: float64
```

**Warning:** Be wary of conversions between libraries. For some zones `pytz` and `dateutil` have different definitions of the zone. This is more of a problem for unusual timezones than for `standard` zones like `US/Eastern`.

**Warning:** Be aware that a timezone definition across versions of timezone 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:** It is incorrect to pass a timezone directly into the `datetime.datetime` constructor (e.g., `datetime.datetime(2011, 1, 1, tz=timezone('US/Eastern'))`). Instead, the `datetime` needs to be localized using the the `localize` method on the timezone.

Under the hood, all timestamps are stored in UTC. Scalar values from a `DatetimeIndex` with a time zone will have their fields (day, hour, minute) 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 [390]: rng_eastern = rng_utc.tz_convert('US/Eastern')
In [391]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')
In [392]: rng_eastern[5]
Out[392]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', freq='D')
In [393]: rng_berlin[5]
Out[393]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', freq='D')
Out[394]: True
```

Like `Series`, `DataFrame`, and `DatetimeIndex`, `Timestamp`'s can be converted to other
Localization of Timestamp functions just like DatetimeIndex and Series:

Operations between Series in different time zones will yield UTC Series, aligning the data on the UTC timestamps:
To remove timezone from tz-aware DatetimeIndex, use `tz_localize(None)` or `tz_convert(None)`. `tz_localize(None)` will remove timezone holding local time representations. `tz_convert(None)` will remove timezone after converting to UTC time.

```
In [405]: didx = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')

In [406]: didx
Out[406]:
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 [407]: didx.tz_localize(None)
Out[407]:
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='H')

In [408]: didx.tz_convert(None)
Out[408]:
DatetimeIndex(['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',
              '2014-08-01 19:00:00', '2014-08-01 20:00:00',
              '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
       dtype='datetime64[ns]', freq='H')
```

```
# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [409]: didx.tz_convert('UTC').tz_localize(None)
Out[409]:
DatetimeIndex(['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',
              '2014-08-01 19:00:00', '2014-08-01 20:00:00',
              '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
       dtype='datetime64[ns]', freq='H')
```

19.14.2 Ambiguous Times when Localizing

In some cases, localize cannot determine the DST and non-DST hours when there are duplicates. This often happens when reading files or database records that simply duplicate the hours. Passing `ambiguous='infer'` (infer_dst argument in prior releases) into `tz_localize` will attempt to determine the right offset. Below the top example will fail as it contains ambiguous times and the bottom will infer the right offset.

```
In [410]: rng_hourly = pd.DatetimeIndex(['11/06/2011 00:00', '11/06/2011 01:00',
               '11/06/2011 02:00', '11/06/2011 03:00',
               '11/06/2011 04:00', '11/06/2011 05:00',
               '11/06/2011 06:00', '11/06/2011 07:00',
               '11/06/2011 08:00', '11/06/2011 09:00',
               '11/06/2011 10:00', '11/06/2011 11:00',
               '11/06/2011 12:00', '11/06/2011 13:00',
               '11/06/2011 14:00', '11/06/2011 15:00',
               '11/06/2011 16:00', '11/06/2011 17:00',
               '11/06/2011 18:00', '11/06/2011 19:00',
               '11/06/2011 20:00', '11/06/2011 21:00',
               '11/06/2011 22:00', '11/06/2011 23:00'],
              dtype='datetime64[ns]', freq='H')
```

This will fail as there are ambiguous times

```
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
```

Infer the ambiguous times

```
In [411]: rng_hourly_eastern = rng_hourly.tz_localize('US/Eastern', ambiguous='infer')
```

```
In [412]: rng_hourly_eastern.tolist()
Out[412]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
  Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
  Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
  Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
  Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]
```

In addition to 'infer', there are several other arguments supported. Passing an array-like of bools or 0s/1s where True represents a DST hour and False a non-DST hour, allows for distinguishing more than one DST transition (e.g., if you have multiple records in a database each with their own DST transition). Or passing ‘NaT’ will fill in transition times with not-a-time values. These methods are available in the DatetimeIndex constructor as well as tz_localize.

```
In [413]: rng_hourly_dst = np.array([1, 1, 0, 0, 0])
```

```
In [414]: rng_hourly.tz_localize('US/Eastern', ambiguous=rng_hourly_dst).tolist()
Out[414]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
  Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
  Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
  Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
  Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]
```

```
In [415]: rng_hourly.tz_localize('US/Eastern', ambiguous='NaT').tolist()
```

```
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
  NaT,
  NaT,
  Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
  Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]
```

```
In [416]: didx = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')
```

```
In [417]: didx
Out[417]:
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 [418]: didx.tz_localize(None)
```

```
19.14. Time Zone Handling
```
19.14.3 TZ Aware Dtypes

New in version 0.17.0.

Series/DatetimeIndex with a timezone naive value are represented with a dtype of datetime64[ns].

```python
In [421]: s_naive = pd.Series(pd.date_range('20130101', periods=3))
```

```text
In [422]: s_naive
Out[422]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
dtype: datetime64[ns]
```

Series/DatetimeIndex with a timezone aware value are represented with a dtype of datetime64[ns, tz].

```python
In [423]: s_aware = pd.Series(pd.date_range('20130101', periods=3, tz='US/Eastern'))
```

```text
In [424]: s_aware
Out[424]:
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]
```
Both of these `Series` can be manipulated via the `.dt` accessor, see here.

For example, to localize and convert a naive stamp to timezone aware.

```
In [425]: s_naive.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[425]:
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]
```

Further more you can `.astype(...)` timezone aware (and naive). This operation is effectively a localize AND convert on a naive stamp, and a convert on an aware stamp.

```
# localize and convert a naive timezone
In [426]: s_naive.astype('datetime64[ns, US/Eastern]')
Out[426]:
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]

# make an aware tz naive
In [427]: s_aware.astype('datetime64[ns]')
Out[427]:
0 2013-01-01 05:00:00
1 2013-01-02 05:00:00
2 2013-01-03 05:00:00
dtype: datetime64[ns]

# convert to a new timezone
In [428]: s_aware.astype('datetime64[ns, CET]')
Out[428]:
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 the `.values` accessor on a `Series`, returns an numpy array of the data. These values are converted to UTC, as numpy does not currently support timezones (even though it is *printing* in the local timezone!).

```
In [429]: s_naive.values
Out[429]:
array(['2013-01-01T00:00:00.000000000', '2013-01-02T00:00:00.000000000',
      '2013-01-03T00:00:00.000000000'], dtype='datetime64[ns]')

In [430]: s_aware.values
Out[430]:
array(['2013-01-01T05:00:00.000000000', '2013-01-02T05:00:00.000000000',
      '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

Further note that once converted to a numpy array these would lose the tz tenor.

```
In [431]: pd.Series(s_aware.values)
Out[431]:
```

---

**19.14. Time Zone Handling**
However, these can be easily converted

```python
In [432]: pd.Series(s_aware.values).dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[432]:
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]
```
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.

**20.1 Parsing**

You can construct a Timedelta scalar through various arguments:

```python
# strings
In [1]: pd.Timedelta('1 days')
Out[1]: Timedelta('1 days 00:00:00')

In [2]: pd.Timedelta('1 days 00:00:00')
Out[2]: Timedelta('1 days 00:00:00')

In [3]: pd.Timedelta('1 days 2 hours')
Out[3]: Timedelta('1 days 02:00:00')

In [4]: pd.Timedelta('-1 days 2 min 3us')
Out[4]: Timedelta('-2 days +23:57:59.999997')

# like datetime.timedelta
# note: these MUST be specified as keyword arguments
In [5]: pd.Timedelta(days=1, seconds=1)
Out[5]: Timedelta('1 days 00:00:01')

# integers with a unit
In [6]: pd.Timedelta(1, unit='d')
Out[6]: Timedelta('1 days 00:00:00')

# from a datetime.timedelta/np.timedelta64
In [7]: pd.Timedelta(datetime.timedelta(days=1, seconds=1))
Out[7]: Timedelta('1 days 00:00:01')

In [8]: pd.Timedelta(np.timedelta64(1, 'ms'))
Out[8]: Timedelta('0 days 00:00:00.001000')
```
# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [9]: pd.Timedelta('-1us')

Timedelta('-1 days +23:59:59.999999')

# a NaT
In [10]: pd.Timedelta('nan')

NaT

In [11]: pd.Timedelta('nat')

NaT

DateOffsets (Day, Hour, Minute, Second, Milli, Micro, Nano) can also be used in construction.

In [12]: pd.Timedelta(Second(2))

Timedelta('0 days 00:00:02')

Further, operations among the scalars yield another scalar Timedelta.

In [13]: pd.Timedelta(Day(2)) + pd.Timedelta(Second(2)) + pd.Timedelta('00:00:00.000123')

Timedelta('2 days 00:00:02.000123')

20.1.1 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 [14]: pd.to_timedelta('1 days 06:05:01.00003')

Timedelta('1 days 06:05:01.000030')

In [15]: pd.to_timedelta('15.5us')

Timedelta('0 days 00:00:00.000015')

or a list/array of strings:

In [16]: pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])

TimedeltaIndex(['1 days 06:05:01.00003', '0 days 00:00:00.000015', NaT],
          dtype='timedelta64[ns]', freq=None)

The unit keyword argument specifies the unit of the Timedelta:

In [17]: pd.to_timedelta(np.arange(5), unit='s')

TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'],
               dtype='timedelta64[ns]', freq=None)

In [18]: pd.to_timedelta(np.arange(5), unit='d')

TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
20.1.2 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 [19]: pd.Timedelta.min
Out[19]: Timedelta('-106752 days +00:12:43.145224''

In [20]: pd.Timedelta.max
```

| Out[20]: Timedelta('106751 days +23:47:16.854775') |

20.2 Operations

You can operate on Series/DataFrames and construct timedelta64[ns] Series through subtraction operations on datetime64[ns] Series, or Timestamps.

```python
In [21]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [22]: td = pd.Series([pd.Timedelta(days=i) for i in range(3)])

In [23]: df = pd.DataFrame(dict(A = s, B = td))

In [24]: df
Out[24]:
   A        B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [25]: df['C'] = df['A'] + df['B']

In [26]: df
Out[26]:
   A        B        C
0 2012-01-01 0 days 2012-01-01
1 2012-01-02 1 days 2012-01-03
2 2012-01-03 2 days 2012-01-05

In [27]: df.dtypes
   A      datetime64[ns]
   B      timedelta64[ns]
   C      datetime64[ns]
dtype: object

In [28]: s - s.max()
```

| 0 -2 days |
| 1 -1 days |
| 2 0 days |
```python
dtype: timedelta64[ns]

In [29]: s - datetime.datetime(2011, 1, 1, 3, 5)
    →
0   364 days 20:55:00
1   365 days 20:55:00
2   366 days 20:55:00
dtype: timedelta64[ns]

In [30]: s + datetime.timedelta(minutes=5)
    →
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 [31]: s + Minute(5)
    →
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 [32]: s + Minute(5) + Milli(5)
    →
0   2012-01-01 00:05:00.005
1   2012-01-02 00:05:00.005
2   2012-01-03 00:05:00.005
dtype: datetime64[ns]

Operations with scalars from a timedelta64[ns] series:

In [33]: y = s - s[0]

In [34]: y
Out[34]:
0   0 days
1   1 days
2   2 days
dtype: timedelta64[ns]

Series of timedeltas with NaT values are supported:

In [35]: y = s - s.shift()

In [36]: y
Out[36]:
0   NaT
1   1 days
2   1 days
dtype: timedelta64[ns]

Elements can be set to NaT using np.nan analogously to datetimes:
```
In [37]: y[1] = np.nan

In [38]: y
Out[38]:
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):

In [39]: s.max() - s
Out[39]:
0  2 days
1   1 days
2    0 days
dtype: timedelta64[ns]

In [40]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[40]:
0  -365 days +03:05:00
1  -366 days +03:05:00
2  -367 days +03:05:00
dtype: timedelta64[ns]

In [41]: datetime.timedelta(minutes=5) + s
Out[41]:
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:

In [42]: A = s - pd.Timestamp('20120101') - pd.Timedelta('00:05:05')
In [43]: B = s - pd.Series(pd.date_range('2012-1-2', periods=3, freq='D'))
In [44]: df = pd.DataFrame(dict(A=A, B=B))
In [45]: df
Out[45]:
    A     B
0 -1 days  +23:54:55
1   0 days +23:54:55
2   1 days +23:54:55

In [46]: df.min()
Out[46]:
    A    
0 -1 days +23:54:55
1   B    
   -1 days +00:00:00

dtype: timedelta64[ns]

In [47]: df.min(axis=1)
min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

You can fillna on timedeltas. Integers will be interpreted as seconds. You can pass a timedelta to get a particular value.
You can also negate, multiply and use `abs` on Timedeltas:

```
In [57]: td1 = pd.Timedelta('-1 days 2 hours 3 seconds')

In [58]: td1
Out[58]: Timedelta('-2 days +21:59:57')

In [59]: -1 * td1
Out[59]: Timedelta('1 days 02:00:03')

In [60]: - td1
Out[60]: Timedelta('1 days 02:00:03')

In [61]: abs(td1)
Out[61]: Timedelta('1 days 02:00:03')
```

## 20.3 Reductions

Numeric reduction operation for `timedelta64[ns]` will return `Timedelta` objects. As usual NaT are skipped during evaluation.

```
In [62]: y2 = pd.Series(pd.to_timedelta(['-1 days +00:00:05', 'nat', '-1 days
˓→+00:00:05', '1 days']))

In [63]: y2
Out[63]
0   -1 days +00:00:05
1       NaT
2   -1 days +00:00:05
3       1 days
 dtype: timedelta64[ns]

In [64]: y2.mean()
Out[64]: Timedelta('-1 days +16:00:03.333333')

In [65]: y2.median()
Out[65]: Timedelta('-1 days +00:00:05')

In [66]: y2.quantile(.1)
Out[66]: Timedelta('-1 days +00:00:05')

In [67]: y2.sum()
Out[67]: Timedelta('-1 days +00:00:10')
```

## 20.4 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.

```python
In [68]: td = pd.Series(pd.date_range('20130101', periods=4)) - 
       pd.Series(pd.date_range('20121201', periods=4))

In [69]: td[2] += datetime.timedelta(minutes=5, seconds=3)

In [70]: td[3] = np.nan

In [71]: td
Out[71]:
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 [72]: td / np.timedelta64(1, 'D')

Out[72]:
 0  31.000000
 1  31.000000
 2  31.003507
 3  NaN
 dtype: float64

In [73]: td.astype('timedelta64[D]')

Out[73]:
 0  31.0
 1  31.0
 2  31.0
 3  NaN
 dtype: float64

# to seconds
In [74]: td / np.timedelta64(1, 's')

Out[74]:
 0  2678400.0
 1  2678400.0
 2  2678703.0
 3  NaN
 dtype: float64

In [75]: td.astype('timedelta64[s]')

Out[75]:
 0  2678400.0
 1  2678400.0
 2  2678703.0
 3  NaN
 dtype: float64

# to months (these are constant months)
In [76]: td / np.timedelta64(1, 'M')
```

904 Chapter 20. Time Deltas
Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series yields another timedelta64[ns] dtypes Series.

```
In [77]: td * -1
Out[77]:
0   -31 days 00:00:00
1   -31 days 00:00:00
2   -32 days 23:54:57
3      NaT
dtype: timedelta64[ns]

In [78]: td * pd.Series([1, 2, 3, 4])
˓→
0   31 days 00:00:00
1   62 days 00:00:00
2   93 days 00:15:09
3      NaT
dtype: timedelta64[ns]
```

## 20.5 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.

For a Series:

```
In [79]: td.dt.days
Out[79]:
0   31.0
1   31.0
2   31.0
3      NaN
dtype: float64

In [80]: td.dt.seconds
Out[80]:
0   0.0
1   0.0
2  303.0
```
You can access the value of the fields for a scalar `Timedelta` directly.

```python
In [81]: tds = pd.Timedelta('31 days 5 min 3 sec')
In [82]: tds.days
Out[82]: 31
In [83]: tds.seconds
Out[83]: 303
In [84]: (-tds).seconds
Out[84]: 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`.

```python
In [85]: tds.dt.components
Out[85]:
    days   hours   minutes   seconds  milliseconds  microseconds  nanoseconds
0     31.0     0.0      0.0      0.0            0            0             0
1     31.0     0.0      0.0      0.0            0            0             0
2     31.0     5.0      3.0      0.0            0            0             0
3  NaN     NaN      NaN      NaN            NaN            NaN            NaN
```

You can convert a `Timedelta` to an ISO 8601 Duration string with the `.isoformat` method.

```python
New in version 0.20.0.
```

```python
In [86]: pd.Timedelta(days=6, minutes=50, seconds=3, milliseconds=10, microseconds=10, nanoseconds=12).isoformat()
Out[86]: 'P6DT0H50M3.010010012S'
```

### 20.6 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 [87]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05', np.timedelta64(2, 'D'), datetime.timedelta(days=2, seconds=2))])
```

```python
906 Chapter 20. Time Deltas
```
Similarly to `date_range`, you can construct regular ranges of a `TimedeltaIndex`:

```python
In [89]: pd.timedelta_range(start='1 days', periods=5, freq='D')
Out[89]:
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'],
               dtype='timedelta64[ns]', freq='D')
```

```python
In [90]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')  
```

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.

```python
In [91]: s = pd.Series(np.arange(100),
                      index=pd.timedelta_range('1 days', periods=100, freq='h'))
```

```
In [92]: s
Out[92]:
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
1 days 06:00:00    6
        ....
4 days 21:00:00   93
4 days 22:00:00   94
4 days 23:00:00   95
```

---

**20.6.1 Using the TimedeltaIndex**

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.
Selections work similarly, with coercion on string-likes and slices:

```python
In [93]: s['1 day':'2 day']
Out[93]:
0 days 00:00:00 0
0 days 01:00:00 1
0 days 02:00:00 2
0 days 03:00:00 3
0 days 04:00:00 4
0 days 05:00:00 5
0 days 06:00:00 6
.. 2 days 17:00:00 41
2 days 18:00:00 42
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 [94]: s['1 day 01:00:00']
```

```python
In [95]: s[pd.Timedelta('1 day 1h')]
```

Furthermore you can use partial string selection and the range will be inferred:

```python
In [96]: s['1 day':'1 day 5 hours']
Out[96]:
0 days 00:00:00 0
0 days 01:00:00 1
0 days 02:00:00 2
0 days 03:00:00 3
0 days 04:00:00 4
0 days 05:00:00 5
Freq: H, dtype: int64
```

20.6.2 Operations

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are NaT preserving:

```python
In [97]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])
In [98]: tdi.tolist()
Out[98]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]
```
In [99]: dti = pd.date_range('20130101', periods=3)

In [100]: dti.tolist()

Out[100]:
[\text{Timestamp('2013-01-01 00:00:00', freq='D'),}
  \text{Timestamp('2013-01-02 00:00:00', freq='D'),}
  \text{Timestamp('2013-01-03 00:00:00', freq='D')}]

In [101]: (dti + tdi).tolist()

Out[101]:
[\text{Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')}]

In [102]: (dti - tdi).tolist()

Out[102]:
[\text{Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')}]

20.6.3 Conversions

Similarly to frequency conversion on a Series above, you can convert these indices to yield another Index.

In [103]: tdi / np.timedelta64(1,'s')

Out[103]:
\text{Float64Index([86400.0, nan, 172800.0], dtype='float64')}

In [104]: tdi.astype('timedelta64[s]')

Out[104]:
\text{Float64Index([86400.0, nan, 172800.0], dtype='float64')}

Scalars type ops work as well. These can potentially return a \textit{different} type of index.

# adding or timedelta and date -> datelike
In [105]: tdi + pd.Timestamp('20130101')

Out[105]:
\text{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 [106]: (pd.Timestamp('20130101') - tdi).tolist()

Out[106]:
[\text{Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')}]

# timedelta + timedelta -> timedelta
In [107]: tdi + pd.Timedelta('10 days')

Out[107]:
\text{TimedeltaIndex(['11 days', NaT, '12 days'], dtype='timedelta64[ns]', freq=None)}

# division can result in a Timedelta if the divisor is an integer
In [108]: tdi / 2

Out[108]:
\text{TimedeltaIndex(['0 days 12:00:00', NaT, '1 days 00:00:00'], dtype='timedelta64[ns]', freq=None)}

# or a Float64Index if the divisor is a Timedelta
In [109]: tdi / tdi[0]

Out[109]:
\text{Float64Index([1.0, nan, 2.0], dtype='float64')}
20.7 Resampling

Similar to timeseries resampling, we can resample with a TimedeltaIndex.

```python
In [110]: s.resample('D').mean()
Out[110]:
1 days  11.5
2 days  35.5
3 days  59.5
4 days  83.5
5 days  97.5
Freq: D, dtype: float64
```
CHAPTER TWENTYONE

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, which correspond to categorical variables in statistics: a variable, which can take on only a limited, and usually fixed, number of possible values (categories; levels in R). Examples are gender, social class, blood types, country affiliations, observation time or ratings 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.

21.1 Object Creation

Categorical Series or columns in a DataFrame can be created in several ways:

By specifying dtype="category" when constructing a Series:

```
In [1]: s = pd.Series(["a","b","c","a"], dtype="category")

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:

```
In [3]: df = pd.DataFrame({"A" : ["a", "b", "c", "a"]})

In [4]: df["B"] = df["A"].astype('category')

In [5]: df
Out[5]:
   A  B
0  a  a
1  b  b
2  c  c
3  a  a
```

By using some special functions:

```
In [6]: df = pd.DataFrame({'value': np.random.randint(0, 100, 20)})

In [7]: labels = [ "[0] - {1}" .format(i, i + 9) for i in range(0, 100, 10) ]

In [8]: df[ 'group' ] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)

In [9]: df.head(10)
Out[9]:
   value  group
0     65  60 - 69
1      49  40 - 49
2      56  50 - 59
3      43  40 - 49
4      43  40 - 49
5      91  90 - 99
6      32  30 - 39
7      87  80 - 89
8      36  30 - 39
9      81  90 - 99
```

See documentation for `cut()`.

By passing a `pandas.Categorical` object to a `Series` or assigning it to a `DataFrame`.

```
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
```
Anywhere above we passed a keyword `dtype='category'`, we used the default behavior of

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 [16]: from pandas.api.types import CategoricalDtype
In [17]: s = pd.Series(['a', 'b', 'c', 'a'])
In [18]: cat_type = CategoricalDtype(categories=['b', 'c', 'd'], ordered=True)
   ....:
   ....:
In [19]: s_cat = s.astype(cat_type)
In [20]: s_cat
Out[20]:
   0  NaN
   1   b
   2   c
   3  NaN
dtype: category
Categories (3, object): [b < c < d]
```

Categorical data has a specific category `dtype`:

```python
In [21]: df.dtypes
Out[21]:
   A  object
   B category
dtype: object
```

Note: In contrast to R’s `factor` function, categorical data is not converting input values to strings and 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.

To get back to the original Series or `numpy` array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:  

```python
In [22]: s = pd.Series(['a','b','c','a'])
In [23]: s
```
If you have already `codes` and `categories`, you can use the `from_codes()` constructor to save the factorize step during normal constructor mode:

```
In [28]: splitter = np.random.choice([0,1], 5, p=[0.5,0.5])

In [29]: s = pd.Series(pd.Categorical.from_codes(splitter, categories=["train", "test"]))
```

## 21.2 CategoricalDtype

Changed in version 0.21.0.

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.
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, `dtype='category'` is equivalent to `dtype=CategoricalDtype()`.

### 21.2.1 Equality Semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and orderedness. When comparing two unordered categoricals, the order of the categories is not considered.

```python
In [34]: c1 = CategoricalDtype(['a', 'b', 'c'], ordered=False)
# Equal, since order is not considered when ordered=False
In [35]: c1 == CategoricalDtype(['b', 'c', 'a'], ordered=False)
Out[35]: True
# Unequal, since the second CategoricalDtype is ordered
In [36]: c1 == CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[36]: False
```

All instances of `CategoricalDtype` compare equal to the string `'category'`.

```python
In [37]: c1 == 'category'
Out[37]: True
```

**Warning:** Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances `CategoricalDtype` compare equal to `'category'`, all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of categories or ordered.

### 21.3 Description

Using `.describe()` on categorical data will produce similar output to a `Series` or `DataFrame` of type string.

```python
In [38]: cat = pd.Categorical(['a', 'c', 'c', np.nan], categories=['b', 'a', 'c'])
In [39]: df = pd.DataFrame({'cat':cat, 's':['a', 'c', 'c', np.nan]})
In [40]: df.describe()
Out[40]:
```

21.3. Description 915
### Working with categories

Categorical data has a *categories* and a *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 in values.

```python
In [42]: s = pd.Series(["a","b","c","a"], dtype="category")
In [43]: s.cat.categories
Out[43]: Index(["a", "b", "c"], dtype='object')
In [44]: s.cat.ordered
Out[44]: False
```

It’s also possible to pass in the categories in a specific order:

```python
In [45]: s = pd.Series(pd.Categorical(["a","b","c","a"], categories=["c","b","a"]))
In [46]: s.cat.categories
Out[46]: Index(["c", "b", "a"], dtype='object')
In [47]: s.cat.ordered
Out[47]: False
```

**Note:** New categorical data are NOT automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered *Categorical*.

**Note:** The result of `Series.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 [48]: s = pd.Series(list('babc')).astype(CategoricalDtype(list('abcd')))  
In [49]: s
Out[49]:
0   b
1   a
```
21.4.1 Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `Categorical.rename_categories()` method:

```python
In [52]: s = pd.Series(['a', 'b', 'c', 'a'], dtype="category")
In [53]: s
Out[53]:
   0 a
   1 b
   2 c
   3 a
dtype: category
Categories (3, object): [a, b, c]
In [54]: s.cat.categories = ['Group %s' % g for g in s.cat.categories]
In [55]: s
Out[55]:
   0 Group a
   1 Group b
   2 Group c
   3 Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
In [56]: s.cat.rename_categories([1,2,3])
```

```python
Out[56]:
   0 1
   1 2
   2 3
   3 1
dtype: category
Categories (3, int64): [1, 2, 3]
In [57]: s
```
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 inplace 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:

```python
In [60]: 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:

```python
In [61]: try:
    ....:     s.cat.categories = [1,2,np.nan]
    ....:     except ValueError as e:
    ....:         print("ValueError: " + str(e))
    ....:
ValueError: Categorial categories cannot be null
```
21.4.2 Appending new categories

Appending categories can be done by using the `Categorical.add_categories()` method:

```python
In [62]: s = s.cat.add_categories([4])
In [63]: s.cat.categories
Out[63]: Index(['Group a', 'Group b', 'Group c', 4], dtype='object')
In [64]: s
Out[64]:
0 Group a
1 Group b
2 Group c
3 Group a
```

21.4.3 Removing categories

Removing categories can be done by using the `Categorical.remove_categories()` method. Values which are removed are replaced by `np.nan`:

```python
In [65]: s = s.cat.remove_categories([4])
In [66]: s
Out[66]:
0 Group a
1 Group b
2 Group c
3 Group a
```

21.4.4 Removing unused categories

Removing unused categories can also be done:

```python
In [67]: s = pd.Series(pd.Categorical(['a','b','a'], categories=['a','b','c','d']))
In [68]: s
Out[68]:
0 a
1 b
2 a
dtype: category
Categories (4, object): [a, b, c, d]
In [69]: s.cat.remove_unused_categories()
```

21.4. Working with categories
21.4.5 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 `Categorical.set_categories()`.

```python
In [70]: s = pd.Series(["one","two","four", "-"], dtype="category")

In [71]: s
Out[71]:
0 one
1 two
2 four
3 -
dtype: category
Categories (4, object): [-, four, one, two]

In [72]: s = s.cat.set_categories(["one","two","three","four"])

In [73]: s
Out[73]:
0 one
1 two
2 four
3 NaN
dtype: category
Categories (4, object): [one, two, three, 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's S1 dtype and python strings). This can result in surprising behaviour!

21.5 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 [74]: s = pd.Series(pd.Categorical(["a","b","c","a"], ordered=False))

In [75]: s.sort_values(inplace=True)

In [76]: s = pd.Series(["a","b","c","a"]).astype(
    .....:     CategoricalDtype(ordered=True)
    .....: )
    .....:

In [77]: s.sort_values(inplace=True)

In [78]: s
Out[78]:
```

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.

```
In [80]: s.cat.as_ordered()
Out[80]:
 0  a
 3  a
 1  b
 2  c
dtype: category
Categories (3, object): [a < b < c]
```

```
In [81]: s.cat.as_unordered()
Out[81]:
 0  a
 3  a
 1  b
 2  c
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 [82]: s = pd.Series([1,2,3,1], dtype="category")
```

```
In [83]: s = s.cat.set_categories([2,3,1], ordered=True)
```

```
In [84]: s
Out[84]:
 0  1
 1  2
 2  3
 3  1
dtype: category
Categories (3, int64): [2 < 3 < 1]
```

```
In [85]: s.sort_values(inplace=True)
```

```
In [86]: s
Out[86]:
 1  2
 2  3
 0  1
 3  1
dtype: category
```
21.5.1 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.

```python
In [88]: s = pd.Series([1,2,3,1], dtype="category")
In [89]: s = s.cat.reorder_categories([2,3,1], ordered=True)
In [90]: s
Out[90]:
0 1
1 2
2 3
3 1
dtype: category
Categories (3, int64): [2 < 3 < 1]
In [91]: s.sort_values(inplace=True)
In [92]: s
Out[92]:
1 2
2 3
0 1
3 1
dtype: category
Categories (3, int64): [2 < 3 < 1]
In [93]: s.min(), s.max()
```

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`. 

---

922 Chapter 21. Categorical Data
21.5.2 Multi Column Sorting

A categorical dtype 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.

In [94]: dfs = pd.DataFrame({'A' : pd.Categorical(list('bbeebbaa'), categories=['e','a','b'], ordered=True),
                      'B' : [1,2,1,2,2,1,2,1] })

In [95]: dfs.sort_values(by=['A', 'B'])

Out[95]:
        A B
0    b 1
1    e 1
2    a 1
3    a 2
4    b 1
5    b 1
6    b 2
7    e 2

Reordering the `categories` changes a future sort.

In [96]: dfs['A'] = dfs['A'].cat.reorder_categories(['a','b','e'])

In [97]: dfs.sort_values(by=['A', 'B'])

Out[97]:
        A B
1    e 1
2    e 2
3    a 1
4    a 2
5    b 1
6    b 1
7    b 2

21.6 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.

```python
In [98]: cat = pd.Series([1,2,3]).astype(CategoricalDtype([3, 2, 1], ordered=True))

In [99]: cat_base = pd.Series([2,2,2]).astype(CategoricalDtype([3, 2, 1], ordered=True))

In [100]: cat_base2 = pd.Series([2,2,2]).astype(CategoricalDtype(ordered=True))

In [101]: cat
Out[101]:
0 1
1 2
2 3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [102]: cat_base
Out[102]:
0 2
1 2
2 2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [103]: cat_base2
Out[103]:
0 2
1 2
2 2
dtype: category
Categories (1, int64): [2]
```

Comparing to a categorical with the same categories and ordering or to a scalar works:

```python
In [104]: cat > cat_base
Out[104]:
0   True
1  False
2  False
dtype: bool

In [105]: cat > 2
Out[105]:
0   True
1  False
2  False
dtype: bool
```
Equality comparisons work with any list-like object of same length and scalars:

```python
In [106]: cat == cat_base
Out[106]:
0  False
1   True
2  False
dtype: bool

In [107]: cat == np.array([1,2,3])
Out[107]:
0   True
1   True
2   True
dtype: bool

In [108]: cat == 2
→
0  False
1   True
2  False
dtype: bool
```

This doesn’t work because the categories are not the same:

```python
In [109]: try:
......:    cat > cat_base2
......: except TypeError as e:
......:    print("TypeError: " + str(e))
......:TypeError: Categoricals can only be compared if 'categories' are the same. Categories are different lengths
```

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 [110]: base = np.array([1,2,3])
In [111]: 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'.
In [112]: np.asarray(cat) > base
→array([False, False, False], dtype=bool)
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```python
In [113]: c1 = pd.Categorical(['a', 'b'], categories=['a', 'b'], ordered=False)
In [114]: c2 = pd.Categorical(['a', 'b'], categories=['b', 'a'], ordered=False)
```
21.7 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 [116]: s = pd.Series(pd.Categorical(["a","b","c","c"], categories=["c","a","b","d ->"]))

In [117]: s.value_counts()
Out[117]:
c    2
b    1
a    1
d    0
dtype: int64
```

Groupby will also show “unused” categories:

```python
In [118]: cats = pd.Categorical(["a","b","b","b","c","c","c"], categories=["a","b","c ->","d"])

In [119]: df = pd.DataFrame({"cats":cats,"values":[1,2,2,2,3,4,5]})

In [120]: df.groupby("cats").mean()
Out[120]:
   values
cats  
a  1.0
b  2.0
c  4.0
     
In [121]: cats2 = pd.Categorical(["a","a","b","b"], categories=["a","b","c"])

In [122]: df2 = pd.DataFrame({"cats":cats2,"B":["c","d","c","d"], "values":[1,2,3,4]})

In [123]: df2.groupby(["cats","B"]).mean()
Out[123]:
   values
cats B  
a  c  1.0
     d  2.0
b  c  3.0
     d  4.0
c  c  NaN
d  d  NaN
```

Pivot tables:
21.8 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.

21.8.1 Getting

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.
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 [135]: df.loc["h", :]
Out[135]:
cats  a
  values  1
Name: h, dtype: object
```

Returning a single item from categorical data will also return the value, not a categorical of length “1”:

```python
In [136]: df.iat[0,0]
Out[136]: 'a'
In [137]: df["cats"].cat.categories = ["x","y","z"]
In [138]: df.at["h","cats"] # returns a string
Out[138]: 'x'
```

**Note:** This is a difference 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` pass in a list with a single value:

```python
In [139]: df.loc["h"],"cats"
Out[139]:
h  x
Name: cats, dtype: category
Categories (3, object): [x, y, z]
```

### 21.8.2 String and datetime accessors

New in version 0.17.1.

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```python
In [140]: str_s = pd.Series(list('aabb'))
In [141]: str_cat = str_s.astype('category')
In [142]: str_cat
Out[142]:
0  a
1  a
2  b
3  b
dtype: category
Categories (2, object): [a, b]
```
In [143]: str_cat.str.contains("a")
Out[143]:
0    True
1    True
2   False
3   False
dtype: bool

In [144]: date_s = pd.Series(pd.date_range('1/1/2015', periods=5))

In [145]: date_cat = date_s.astype('category')

In [146]: date_cat
Out[146]:
    0    2015-01-01
    1    2015-01-02
    2    2015-01-03
    3    2015-01-04
    4    2015-01-05
dtype: category

In [147]: date_cat.dt.day
Out[147]:
    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.<method> 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 [148]: ret_s = str_s.str.contains("a")

In [149]: ret_cat = str_cat.str.contains("a")

In [150]: ret_s.dtype == ret_cat.dtype
Out[150]: True

In [151]: ret_s == ret_cat
Out[151]:
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.

21.8.3 Setting

Setting values in a categorical column (or Series) works as long as the value is included in the categories:

In [152]: idx = pd.Index(["h","i","j","k","l","m","n"])
In [153]: cats = pd.Categorical(["a","a","a","a","a","a","a"], categories=["a","b"])
In [154]: values = [1,1,1,1,1,1,1]
In [155]: df = pd.DataFrame({"cats":cats,"values":values}, index=idx)
In [156]: df.iloc[2:4,:] = [["b",2],["b",2]]

In [157]: df
Out[157]:
cats     values
  h      a      1
  i      a      1
  j      b      2
  k      b      2
  l      a      1
  m      a      1
  n      a      1

In [158]: try:
   ...:     df.iloc[2:4,:] = [["c",3],["c",3]]
   ....: except ValueError as e:
   ....:     print("ValueError: " + str(e))
   ....:
ValueError: 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:

In [159]: df.loc["j":"k","cats"] = pd.Categorical(["a","a"], categories=["a","b"])
In [160]: df
Out[160]:
cats     values
  h      a      1
  l      a      1
  j      a      2
  k      a      2
  l      a      1
  m      a      1
  n      a      1

In [161]: try:
   ....:     df.loc["j":"k","cats"] = pd.Categorical(["b","b"], categories=["a","b", "c"])
   ....: except ValueError as e:
   ....:     print("ValueError: " + str(e))
   ....:
ValueError: Cannot setitem on a Categorical with a new category, set the categories first
Assigning a `Categorical` to parts of a column of other types will use the values:

```python
In [162]: df = pd.DataFrame({'a':[1,1,1,1,1], 'b':['a','a','a','a','a']})
In [163]: df.loc[1:2,'a'] = pd.Categorical(['b','b'], categories=['a','b'])
In [164]: df.loc[2:3,'b'] = pd.Categorical(['b','b'], categories=['a','b'])
In [165]: df
Out[165]:
   a   b
0  1   a
1  b   a
2  b   b
3  1   b
4  1   a
```

```python
In [166]: df.dtypes
Out[166]:
a  object
b  object
dtype: object
```

### 21.8.4 Merging

You can concat two `DataFrames` containing categorical data together, but the categories of these categoricals need to be the same:

```python
In [167]: cat = pd.Series(['a','b'], dtype='category')
In [168]: vals = [1,2]
In [169]: df = pd.DataFrame({'cats':cat, 'vals':vals})
In [170]: res = pd.concat([df,df])
In [171]: res
Out[171]:
   cats  vals
0    a    1
1    b    2
2    a    1
3    b    2
```

```python
In [172]: res.dtypes
Out[172]:
cats  category
vals  int64
dtype: object
```
In this case the categories are not the same and so an error is raised:

```
In [173]: df_different = df.copy()
In [174]: df_different["cats"].cat.categories = ["c","d"]
In [175]: try:
.....:     pd.concat([df,df_different])
.....: except ValueError as e:
.....:     print("ValueError: " + str(e))
```

The same applies to `df.append(df_different)`.

See also the section on `merge dtypes` for notes about preserving merge dtypes and performance.

### 21.8.5 Unioning

New in version 0.19.0.

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.

```
In [166]: from pandas.api.types import union_categoricals
In [177]: a = pd.Categorical(["b", "c"])
In [178]: b = pd.Categorical(["a", "b"])
In [179]: union_categoricals([a, b])
Out[179]:
[ 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 lexicographically sorted, use `sort_categories=True` argument.

```
In [180]: union_categoricals([a, b], sort_categories=True)
Out[180]:
[ b, c, a, b]
Categories (3, object): [ a, b, c]
```

`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).

```
In [181]: a = pd.Categorical(["a", "b"], ordered=True)
In [182]: b = pd.Categorical(["a", "b", "a"], ordered=True)
In [183]: union_categoricals([a, b])
Out[183]:
[a, b, a, b, a]
Categories (2, object): [ a < b]
```

The below raises `TypeError` because the categories are ordered and not identical.

```
In [1]: a = pd.Categorical(["a", "b"], ordered=True)
In [2]: b = pd.Categorical(["a", "b", "c"], ordered=True)
```
New in version 0.20.0.

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```python
In [184]: a = pd.Categorical(["a", "b", "c"], ordered=True)
In [185]: b = pd.Categorical(["c", "b", "a"], ordered=True)
In [186]: union_categoricals([a, b], ignore_order=True)
Out[186]:
[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 [187]: a = pd.Series(["b", "c"], dtype='category')
In [188]: b = pd.Series(["a", "b"], dtype='category')
In [189]: union_categoricals([a, b])
Out[189]:
[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 [190]: c1 = pd.Categorical(["b", "c"])
In [191]: c2 = pd.Categorical(["a", "b"])
In [192]: c1
Out[192]:
[b, c]
Categories (2, object): [b, c]
# "b" is coded to 0
In [193]: c1.codes
Out[193]: array([0, 1], dtype=int8)
In [194]: c2
Out[194]:
[a, b]
Categories (2, object): [a, b]
# "b" is coded to 1
In [195]: c2.codes
Out[195]:
array([0, 1], dtype=int8)
```
In [196]: c = union_categoricals([c1, c2])

In [197]: c
Out[197]:
[b, c, a, b]
Categories (3, object): [b, c, a]

# "b" is coded to 0 throughout, same as c1, different from c2
In [198]: c.codes
Out[198]: array([0, 1, 2, 0], dtype=int8)

21.8.6 Concatenation

This section describes concatenations specific to category dtype. See Concatenating objects for general description.

By default, Series or DataFrame concatenation which contains the same categories results in category dtype, otherwise results in object dtype. Use .astype or union_categoricals to get category result.

# same categories
In [199]: s1 = pd.Series(['a', 'b'], dtype='category')
In [200]: s2 = pd.Series(['a', 'b', 'a'], dtype='category')
In [201]: pd.concat([s1, s2])
Out[201]:
0  a
1  b
0  a
1  b
2  a
dtype: category
Categories (2, object): [a, b]

# different categories
In [202]: s3 = pd.Series(['b', 'c'], dtype='category')
In [203]: pd.concat([s1, s3])
Out[203]:
0  a
1  b
0  b
1  c
dtype: object
In [204]: pd.concat([s1, s3]).astype('category')
Out[204]:
0  a
1  b
0  b
1  c
dtype: category
Categories (3, object): [a, b, c]
In [205]: union_categoricals([s1.values, s3.values]}
Following table summarizes the results of Categoricals related concatenations.

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>category (identical categories)</td>
<td>category</td>
</tr>
<tr>
<td>category</td>
<td>category (different categories, both not ordered)</td>
<td>object (dtype is inferred)</td>
</tr>
<tr>
<td>category</td>
<td>category (different categories, either one is ordered)</td>
<td>object (dtype is inferred)</td>
</tr>
<tr>
<td>category</td>
<td>not category</td>
<td>object (dtype is inferred)</td>
</tr>
</tbody>
</table>

### 21.9 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.

```
In [206]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))

# rename the categories
In [207]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [208]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [209]: df = pd.DataFrame({"cats":s, "vals":[1,2,3,4,5,6]})

In [210]: csv = StringIO()

In [211]: df.to_csv(csv)

In [212]: df2 = pd.read_csv(StringIO(csv.getvalue()))

In [213]: df2.dtypes

Out[213]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unnamed: 0</td>
<td>int64</td>
</tr>
<tr>
<td>cats</td>
<td>object</td>
</tr>
<tr>
<td>vals</td>
<td>int64</td>
</tr>
<tr>
<td>dtype:</td>
<td>object</td>
</tr>
</tbody>
</table>

In [214]: df2["cats"]

Out[214]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>very good</td>
</tr>
<tr>
<td>1</td>
<td>good</td>
</tr>
<tr>
<td>2</td>
<td>good</td>
</tr>
<tr>
<td>3</td>
<td>very good</td>
</tr>
<tr>
<td>4</td>
<td>very good</td>
</tr>
<tr>
<td>5</td>
<td>bad</td>
</tr>
</tbody>
</table>

Name: cats, dtype: object
```
# Redo the category

```
In [215]: df2["cats"] = df2["cats"].astype("category")
```

```
In [216]: df2["cats"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"], inplace=True)
```

```
In [217]: df2.dtypes
Out[217]:
       Unnamed: 0  cats  vals
dtype: int64  category  int64
```

```
In [218]: df2["cats"]
```

```
Out[218]:
     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`.

## 21.10 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 [219]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")
```

```
# only two categories
In [220]: s
Out[220]:
0     a
1     b
2    NaN
3     a
dtype: category
Categories (2, object): [a, b]
```

```
In [221]: s.cat.codes
```

```
Out[221]:
    0
  0  0
  1  1
```
Methods for working with missing data, e.g. `isna()`, `fillna()`, `dropna()`, all work normally:

```python
In [222]: s = pd.Series(['a', 'b', np.nan], dtype='category')

In [223]: s
Out[223]:
0  a
1  b
2  NaN
dtype: category
Categories (2, object): [a, b]

In [224]: pd.isna(s)
Out[224]:
0  False
1  False
2   True
dtype: bool

In [225]: s.fillna('a')
Out[225]:
0  a
1  b
2  a
dtype: category
Categories (2, object): [a, b]
```

## 21.11 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`. 
21.12 Gotchas

21.12.1 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.

```python
In [226]: s = pd.Series(['foo','bar']*1000)
# object dtype
In [227]: s.nbytes
Out[227]: 16000

# category dtype
In [228]: s.astype('category').nbytes
```

**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.

```python
In [229]: s = pd.Series(['foo%04d' % i for i in range(2000)])
# object dtype
In [230]: s.nbytes
Out[230]: 16000

# category dtype
In [231]: s.astype('category').nbytes
```

21.12.2 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 [232]: try:
       ....:   np.dtype("category")
       ....: except TypeError as e:
       ....:   print("TypeError: " + str(e))
       ....:
TypeError: data type "category" not understood

In [233]: dtype = pd.Categorical(['a']).dtype

In [234]: try:
       ....:   np.dtype(dtype)
       ....: except TypeError as e:
       ....:   print("TypeError: " + str(e))
       ....:
TypeError: data type not understood
```
Dtype comparisons work:

```
In [235]: dtype == np.str_
Out[235]: False
In [236]: np.str_ == dtype
Out[236]: False
```

To check if a Series contains Categorical data, use `hasattr(s, 'cat')`:

```
In [237]: hasattr(pd.Series(['a'], dtype='category'), 'cat')
Out[237]: True
In [238]: hasattr(pd.Series(['a']), 'cat')
Out[238]: False
```

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).

```
In [239]: s = pd.Series(pd.Categorical([1,2,3,4]))
In [240]: try:
.....: np.sum(s)
.....: except TypeError as e:
.....: print("TypeError: " + str(e))
.....:
TypeError: Categorical cannot perform the operation sum
```

Note: If such a function works, please file a bug at https://github.com/pandas-dev/pandas!

### 21.12.3 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.

```
In [241]: df = pd.DataFrame({'a':[1,2,3,4],
.....: 'b':['a','b','c','d'],
.....: 'cats':pd.Categorical([1,2,3,2])})
In [242]: df.apply(lambda row: type(row['cats']), axis=1)
Out[242]:
0 <class 'int'>
1 <class 'int'>
2 <class 'int'>
3 <class 'int'>
dtype: object
In [243]: df.apply(lambda col: col.dtype, axis=0)
→
a object
b object
```

---

21.12. Gotchas

939
21.12.4 Categorical Index

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 [244]: cats = pd.Categorical([1,2,3,4], categories=[4,2,3,1])
In [245]: strings = ["a","b","c","d"]
In [246]: values = [4,2,3,1]
In [247]: df = pd.DataFrame({"strings":strings, "values":values}, index=cats)
In [248]: df.index
```

```
Out[248]:
CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False, dtype='category')
```

# This now sorts by the categories order
```
In [249]: df.sort_index()
```

```
strings  values
4        d  1
2        b  2
3        c  3
1        a  4
```

21.12.5 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 [250]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])
In [251]: s = pd.Series(cat, name="cat")
In [252]: cat
Out[252]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
In [253]: s.iloc[0:2] = 10
In [254]: cat
Out[254]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
In [255]: df = pd.DataFrame(s)
```

940 Chapter 21. Categorical Data
In [256]: df["cat"].cat.categories = [1,2,3,4,5]

In [257]: cat
Out[257]:
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]

Use copy=True to prevent such a behaviour or simply don’t reuse Categoricals:

In [258]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])

In [259]: s = pd.Series(cat, name="cat", copy=True)

In [260]: cat
Out[260]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [261]: s.iloc[0:2] = 10

In [262]: cat
Out[262]:
[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 behaviour, while using a string array (e.g. np.array(["a","b","c","a"])) will not.
We use the standard convention for referencing the matplotlib API:

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

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.

### 22.1 Basic Plotting: `plot`

See the *cookbook* for some advanced strategies.

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`:

```
In [2]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [3]: ts = ts.cumsum()
In [4]: ts.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x128f4bcc0>
```
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 [5]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))
In [6]: df = df.cumsum()
In [7]: plt.figure(); df.plot();
```
You can plot one column versus another using the \texttt{x} and \texttt{y} keywords in \texttt{plot()}:  

\begin{Verbatim}
In [8]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
In [9]: df3['A'] = pd.Series(list(range(len(df))))
In [10]: df3.plot(x='A', y='B')
\end{Verbatim}
22.2 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()`. These 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:

```python
In [11]: plt.figure();
In [12]: df.iloc[5].plot(kind='bar');
```
New in version 0.17.0.

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:

```python
In [13]: df = pd.DataFrame()

In [14]: df.plot.<TAB>
```

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`. 
22.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

```
In [15]: plt.figure();

In [16]: df.iloc[5].plot.bar(); plt.axhline(0, color='k')
Out[16]: <matplotlib.lines.Line2D at 0x120654be0>
```

Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

```
In [17]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [18]: df2.plot.bar();
```
To produce a stacked bar plot, pass `stacked=True`:

```
In [19]: df2.plot.bar(stacked=True);
```
To get horizontal bar plots, use the `barh` method:

```
In [20]: df2.plot.barh(stacked=True);
```

### 22.2.2 Histograms

Histogram can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [21]: 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 [22]: plt.figure();
```

```
In [23]: df4.plot.hist(alpha=0.5)
```

```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1150fb6a0>
```
Histogram can be stacked by `stacked=True`. Bin size can be changed by `bins` keyword.

```
In [24]: plt.figure();
In [25]: df4.plot.hist(stacked=True, bins=20)
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x12d400eb8>
```
You can pass other keywords supported by matplotlib hist. For example, horizontal and cumulative histogram can be drawn by `orientation='horizontal'` and `cumulative=True`.

```
In [26]: plt.figure();

In [27]: df4['a'].plot.hist(orientation='horizontal', cumulative=True)
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x128ea3da0>
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 [28]: plt.figure();
In [29]: df['A'].diff().hist()
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x12ee45a20>
```
DataFrame.hist() plots the histograms of the columns on multiple subplots:

```python
In [30]: plt.figure()
Out[30]: <matplotlib.figure.Figure at 0x12b6df208>

In [31]: df.diff().hist(color='k', alpha=0.5, bins=50)
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1150d7748>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12266cb38>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x131130240>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x128691080>]], dtype=object)
```
The `by` keyword can be specified to plot grouped histograms:

```python
In [32]: data = pd.Series(np.random.randn(1000))
In [33]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[33]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x12a7855c0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12616ea58>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x12a173908>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12a41ef28>]], dtype=object)
```
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 [34]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])
In [35]: df.plot.box()
```

```
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x128eb0080>
```
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`.

```
In [36]: color = dict(boxes='DarkGreen', whiskers='DarkOrange',
                ....: medians='DarkBlue', caps='Gray')
     ....:

In [37]: df.plot.box(color=color, sym='r+')
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x1279f08d0>
```
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 [38]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8])
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x13134c7b8>
```
See the `boxplot` method and the `matplotlib` boxplot documentation for more.

The existing interface `DataFrame.boxplot` to plot boxplot still can be used.

```
In [39]: df = pd.DataFrame(np.random.rand(10,5))
In [40]: plt.figure();
In [41]: bp = df.boxplot()
```
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```python
In [42]: df = pd.DataFrame(np.random.rand(10,2), columns=['Col1', 'Col2'])
In [43]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [44]: plt.figure();
In [45]: bp = df.boxplot(by='X')
```
You can also pass a subset of columns to plot, as well as group by multiple columns:

```python
In [46]: df = pd.DataFrame(np.random.rand(10,3), columns=['Col1', 'Col2', 'Col3'])
In [47]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [48]: df['Y'] = pd.Series(['A','B','A','B','A','B','A','B','A','B'])
In [49]: plt.figure();
In [50]: bp = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```
Boxplot grouped by ['X', 'Y']

Warning: The default changed from 'dict' to 'axes' in version 0.19.0.

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 [51]: np.random.seed(1234)
In [52]: df_box = pd.DataFrame(np.random.randn(50, 2))
In [53]: df_box['g'] = np.random.choice(['A', 'B'], size=50)
In [54]: df_box.loc[df_box['g'] == 'B', 1] += 3
In [55]: bp = df_box.boxplot(by='g')

Boxplot grouped by g

In [56]: bp = df_box.groupby('g').boxplot()
22.2.4 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 [57]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
In [58]: df.plot.area();
```
To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

```python
In [59]: df.plot.area(stacked=False);
```
22.2.5 Scatter Plot

Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for x and y axis. These can be specified by x and y keywords each.

```python
In [60]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])
In [61]: 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 [62]: ax = df.plot.scatter(x='a', y='b', color='DarkBlue', label='Group 1');
In [63]: 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:

```
In [64]: df.plot.scatter(x='a', y='b', c='c', s=50);
```
You can pass other keywords supported by matplotlib `scatter`. Below example shows a bubble chart using a dataframe column values as bubble size.

```
In [65]: df.plot.scatter(x='a', y='b', s=df['c']*200);
```
22.2.6 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 [66]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [67]: df['b'] = df['b'] + np.arange(1000)
In [68]: 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
def df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [70]: df['b'] = df['b'] = df['b'] + np.arange(1000)
In [71]: df['z'] = np.random.uniform(0, 3, 1000)
In [72]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max,
            gridsize=25)
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x12df6f978>
```
See the `hexbin` method and the matplotlib hexbin documentation for more.

### 22.2.7 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.

```
In [73]: series = pd.Series(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], name='series')
In [74]: series.plot.pie(figsize=(6, 6))
Out[74]: <matplotlib.axes._subplots.AxesSubplot at 0x12da00438>
```
For pie plots it’s best to use square figures, one’s with an equal aspect ratio. 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 [75]: df = pd.DataFrame(3 * np.random.rand(4, 2), index=['a', 'b', 'c', 'd'],
                   columns=['x', 'y'])
In [76]: df.plot.pie(subplots=True, figsize=(8, 4))
Out[76]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x128f2bf60>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x127ebcc50>], dtype=object)
```
You can use the `labels` and `colors` keywords to specify the labels and colors of each wedge.

**Warning:** Most pandas plots use the 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.

```
In [77]: series.plot.pie(labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
                      autopct='%.2f', fontsize=20, figsize=(6, 6))
```

```
Out[77]: <matplotlib.axes._subplots.AxesSubplot at 0x12cdb8b38>
```
If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```python
In [78]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')
In [79]: series.plot.pie(figsize=(6, 6))
Out[79]: <matplotlib.axes._subplots.AxesSubplot at 0x128f2b908>
```
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.

<table>
<thead>
<tr>
<th>Plot Type</th>
<th>NaN Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>Leave gaps at NaNs</td>
</tr>
<tr>
<td>Line (stacked)</td>
<td>Fill 0’s</td>
</tr>
<tr>
<td>Bar</td>
<td>Fill 0’s</td>
</tr>
<tr>
<td>Scatter</td>
<td>Drop NaNs</td>
</tr>
<tr>
<td>Histogram</td>
<td>Drop NaNs (column-wise)</td>
</tr>
<tr>
<td>Box</td>
<td>Drop NaNs (column-wise)</td>
</tr>
<tr>
<td>Area</td>
<td>Fill 0’s</td>
</tr>
<tr>
<td>KDE</td>
<td>Drop NaNs (column-wise)</td>
</tr>
<tr>
<td>Hexbin</td>
<td>Drop NaNs</td>
</tr>
<tr>
<td>Pie</td>
<td>Fill 0’s</td>
</tr>
</tbody>
</table>

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.
22.4 Plotting Tools

These functions can be imported from pandas.plotting and take a Series or DataFrame as an argument.

22.4.1 Scatter Matrix Plot

You can create a scatter plot matrix using the scatter_matrix method in pandas.plotting:

```python
In [80]: from pandas.plotting import scatter_matrix
In [81]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])
In [82]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
```

```python
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x128dd3470>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12d75d898>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1299fe860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12d787630>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x12997c128>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12d82d470>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12997c0f0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12d6e4400>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1291a2a20>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x127a748d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1292d470>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12d82d470>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1266b2940>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x128d07470>]], dtype=object)
```
22.4.2 Density Plot

You can create density plots using the `Series.plot.kde()` and `DataFrame.plot.kde()` methods.

```
In [83]: ser = pd.Series(np.random.randn(1000))
In [84]: ser.plot.kde()
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x12a98acf8>
```
22.4.3 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. 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.

```python
In [85]: from pandas.plotting import andrews_curves
In [86]: data = pd.read_csv('data/iris.data')
In [87]: plt.figure()
Out[87]: <matplotlib.figure.Figure at 0x128db1470>
In [88]: andrews_curves(data, 'Name')
```

![Andrews Curve Plot](image-url)
22.4.4 Parallel Coordinates

Parallel coordinates is a plotting technique for plotting multivariate data. It 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 [89]: from pandas.plotting import parallel_coordinates

In [90]: data = pd.read_csv('data/iris.data')

In [91]: plt.figure()
Out[91]: <matplotlib.figure.Figure at 0x12e2ef4a8>

In [92]: parallel_coordinates(data, 'Name')
```

Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0x12aa8bf60>
22.4.5 Lag Plot

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.

```
In [93]: from pandas.plotting import lag_plot

In [94]: plt.figure()
Out[94]: <matplotlib.figure.Figure at 0x12e27e0f0>

In [95]: data = pd.Series(0.1 * np.random.rand(1000) +
                      ....: 0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
       ....:

In [96]: lag_plot(data)
Out[96]: <matplotlib.axes._subplots.AxesSubplot at 0x12a294940>
```
22.4.6 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.

```python
In [97]: from pandas.plotting import autocorrelation_plot

In [98]: plt.figure()
Out[98]: <matplotlib.figure.Figure at 0x12a218550>

In [99]: data = pd.Series(0.7 * np.random.rand(1000) +
    .....: 0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
    .....:

In [100]: autocorrelation_plot(data)
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x12e31b278>
```
22.4.7 Bootstrap Plot

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.

```
In [101]: from pandas.plotting import bootstrap_plot

In [102]: data = pd.Series(np.random.rand(1000))

In [103]: bootstrap_plot(data, size=50, samples=500, color='grey')
```

```
Out[103]: <matplotlib.figure.Figure at 0x1255c7a58>
```
22.4.8 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.

Note: The “Iris” dataset is available here.

```
In [104]: from pandas.plotting import radviz

In [105]: data = pd.read_csv('data/iris.data')

In [106]: plt.figure()
Out[106]: <matplotlib.figure.Figure at 0x127ae4898>

In [107]: radviz(data, 'Name')
```
22.5 Plot Formatting

22.5.1 Setting the plot style

From version 1.5 and up, matplotlib offers a range of preconfigured 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 do `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.

22.5.2 General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```python
In [108]: plt.figure(); 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.

### 22.5.3 Controlling the Legend

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```bash
In [109]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))

In [110]: df = df.cumsum()

In [111]: df.plot(legend=False)
Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0x128f9b7b8>
```
22.5.4 Scales

You may pass `logy` to get a log-scale Y axis.

```
In [112]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [113]: ts = np.exp(ts.cumsum())

In [114]: ts.plot(logy=True)
Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x1299f2ac8>
```
22.5.5 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```python
In [115]: df.A.plot()
Out[115]: <matplotlib.axes._subplots.AxesSubplot at 0x12f07c9b0>

In [116]: df.B.plot(secondary_y=True, style='g')
```

See also the `logx` and `loglog` keyword arguments.
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```python
In [117]: plt.figure()
Out[117]: <matplotlib.figure.Figure at 0x12de88710>

In [118]: ax = df.plot(secondary_y=['A', 'B'])

In [119]: ax.set_ylabel('CD scale')
Out[119]: Text(0,0.5,'CD scale')

In [120]: 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 [121]: plt.figure()
Out[121]: <matplotlib.figure.Figure at 0x12d261e80>

In [122]: df.plot(secondary_y=['A', 'B'], mark_right=False)
```

```python
Out[122]: <matplotlib.axes._subplots.AxesSubplot at 0x129918978>
```
22.5.6 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 labelling is performed:

```python
In [123]: plt.figure()
Out[123]: <matplotlib.figure.Figure at 0x12ed98cf8>

In [124]: df.A.plot()
```

22.5. Plot Formatting
Using the `x_compat` parameter, you can suppress this behavior:

```python
In [125]: plt.figure()
Out[125]: <matplotlib.figure.Figure at 0x12edb0ba8>

In [126]: df.A.plot(x_compat=True)
```

```python
Out[126]: <matplotlib.axes._subplots.AxesSubplot at 0x12ec991d0>
```
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 [127]: plt.figure()
Out[127]: <matplotlib.figure.Figure at 0x12ec98198>

In [128]: with pd.plotting.plot_params.use('x_compat', True):
    ......:   df.A.plot(color='r')
    ......:   df.B.plot(color='g')
    ......:   df.C.plot(color='b')
    ......:
```
22.5.7 Automatic Date Tick Adjustment

New in version 0.20.0.

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.

22.5.8 Subplots

Each Series in a DataFrame can be plotted on a different axis with the `subplots` keyword:

```
In [129]: df.plot(subplots=True, figsize=(6, 6));
```
22.5.9 Using Layout and Targeting Multiple Axes

The layout of subplots can be specified by `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If input is invalid, `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 [130]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```
The above example is identical to using

```python
In [131]: 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). Also, you can pass multiple axes created beforehand as list-like via `ax` keyword. This allows to use more complicated layout. The passed axes must be the same number as the subplots being drawn.

When multiple axes are passed via `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 [132]: fig, axes = plt.subplots(4, 4, figsize=(6, 6));
In [133]: plt.subplots_adjust(wspace=0.5, hspace=0.5);
In [134]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
In [135]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]
In [136]: df.plot(subplots=True, ax=target1, legend=False, sharex=False, sharey=False);
In [137]: (-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:

```python
In [138]: fig, axes = plt.subplots(nrows=2, ncols=2)
In [139]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A');
In [140]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B');
In [141]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C');
In [142]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D');
```
22.5.10 Plotting With Error Bars

Plotting with error bars is now supported in the `DataFrame.plot()` and `Series.plot()`
Horizontal and vertical errorbars 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`

Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a $M$ length `Series`, a $M\times2$ array should be provided indicating lower and upper (or left and right) errors. For a $M\times N$ `DataFrame`, asymmetrical errors should be in a $M\times2\times N$ array.

Here is an example of one way to easily plot group means with standard deviations from the raw data.

```python
# Generate the data
In [143]: ix3 = pd.MultiIndex.from_arrays([["a", "a", "a", "a", "b", "b", "b"], ["foo", "foo", "bar", "bar", "foo", "foo", "bar"], names=["letter", "word"])

In [144]: df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2], 'data2': [6, 5, 7, 5, 4, 5, 6, 5]}, index=ix3)
```
# Group by index labels and take the means and standard deviations for each group

```python
In [145]: gp3 = df3.groupby(level=('letter', 'word'))

In [146]: means = gp3.mean()

In [147]: errors = gp3.std()

In [148]: means
Out[148]:
   data1  data2
letter word
a   bar  3.5   6.0
    foo  2.5   5.5
b   bar  2.5   5.5
    foo  3.0   4.5
```

```python
In [149]: errors
```

```python
# Plot
In [150]: fig, ax = plt.subplots()

In [151]: means.plot.bar(yerr=errors, ax=ax)
```

```python
Out[151]: <matplotlib.axes._subplots.AxesSubplot at 0x1364c3c50>
```
22.5.11 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 [152]: fig, ax = plt.subplots(1, 1)
In [153]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])
In [154]: ax.get_xaxis().set_visible(False)  # Hide Ticks
In [155]: df.plot(table=True, ax=ax)
Out[155]: <matplotlib.axes._subplots.AxesSubplot at 0x133634a58>
```
Also, you can pass different `DataFrame` or `Series` for `table` keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as below example.

```python
In [156]: fig, ax = plt.subplots(1, 1)
In [157]: ax.get_xaxis().set_visible(False) # Hide Ticks
In [158]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0x136543cc0>
```
Finally, there is a helper function `pandas.plotting.table` to create a table from DataFrame and Series, and add it to an matplotlib.Axes. This function can accept keywords which matplotlib table has.

```
In [159]: from pandas.plotting import table

In [160]: fig, ax = plt.subplots(1, 1)

In [161]: table(ax, np.round(df.describe(), 2),
       ....:   loc='upper right', colWidths=[0.2, 0.2, 0.2])
       ....:
Out[161]: <matplotlib.table.Table at 0x1295b7a58>

In [162]: df.plot(ax=ax, ylim=(0, 2), legend=None)
```

```
\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\Out[162]: <matplotlib.axes._subplots.AxesSubplot at 0x12d52f470>
```
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 simply pass 'cubehelix' to `colormap=`

```python
In [163]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
In [164]: df = df.cumsum()
In [165]: plt.figure()
Out[165]: <matplotlib.figure.Figure at 0x1295c48d0>
In [166]: df.plot(colormap='cubehelix')
```

Note: You can get table instances on the axes using `axes.tables` property for further decorations. See the matplotlib table documentation for more.

### 22.5.12 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 simply pass 'cubehelix' to `colormap=`
or we can pass the colormap itself

```
In [167]: from matplotlib import cm

In [168]: plt.figure()
Out[168]: <matplotlib.figure.Figure at 0x115ce9cf8>

In [169]: df.plot(colormap=cm.cubehelix)
```

```
\n\n```

Out[169]: <matplotlib.axes._subplots.AxesSubplot at 0x12d1efbe0>
Colormaps can also be used other plot types, like bar charts:

```python
In [170]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
In [171]: dd = dd.cumsum()
In [172]: plt.figure()
Out[172]: <matplotlib.figure.Figure at 0x12eb384e0>
In [173]: dd.plot.bar(colormap='Greens')
```

Output image showing a bar chart with colormaps.
Parallel coordinates charts:

```python
In [174]: plt.figure()
Out[174]: <matplotlib.figure.Figure at 0x115d0f400>

In [175]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
```

```
Out[175]: <matplotlib.axes._subplots.AxesSubplot at 0x12365ac18>
```
Andrews curves charts:

```
In [176]: plt.figure()
Out[176]: <matplotlib.figure.Figure at 0x115d0f048>

In [177]: andrews_curves(data, 'Name', colormap='winter')
```

```
Out[177]: <matplotlib.axes._subplots.AxesSubplot at 0x12a4dd1d0>
```
22.6 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.

In [178]: price = pd.Series(np.random.randn(150).cumsum(),
                   index=pd.date_range('2000-1-1', periods=150, freq='B'))

In [179]: ma = price.rolling(20).mean()

In [180]: mstd = price.rolling(20).std()

In [181]: plt.figure()
Out[181]: <matplotlib.figure.Figure at 0x128ee9080>

In [182]: plt.plot(price.index, price, 'k')
Out[182]: [<matplotlib.lines.Line2D at 0x1292f6668>]

In [183]: plt.plot(ma.index, ma, 'b')
Out[183]: [<matplotlib.lines.Line2D at 0x1298ee9080>]

In [184]: plt.plot(mstd.index, mstd, 'g')
Out[184]: [<matplotlib.lines.Line2D at 0x1292f6668>]

In [185]: plt.plot(price.index, mstd, 'c')
Out[185]: [<matplotlib.lines.Line2D at 0x1298ee9080>]

Chapter 22. Visualization
22.7 Trellis plotting interface

**Warning:** The `rplot` trellis plotting interface has been removed. Please use external packages like `seaborn` for similar but more refined functionality and refer to our 0.18.1 documentation here for how to convert to using it.
New in version 0.17.1

Provisional: This is a new feature and still under development. We’ll be adding features and possibly making breaking changes in future releases. We’d love to hear your feedback.

This document is written as a Jupyter Notebook, and can be viewed or downloaded here.

You can apply conditional formatting, the visual styling of a DataFrame depending on the data within, by using the DataFrame.style property. This is a property that returns a Styler object, which has useful methods for formatting and displaying DataFrames.

The styling is accomplished using CSS. You write “style functions” that take scalars, DataFrames or Series, and return like-indexed DataFrames or Series with CSS "attribute: value" pairs for the values. These functions can be incrementally passed to the Styler which collects the styles before rendering.

23.1 Building Styles

Pass your style functions into one of the following methods:

- Styler.applymap: elementwise
- Styler.apply: column-/row-/table-wise

Both of those methods take a function (and some other keyword arguments) and applies your function to the DataFrame in a certain way. Styler.applymap works through the DataFrame elementwise. Styler.apply passes each column or row into 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.

For Styler.applymap your function should take a scalar and return a single string with the CSS attribute-value pair.

For Styler.apply your function should take a Series or DataFrame (depending on the axis parameter), and return a Series or DataFrame with an identical shape where each value is a string with a CSS attribute-value pair.

Let’s see some examples.

In [2]: import pandas as pd
   import numpy as np

   np.random.seed(24)
   df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
   df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))], axis=1)
   df.iloc[0, 2] = np.nan

Here’s a boring example of rendering a DataFrame, without any (visible) styles:
In [3]: df.style

Out[3]: <pandas.io.formats.style.Styler at 0x10654ee80>

Note: The DataFrame.style attribute is a property that returns a Styler object. Styler has a _repr_html_ method defined on it so they are rendered automatically. If you want the actual HTML back for further processing or for writing to file call the .render() method which returns a string.

The above output looks very similar to the standard DataFrame HTML representation. But we've done some work behind the scenes to attach CSS classes to each cell. We can view these by calling the .render method.

In [4]: df.style.highlight_null().render().split('n')[:10]

Out[4]: ['<style type="text/css" >', '#T_a298eba8_bb2c_11e7_bd17_186590cd1c87row0_col2 {', 'background-color: red;', '}', '<table id="T_a298eba8_bb2c_11e7_bd17_186590cd1c87" >', '<thead>', '<tr>', '<th class="blank level0" ></th>', '<th class="col_heading level0 col0" >A</th>', '<th class="col_heading level0 col1" >B</th>', '<th class="col_heading level0 col2" >C</th>']

The row0_col2 is the identifier for that particular cell. 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 set the uuid if you’d like to tie together the styling of two DataFrames).

When writing style functions, you take care of producing the CSS attribute / value pairs you want. Pandas matches those up with the CSS classes that identify each cell.

Let’s write a simple style function that will color negative numbers red and positive numbers black.

In [5]: def color_negative_red(val):
    """
    Takes a scalar and returns a string with the css property 'color: red' for negative strings, black otherwise.
    """
    color = 'red' if val < 0 else 'black'
    return 'color: %s' % color

In this case, the cell’s style depends only on it’s own value. That means we should use the Styler.applymap method which works elementwise.

In [6]: s = df.style.applymap(color_negative_red)

Out[6]: <pandas.io.formats.style.Styler at 0x103c04eb8>

Notice the similarity with the standard df.applymap, which operates on DataFrames elementwise. We want you to be able to reuse your existing knowledge of how to interact with DataFrames.

Notice also that our function returned a string containing the CSS attribute and value, separated by a colon just like in a <style> tag. This will be a common theme.

Finally, the input shapes matched. Styler.applymap calls the function on each scalar input, and the function returns a scalar output.

Now suppose you wanted to highlight the maximum value in each column. We can’t use .applymap anymore since that operated elementwise. Instead, we’ll turn to .apply which operates columnwise (or rowwise using the axis keyword). Later on we’ll see that something like highlight_max is already defined on Styler so you wouldn’t need to write this yourself.
In [7]: `def highlight_max(s):
    ...:    '''
    ...:    highlight the maximum in a Series yellow.
    ...:    '''
    ...:    is_max = s == s.max()
    ...:    return ['background-color: yellow' if v else '' for v in is_max]
In [8]: df.style.apply(highlight_max)
Out[8]: <pandas.io.formats.style.Styler at 0x103c2e048>

In this case the input is a Series, one column at a time. Notice that the output shape of `highlight_max` matches the input shape, an array with `len(s)` items.

We encourage you to use method chains to build up a style piecewise, before finally rendering at the end of the chain.

In [9]: df.style.
   ...:     .applymap(color_negative_red).
   ...:     .apply(highlight_max)
Out[9]: <pandas.io.formats.style.Styler at 0x103c28400>

Above we used `Styler.apply` to pass in each column one at a time.

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.

What if you wanted to highlight just the maximum value in the entire table? Use `.apply(function, axis=None)` to indicate that your function wants the entire table, not one column or row at a time. Let’s try that next.

We’ll rewrite our `highlight-max` to handle either Series (from `.apply(axis=0) or 1) or DataFrames (from `.apply(axis=None)`). We’ll also allow the color to be adjustable, to demonstrate that `.apply` and `.applymap` pass along keyword arguments.

In [10]: `def highlight_max(data, color='yellow'):
    ...:    '''
    ...:    highlight the maximum in a Series or DataFrame
    ...:    '''
    ...:    attr = 'background-color: {}'.format(color)
    ...:    if data.ndim == 1:  # Series from `.apply(axis=0) or axis=1
    ...:        is_max = data == data.max()
    ...:        return [attr if v else '' for v in is_max]
    ...:    else:  # from `.apply(axis=None)
    ...:        is_max = data == data.max().max()
    ...:        return pd.DataFrame(np.where(is_max, attr, ''),
    ...:                             index=data.index, columns=data.columns)

When using `Styler.apply(func, axis=None)`, the function must return a DataFrame with the same index and column labels.

In [11]: df.style.apply(highlight_max, color='darkorange', axis=None)
Out[11]: <pandas.io.formats.style.Styler at 0x103c344a8>

### 23.1.1 Building Styles Summary

Style functions should return strings with one or more CSS attribute: value delimited by semicolons. Use

- `Styler.applymap(func)` for elementwise styles
- `Styler.apply(func, axis=0)` for columnwise styles
- `Styler.apply(func, axis=1)` for rowwise styles
• Styler.apply(func, axis=None) for tablewise styles

And crucially the input and output shapes of func must match. If \( x \) is the input then \( \text{func}(x).\text{shape} == x.\text{shape} \).

23.2 Finer Control: Slicing

Both Styler.apply, and Styler.applymap accept a subset keyword. This 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)
  • A tuple is treated as \((\text{row_indexer}, \text{column_indexer})\)

Consider using \texttt{pd.IndexSlice} to construct the tuple for the last one.

In [12]: \texttt{df.style.apply(highlight_max, subset=[‘B’, ‘C’, ‘D’])}
Out[12]: <\texttt{pandas.io.formats.style.Styler} at 0x103c2e208>

For row and column slicing, any valid indexer to \texttt{.loc} will work.

In [13]: \texttt{df.style.applymap(color_negative_red, subset=pd.IndexSlice[2:5, [‘B’, ‘D’]])}
Out[13]: <\texttt{pandas.io.formats.style.Styler} at 0x103c2e6d8>

Only label-based slicing is supported right now, not positional.

If your style function uses a subset or axis keyword argument, consider wrapping your function in a \texttt{functools.partial}, partialing out that keyword.

\begin{verbatim}
my_func2 = functools.partial(my_func, subset=42)
\end{verbatim}

23.3 Finer Control: Display Values

We distinguish the display value from the actual value in Styler. To control the display value, the text is printed in each cell, use Styler.format. Cells can be formatted according to a format spec string or a callable that takes a single value and returns a string.

In [14]: \texttt{df.style.format("{:2%}")}
Out[14]: <\texttt{pandas.io.formats.style.Styler} at 0x103c2e5c0>

Use a dictionary to format specific columns.

In [15]: \texttt{df.style.format({‘B’: "(?:<4.0f)\”, ‘D’: ‘\{:+.2f\}"})}
Out[15]: <\texttt{pandas.io.formats.style.Styler} at 0x103c2e908>

Or pass in a callable (or dictionary of callables) for more flexible handling.

In [16]: \texttt{df.style.format({‘B’: \texttt{lambda} x: “\pm{:.2f}".format(abs(x))})}
Out[16]: <\texttt{pandas.io.formats.style.Styler} at 0x103c2edd8>
23.4 Builtin Styles

Finally, we expect certain styling functions to be common enough that we’ve included a few “built-in” to the Styler, so you don’t have to write them yourself.

In [17]: df.style.highlight_null(null_color='red')

Out[17]: <pandas.io.formats.style.Styler at 0x103c2e7b8>

You can create “heatmaps” with the background_gradient method. These require matplotlib, and we’ll use Seaborn to get a nice colormap.

In [18]: import seaborn as sns

   cm = sns.light_palette("green", as_cmap=True)

   s = df.style.background_gradient(cmap=cm)

   s

/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/matplotlib/colors.py:489: RuntimeWarning: invalid value encountered in less
   np.copyto(xa, -1, where=xa < 0.0)

Out[18]: <pandas.io.formats.style.Styler at 0x10531c438>

Styler.background_gradient takes the keyword arguments low and high. Roughly speaking these extend the range of your data by low and high percent so that when we convert the colors, the colormap’s entire range isn’t used. This is useful so that you can actually read the text still.

In [19]: # Uses the full color range

   df.loc[:4].style.background_gradient(cmap='viridis')

/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/matplotlib/colors.py:489: RuntimeWarning: invalid value encountered in less
   np.copyto(xa, -1, where=xa < 0.0)

Out[19]: <pandas.io.formats.style.Styler at 0x103c2e5f8>

In [20]: # Compress the color range

   (df.loc[:4]
    .style
    .background_gradient(cmap='viridis', low=.5, high=0)
    .highlight_null('red'))

/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/matplotlib/colors.py:489: RuntimeWarning: invalid value encountered in less
   np.copyto(xa, -1, where=xa < 0.0)

Out[20]: <pandas.io.formats.style.Styler at 0x10a091f28>

There’s also .highlight_min and .highlight_max.

In [21]: df.style.highlight_max(axis=0)

Out[21]: <pandas.io.formats.style.Styler at 0x10a091cc0>

Use Styler.set_properties when the style doesn’t actually depend on the values.

In [22]: df.style.set_properties(**{'background-color': 'black',
                                      'color': 'lawngreen',
                                      'border-color': 'white'})

Out[22]: <pandas.io.formats.style.Styler at 0x10a15b320>

23.4.1 Bar charts

You can include “bar charts” in your DataFrame.
New in version 0.20.0 is the ability to customize further the bar chart: You can now have the `df.style.bar` be centered on zero or midpoint value (in addition to the already existing way of having the min value at the left side of the cell), and you can pass a list of `[color_negative, color_positive]`.

Here’s how you can change the above with the new `align='mid'` option:

```python
In [24]: df.style.bar(subset=['A', 'B'], align='mid', color=['#d65f5f', '#5fba7d'])
Out[24]: <pandas.io.formats.style.Styler at 0x10a224278>
```

The following example aims to give a highlight of the behavior of the new align options:

```python
def make_html_table(aligns):
    head = "<table>
    <thead>
    <th>Align</th>
    <th>All Negative</th>
    <th>All Positive</th>
    <th>Both Neg and Pos</th>
    </thead>
    <tbody>
    
    for align in aligns:
        row = "<tr><th>{}</th>".format(align)
        for serie in [test1, test2, test3]:
            s = serie.copy()
            s.name = ''
            row += "<td>{}</td>".format(s.to_frame().style.bar(align=align,
                color=['#d65f5f', '#5fba7d'],
                width=100).render())
        row += '</tr>'
        head += row
    
    head += "</tbody>
    </table>"
    return head

head = make_html_table(['left', 'zero', 'mid'])
```

```python
Out[25]: <IPython.core.display.HTML object>
```

```python
In [25]: import pandas as pd
from IPython.display import HTML

# Test series
test1 = pd.Series([-100,-60,-30,-20], name='All Negative')
test2 = pd.Series([10,20,50,100], name='All Positive')
test3 = pd.Series([-10,-5,0,90], name='Both Pos and Neg')
```

```python
head = """
<table>
    <thead>
        <th>Align</th>
        <th>All Negative</th>
        <th>All Positive</th>
        <th>Both Neg and Pos</th>
    </thead>
    <tbody>
    
    for align in aligns:
        row = "<tr><th>{}</th>".format(align)
        for serie in [test1, test2, test3]:
            s = serie.copy()
            s.name = ''
            row += "<td>{}</td>".format(s.to_frame().style.bar(align=align,
                color=['#d65f5f', '#5fba7d'],
                width=100).render())
        row += '</tr>
        head += row
    
    head += "</tbody>
    </table>"
```

```python
head = make_html_table(['left', 'zero', 'mid'])
```
23.5 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`

```python
In [26]: df2 = -df
   style1 = df.style.applymap(color_negative_red)
   style1
Out[26]: <pandas.io.formats.style.Styler at 0x10a21ce48>
In [27]: style2 = df2.style
   style2.use(style1.export())
   style2
Out[27]: <pandas.io.formats.style.Styler at 0x10a257eb8>
```

Notice that you’re able share the styles even though they’re data aware. The styles are re-evaluated on the new DataFrame they’ve been used upon.

23.6 Other Options

You’ve seen a few methods for data-driven styling. `Styler` also provides a few other options for styles that don’t depend on the data.

- precision
- captions
- table-wide styles

Each of these can be specified in two ways:

- A keyword argument to `Styler.__init__`
- A call to one of the `.set_` methods, e.g. `.set_caption`

The best method to use depends on the context. Use the `Styler` constructor when building many styled DataFrames that should all share the same properties. For interactive use, the `.set_` methods are more convenient.

23.6.1 Precision

You can control the precision of floats using pandas’ regular `display.precision` option.

```python
In [28]: with pd.option_context('display.precision', 2):
   html = (df.style
            .applymap(color_negative_red)
            .apply(highlight_max))
   html
Out[28]: <pandas.io.formats.style.Styler at 0x10a22fda0>
```

Or through a `set_precision` method.

```python
In [29]: df.style\
   .applymap(color_negative_red)\
   .apply(highlight_max)\
   .set_precision(2)
Out[29]: <pandas.io.formats.style.Styler at 0x10a2576a0>
```
Setting the precision only affects the printed number; the full-precision values are always passed to your style functions. You can always use `df.round(2).style` if you'd prefer to round from the start.

### 23.6.2 Captions

Regular table captions can be added in a few ways.

```python
In [30]: df.style.set_caption('Colormaps, with a caption.

   .background_gradient(cmap=cm)

/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/matplotlib/colors.py:489: RuntimeWarning: invalid value encountered in less
   np.copyto(xa, -1, where=xa < 0.0)
Out[30]: <pandas.io.formats.style.Styler at 0x10a22ff28>
```

### 23.6.3 Table Styles

The next option you have are “table styles”. These are styles that apply to the table as a whole, but don’t look at the data. Certain sytlings, including pseudo-selectors like `:hover` can only be used this way.

```python
In [31]: from IPython.display import HTML
def hover(hover_color="#ffff99"):
   return dict(selector="tr:hover",
               props=[("background-color", "%s" % hover_color)])

styles = [
    hover(),
    dict(selector="th", props=[("font-size", "150%"),
                                 ("text-align", "center")]),
    dict(selector="caption", props=[("caption-side", "bottom")])
]
html = (df.style.set_table_styles(styles)
        .set_caption("Hover to highlight."))
```

```
Out[31]: <pandas.io.formats.style.Styler at 0x103c2ef98>
```

`table_styles` should be a list of dictionaries. Each dictionary should have the `selector` and `props` keys. The value for `selector` should be a valid CSS selector. Recall that all the styles are already attached to an `id`, unique to each `Styler`. This selector is in addition to that `id`. The value for `props` should be a list of tuples of (`'attribute'`, `'value'`).

`table_styles` are extremely flexible, but not as fun to type out by hand. We hope to collect some useful ones either in pandas, or preferable in a new package that builds on top the tools here.

### 23.6.4 CSS Classes

Certain CSS classes are attached to cells.

- 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

23.6.5 Limitations

• DataFrame only (use Series.to_frame().style)
  • The index and columns must be unique
  • No large repr, and performance isn’t great; this is intended for summary DataFrames
  • You can only style the values, not the index or columns
  • You can only apply styles, you can’t insert new HTML entities

Some of these will be addressed in the future.

23.6.6 Terms

• Style function: a function that’s passed into Styler.apply or Styler.applymap and returns values like 'css attribute: value'
  • Builtin style functions: style functions that are methods on Styler
  • table style: a dictionary with the two keys selector and props. selector is the CSS selector that props will apply to. props is a list of (attribute, value) tuples. A list of table styles passed into Styler.

23.7 Fun stuff

Here are a few interesting examples.

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.

In [32]: from IPython.html import widgets
   @widgets.interact
   def f(h_neg=(0, 359, 1), h_pos=(0, 359), s=(0., 99.9), l=(0., 99.9)):
       return df.style.background_gradient(
           cmap=sns.palettes.diverging_palette(h_neg=h_neg, h_pos=h_pos, s=s, l=l, as_cmap=True)
       )

/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/matplotlib/colors.py:489: RuntimeWarning: invalid value encountered in less
  np.copyto(xa, -1, where=xa < 0.0)
<pandas.io.formats.style.Styler at 0x10a15b748>
In [33]: def magnify():
    return [dict(selector="th",
       props=[("font-size", "4pt")]),
           dict(selector="td",}
In [34]: np.random.seed(25)
cmap = cmap=sns.diverging_palette(5, 250, as_cmap=True)
bigdf = pd.DataFrame(np.random.randn(20, 25)).cumsum()

bigdf.style.background_gradient(cmap, axis=1)\n .set_properties(**{'max-width': '80px', 'font-size': '1pt'})\n .set_caption("Hover to magnify")\n .set_precision(2)\n .set_table_styles(magnify())

Out[34]: <pandas.io.formats.style.Styler at 0x10a22f5c0>

### 23.8 Export to Excel

*New in version 0.20.0*

*Experimental: This is a new feature and still under development. We’ll be adding features and possibly making breaking changes in future releases. We’d love to hear your feedback.*

Some support is available for exporting styled DataFrames to Excel worksheets using the OpenPyXL engine. CSS2.2 properties handled include:

- **background-color**
- **border-style, border-width, border-color and their** {top, right, bottom, left variants}
- **color**
- **font-family**
- **font-style**
- **font-weight**
- **text-align**
- **text-decoration**
- **vertical-align**
- **white-space**: nowrap

Only CSS2 named colors and hex colors of the form #rgb or #rrggbba are currently supported.

In [35]: df.style.\
 .applymap(color_negative_red).\
 .apply(highlight_max).\
 .to_excel('styled.xlsx', engine='openpyxl')

A screenshot of the output:
23.9 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.

23.9.1 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.

In [36]: from jinja2 import Environment, ChoiceLoader, FileSystemLoader from IPython.display import HTML from pandas.io.formats.style import Styler

In [37]: mkdir templates

This next cell writes the custom template. We extend the template html.tpl, which comes with pandas.

In [38]: %file templates/myhtml.tpl
   {% extends "html.tpl" %}
   {% block table %}
      <h1>{{ table_title|default("My Table") }}</h1>
      {{ super() }}
   {% endblock table %}

Writing templates/myhtml.tpl

Now that we’ve created a template, we need to set up a subclass of Styler that knows about it.
In [39]: class MyStyler(Styler):
    env = Environment(
        loader=ChoiceLoader([
            FileSystemLoader("templates"), # contains ours
            Styler.loader, # the default
        ])
    )
    template = 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.

In [40]: MyStyler(df)
Out[40]: <__main__.MyStyler at 0x10c66fe10>

Our custom template accepts a table_title keyword. We can provide the value in the .render method.

In [41]: HTML(MyStyler(df).render(table_title="Extending Example"))
Out[41]: <IPython.core.display.HTML object>

For convenience, we provide the Styler.from_custom_template method that does the same as the custom subclass.

In [42]: EasyStyler = Styler.from_custom_template("templates", "myhtml.tpl")
    EasyStyler(df)
Out[42]: <pandas.io.formats.style.Styler.from_custom_template.<locals>.MyStyler at 0x10c66fdd8>

Here’s the template structure:

In [43]: with open("template_structure.html") as f:
    structure = f.read()

    HTML(structure)
Out[43]: <IPython.core.display.HTML object>

See the template in the GitHub repo for more details.
IO TOOLS (TEXT, CSV, HDF5, ...)

The pandas I/O API is a set of top level reader functions accessed like `pd.read_csv()` that generally return a pandas object. The corresponding writer functions are object methods that are accessed like `df.to_csv()`

<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>read_csv</td>
<td>to_csv</td>
</tr>
<tr>
<td>text</td>
<td>JSON</td>
<td>read_json</td>
<td>to_json</td>
</tr>
<tr>
<td>text</td>
<td>HTML</td>
<td>read_html</td>
<td>to_html</td>
</tr>
<tr>
<td>text</td>
<td>Local clipboard</td>
<td>read_clipboard</td>
<td>to_clipboard</td>
</tr>
<tr>
<td>binary</td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td>HDF5 Format</td>
<td>read_hdf</td>
<td>to_hdf</td>
</tr>
<tr>
<td>binary</td>
<td>Feather Format</td>
<td>read_feather</td>
<td>to_feather</td>
</tr>
<tr>
<td>binary</td>
<td>Parquet Format</td>
<td>read_parquet</td>
<td>to_parquet</td>
</tr>
<tr>
<td>binary</td>
<td>Msgpack</td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td>read_stata</td>
<td>to_stata</td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td>read_sas</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_sql</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td>Google Big Query</td>
<td>read_gbq</td>
<td>to_gbq</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 according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

24.1 CSV & Text files

The two workhorse functions for reading text files (a.k.a. flat files) are `read_csv()` and `read_table()`. They both use the same parsing code to intelligently convert tabular data into a DataFrame object. See the cookbook for some advanced strategies.

24.1.1 Parsing options

`read_csv()` and `read_table()` accept the following arguments:
24.1.1.1 Basic

- **filepath_or_buffer** [various] Either a path to a file (a `str`, `pathlib.Path`, or `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. New in version 0.18.1: support for the Python parser.

24.1.1.2 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 as if `header=0` if no names passed, otherwise as if `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 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, 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 will cause a `UserWarning` to be issued.

- **index_col** [int or sequence or False, default None] Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to not use the first column as the index (row names).

- **usecols** [array-like or callable, default None] Return a subset of the columns. If array-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). For example, a valid array-like `usecols` parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```python
In [1]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [2]: pd.read_csv(StringIO(data))
Out[2]:
   col1  col2  col3
0    a     b     1
1    a     b     2
2    c     d     3

In [3]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['COL1', 'COL3'])
```
Using this parameter results in much faster parsing time and lower memory usage.

**as_recarray** [boolean, default `False`]
Deprecated since version 0.18.2: Please call `pd.read_csv(...).to_records()` instead.

Return a NumPy recarray instead of a DataFrame after parsing the data. If set to `True`, this option takes precedence over the `squeeze` parameter. In addition, as row indices are not available in such a format, the `index_col` parameter will be ignored.

**squeeze** [boolean, default `False`]
If the parsed data only contains one column then return a Series.

**prefix** [str, default `None`]
Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, ...

**mangle_dupe_cols** [boolean, default `True`]
Duplicate columns will be specified as ‘X.0’...’X.N’, rather than ‘X’...’X’. Passing in `False` will cause data to be overwritten if there are duplicate names in the columns.

### 24.1.1.3 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` to preserve and not interpret `dtype`.

New in version 0.20.0: support for the Python parser.

**engine** [{`'c'`}, `{python}`]
Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

**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 [4]: data = 'col1,col2,col3\nna,b,1\na,b,2\nc,d,3'

In [5]: pd.read_csv(StringIO(data))
Out[5]:
   col1  col2  col3
0    a     b     1
1    a     b     2
2    c     d     3

In [6]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
```

...
skipfooter [int, default 0] Number of lines at bottom of file to skip (unsupported with engine=’c’).

skip_footer [int, default 0] Deprecated since version 0.19.0: Use the skipfooter parameter instead, as they are identical.

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)

buffer_lines [int, default None] Deprecated since version 0.19.0: Argument removed because its value is not respected by the parser

compact_ints [boolean, default False] Deprecated since version 0.19.0: Argument moved to pd.to_numeric

    If compact_ints is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the use_unsigned parameter.

use_unsigned [boolean, default False] Deprecated since version 0.18.2: Argument moved to pd.to_numeric

    If integer columns are being compacted (i.e. compact_ints=True), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

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.

24.1.1.4 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] If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they’re appended to.

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.

24.1.1.5 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.

**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.

### 24.1.1.6 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.

### 24.1.1.7 Quoting, Compression, and File Format

**compression** ['infer', 'gzip', 'bz2', 'zip', 'xz', None, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip, or xz if filepath_or_buffer is a string ending in '.gz', '.bz2', '.zip', or '.xz', 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.

New in version 0.18.1: support for ‘zip’ and ‘xz’ compression.

**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.

**quotechar** [str (length 1)] 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** [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.

tupleize_cols [boolean, default False]

Deprecated since version 0.21.0.
This argument will be removed and will always convert to MultiIndex
Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns).

24.1.1.8 Error Handling

tupleize_cols [boolean, default True] 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.

warn_bad_lines [boolean, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

24.1.2 Specifying column data types

You can indicate the data type for the whole DataFrame or individual columns:

```
In [7]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'
In [8]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9
In [9]: df = pd.read_csv(StringIO(data), dtype=object)
In [10]: df
Out[10]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
In [11]: df['a'][0]
Out[11]: '1'
In [12]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})
In [13]: df.dtypes
Out[13]:
   a  int64
   b  object
   c  float64
dtype: object
```

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()`:

```python
In [14]: data = "col_1
n1
n2
'A'
n4.22"

In [15]: df = pd.read_csv(StringIO(data), converters={'col_1':str})

In [16]: df
Out[16]:
   col_1
0     1
1     2
2    'A'
3    4.22

In [17]: df['col_1'].apply(type).value_counts()
Out[17]:
<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,

```python
In [18]: df2 = pd.read_csv(StringIO(data))

In [19]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')

In [20]: df2
Out[20]:
   col_1
0    1.00
1    2.00
2     NaN
3    4.22

In [21]: df2['col_1'].apply(type).value_counts()
Out[21]:
<class 'float'>    4
Name: col_1, dtype: int64
```

which would 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.

New in version 0.20.0: support for the Python parser.

The `dtype` option is supported by the 'python' engine.

**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,

```python
In [22]: df = pd.DataFrame({'col_1': list(range(500000)) + ['a', 'b'] +
                     list(range(500000)))

In [23]: df.to_csv('foo.csv')
```
In [24]: mixed_df = pd.read_csv('foo.csv')

In [25]: mixed_df['col_1'].apply(type).value_counts()

Out[25]:
<class 'int'> 737858
<class 'str'> 262144
Name: col_1, dtype: int64

In [26]: mixed_df['col_1'].dtype

Out[26]:
˓→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.

### 24.1.3 Specifying Categorical dtype

New in version 0.19.0.

Categorical columns can be parsed directly by specifying dtype='category' or dtype=CategoricalDtype(categories, ordered).

In [27]: data = 'col1,col2,col3
a,b,1
a,b,2
c,d,3'

In [28]: pd.read_csv(StringIO(data))

Out[28]:
  col1 col2  col3
0   a    b     1
1   a    b     2
2   c    d     3

In [29]: pd.read_csv(StringIO(data)).dtypes

Out[29]:
col1  object
col2  object
col3  int64
dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes

Out[30]:
col1  category
col2  category
col3  category
dtype: object

dtype: object

Individual columns can be parsed as a Categorical using a dict specification

In [31]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes

Out[31]:
col1  category
col2  object

New in version 0.21.0.

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`.

```python
In [32]: from pandas.api.types import CategoricalDtype

In [33]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [34]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[34]:
        col1    col2    col3  
dtype: category    object   int64

In [35]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[36]:
0    a
1    a
2   NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

When using `dtype=CategoricalDtype`, “unexpected” values outside of `dtype.categories` are treated as missing values.

```python
In [35]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[36]:
0    a
1    a
2   NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

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 homogenous categories (all numeric, all datetimes, etc.), the conversion is done automatically.

```python
In [37]: df = pd.read_csv(StringIO(data), dtype='category')

In [38]: df.dtypes
Out[38]:
        col1    col2    col3  
dtype: category    category   category

In [39]: df['col3']
Out[39]:
0    1
1    2
2    3
Name: col3, dtype: category
```
Categories (3, object): [1, 2, 3]

In [40]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [41]: df['col3']
Out[41]:
0 1
1 2
2 3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]

24.1.4 Naming and Using Columns

24.1.4.1 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 [42]: data = 'a,b,c
1,2,3
4,5,6
7,8,9'

In [43]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [44]: pd.read_csv(StringIO(data))

Out[44]:
a b c
0 1 2 3
1 4 5 6
2 7 8 9

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):

In [45]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [46]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)

Out[46]:
foo bar baz
0 1 2 3
1 4 5 6
2 7 8 9

In [47]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)

foo bar baz
0 a b c
1 1 2 3
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 [48]: data = 'skip this skip it
   ...: a,b,c
   ...: 1,2,3
   ...: 4,5,6
   ...: 7,8,9'

In [49]: pd.read_csv(StringIO(data), header=1)
Out[49]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

### 24.1.5 Duplicate names parsing

If the file or header contains duplicate names, pandas by default will deduplicate these names so as to prevent data overwrite:

```python
In [50]: data = 'a,b,a
   ...: 0,1,2
   ...: 3,4,5'

In [51]: pd.read_csv(StringIO(data))
Out[51]:
   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.0'..'X.N'. If `mangle_dupe_cols=False`, duplicate data can arise:

```python
In [2]: data = 'a,b,a
   ...: 0,1,2
   ...: 3,4,5'

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
   ...: 0,1,2
   ...: 3,4,5'

In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
   ...:
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

### 24.1.5.1 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:

New in version 0.20.0: support for callable `usecols` arguments

```python
In [52]: data = 'a,b,c,d
   ...: 1,2,3,foo
   ...: 4,5,6,bar
   ...: 7,8,9,baz'

In [53]: pd.read_csv(StringIO(data))
```
The `usecols` argument can also be used to specify which columns not to use in the final result:

```
In [57]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
```

```
Out[57]:
    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.

### 24.1.6 Comments and Empty Lines

#### 24.1.6.1 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 [58]: data = 'a,b,c
                 # commented line
1,2,3
4,5,6'
```

```
In [59]: print(data)
```

```
a,b,c

# commented line
1,2,3
```

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.
In [60]: pd.read_csv(StringIO(data), comment='#')

Out[60]:
```
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 [61]: data = 'a,b,c

1,2,3

4,5,6'

In [62]: pd.read_csv(StringIO(data), skip_blank_lines=False)

Out[62]:
```
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 [63]: data = '# comment

a,b,c

A,B,C

1,2,3

4,5,6'

In [64]: pd.read_csv(StringIO(data), comment='#', header=1)

Out[64]:
```
A B C
0 1 2 3
1 4 5 6
```

In [65]: data = 'A,B,C

# comment

a,b,c

1,2,3

4,5,6'

In [66]: pd.read_csv(StringIO(data), comment='#', skiprows=2)

Out[66]:
```
a b c
0 1 2 3
1 4 5 6
```

If both `header` and `skiprows` are specified, `header` will be relative to the end of `skiprows`. For example:

In [67]: data = '# empty

# second empty line

X,Y,Z

1,2,3

A,B,C

1,2.,4.

5.,NaN,10.0'

In [68]: print(data)
# empty
# second empty line
# third empty line
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0

In [69]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)

Out[69]:
```
A B C
0 1 2 3
1 4 5 6
```
24.1.6.2 Comments

Sometimes comments or meta data may be included in a file:

```
In [70]: 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 [71]: df = pd.read_csv('tmp.csv')
In [72]: df
Out[72]:
          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 [73]: df = pd.read_csv('tmp.csv', comment='#')
In [74]: df
Out[74]:
          ID  level category
0   Patient1   123000         x
1   Patient2   23000          y
2  Patient3   1234018         z
```

24.1.7 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 [75]: data = b'word,length\nTräumen,7\nGrüße,5'.decode('utf8').encode('latin-1')
In [76]: df = pd.read_csv(BytesIO(data), encoding='latin-1')
In [77]: df
Out[77]:
          word  length
0        Träumen          7
1          Grüße          5
```

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
24.1.8 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:

```python
In [79]: data = 'a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [80]: pd.read_csv(StringIO(data))
Out[80]:
   a    b    c
0  4  apple  bat  5.7
1  8  orange  cow 10.0
```

```python
In [81]: data = 'index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [82]: pd.read_csv(StringIO(data), index_col=0)
Out[82]:
   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`:

```python
In [83]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [84]: print(data)
    a,b,c
    4,apple,bat,
    8,orange,cow,

In [85]: pd.read_csv(StringIO(data))
Out[85]:
   a    b    c
0  4  apple  bat  NaN
1  8  orange  cow  NaN

In [86]: pd.read_csv(StringIO(data), index_col=False)
   →
   a    b    c
 0  4  apple  bat
 1  8  orange  cow
```

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:

```python
In [87]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [88]: print(data)
    a,b,c
    4,apple,bat,
    8,orange,cow,

In [89]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
```
In [90]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)

Out[90]:

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>bat</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>cow</td>
<td>NaN</td>
</tr>
</tbody>
</table>

### 24.1.9 Date Handling

#### 24.1.9.1 Specifying Date Columns

To better facilitate working with datetime data, `read_csv()` and `read_table()` use 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 `parse_dates=True`:

```python
# Use a column as an index, and parse it as dates.
In [91]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)
```

```python
In [92]: df
Out[92]:
            A  B  C
date
2009-01-01 a 1  2
2009-01-02 b 3  4
2009-01-03 c 4  5
```

```python
# These are python datetime objects
In [93]: df.index
Out[93]:
DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'],
             dtype='datetime64[ns]',
             name='date', freq=None)
```

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:

```python
In [94]: 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
```

```python
In [95]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])
```

```python
In [96]: df
```
By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [97]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
                    keep_date_col=True)
....:
In [98]: df
Out[98]:
     1_2       1_3   0     1     2
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
```

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 [99]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}
In [100]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)
In [101]: df
Out[101]:
    nominal       actual  0     1     2
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 [102]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
                 index_col=0)  # index is the nominal column

In [104]: df

Out[104]:
    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.

Note: When passing a dict as the `parse_dates` argument, the order of the columns prepended is not guaranteed, because `dict` objects do not impose an ordering on their keys. On Python 2.7+ you may use `collections.OrderedDict` instead of a regular `dict` if this matters to you. Because of this, when using a dict for ‘parse_dates’ in conjunction with the `index_col` argument, it’s best to specify `index_col` as a column label rather then as an index on the resulting frame.

### 24.1.9.2 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:

In [105]: import pandas.io.date_converters as conv

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
                 date_parser=conv.parse_date_time)

In [107]: df

Out[107]:
     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
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'])`)

3. If #2 fails, `date_parser` is called once for every row with one or more string arguments from
   the columns indicated with `parse_dates` (e.g., `date_parser('2013', '1')` for the first row,
   `date_parser('2013', '2')` for the second, etc.)

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.

You can explore the date parsing functionality in `date_converters.py` and add your own. We would love to turn
this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contain
functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second
columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single
date rather than the entire array.

### 24.1.9.3 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"

`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 [108]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
                   ..........:     infer_datetime_format=True)
               ..........:
In [109]: df
Out[109]:
```
24.1.9.4 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 [110]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c
```

```python
In [111]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[111]:
date   value  cat
0 2000-01-06   5   a
1 2000-02-06  10   b
2 2000-03-06  15   c
```

```python
In [112]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[112]:
date   value  cat
0 2000-06-01   5   a
1 2000-06-02  10   b
2 2000-06-03  15   c
```

24.1.10 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 [113]: val = '0.3066101993807095471566981359501369297504425048828125'
In [114]: data = 'a,b,c\n1,2,\n\n'.format(val)
In [115]: abs(pd.read_csv(StringIO(data), engine='c', float_precision=None)['c'][0] - float(val))
Out[115]: 1.1102230246251565e-16
In [116]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='high')['c'][0] - float(val))
Out[116]: 5.5511151231257827e-17
In [117]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='round_trip')['c'][0] - float(val))
Out[117]: 0.0
```
24.1.11 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 [118]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [119]: df = pd.read_csv('tmp.csv', sep='|')
In [120]: df
Out[120]:
       ID    level category
0  Patient1  123,000     x
1  Patient2    23,000     y
2  Patient3  1,234,018    z

In [121]: df.level.dtype
  ˓→dtype('O')

The `thousands` keyword allows integers to be parsed correctly

```python
In [122]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [123]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')
In [124]: df
Out[124]:
       ID    level category
0  Patient1   123000     x
1  Patient2     23000     y
2  Patient3  1234018     z

In [125]: df.level.dtype
   ˓→dtype('int64')
```

24.1.12 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']
read_csv(path, na_values=[5])

the default values, in addition to 5, 5.0 when interpreted as numbers are recognized as NaN

read_csv(path, keep_default_na=False, na_values=[''])

only an empty field will be NaN

read_csv(path, keep_default_na=False, na_values=["NA", "0"])

only NA and 0 as strings are NaN

read_csv(path, na_values=['Nope'])

the default values, in addition to the string "Nope" are recognized as NaN

24.1.13 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.

24.1.14 Returning Series

Using the squeeze keyword, the parser will return output with a single column as a Series:

In [126]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [127]: output = pd.read_csv('tmp.csv', squeeze=True)

In [128]: output
Out[128]:
Patient1   123000
Patient2    23000
Patient3   1234018
Name: level, dtype: int64

In [129]: type(output)

→pandas.core.series.Series

24.1.15 Boolean values

The common values True, False, TRUE, and FALSE are all recognized as boolean. Sometime you would want to recognize some other values as being boolean. To do this use the true_values and false_values options:
In [130]: data = 'a,b,c\n1,Yes,2\n3,No,4'

In [131]: print(data)

In [132]: pd.read_csv(StringIO(data))

In [133]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])

24.1.16 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 will cause an error by default:

In [27]: data = 'a,b,c\n1,2,3\n4,5,6,7\n8,9,10'

In [28]: pd.read_csv(StringIO(data))

ParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4

You can elect to skip bad lines:

In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)

Skipping line 3: expected 3 fields, saw 4

You can also use the usecols parameter to eliminate extraneous column data that appear in some lines but not others:

In [30]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

24.1.17 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 [134]: 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 [135]: dia = csv.excel()
In [136]: dia.quoting = csv.QUOTE_NONE
In [137]: pd.read_csv(StringIO(data), dialect=dia)
Out[137]:
   label1   label2   label3
index1    "a   c   e
index2     b   d   f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [138]: data = 'a,b,c\n1,2,3\n4,5,6'
In [139]: pd.read_csv(StringIO(data), lineterminator='\n')
Out[139]:
   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 [140]: data = 'a, b, c
1, 2, 3
4, 5, 6'
In [141]: print(data)
a, b, c
1, 2, 3
4, 5, 6
In [142]: pd.read_csv(StringIO(data), skipinitialspace=True)
```

The parsers make every attempt to “do the right thing” and not be very fragile. Type inference is a pretty big deal. So if a column can be coerced to integer dtype without altering the contents, it will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

### 24.1.18 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:
```python
In [143]: data = 'a,b

"hello, "Bob", nice to see you",5'  
In [144]: print(data)  
a,b
"hello, "Bob", nice to see you",5  
In [145]: pd.read_csv(StringIO(data), escapechar='\')  
```

### 24.1.19 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 behaviour, 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:

```python
In [146]: 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:

```python
#Column specifications are a list of half-intervals  
In [147]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]  
In [148]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)  
In [149]: df  
```

```
0     1         2         3  
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 [150]: widths = [6, 14, 13, 10]

In [151]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [152]: df
Out[152]:
   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 [153]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [154]: df
Out[154]:
     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

New in version 0.20.0.

read_fwf supports the dtype parameter for specifying the types of parsed columns to be different from the inferred
type.

In [155]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[155]:
    0    1    2    3
float64  float64  float64

dtype: object

In [156]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[156]:
    0    1    2    3
object  object  float64  float64

dtype: object
24.1.20 Indexes

24.1.20.1 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data columns:

```
In [157]: 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:

```
In [158]: pd.read_csv('foo.csv')
Out[158]:
A  B  C
20090101 a  1  2
20090102 b  3  4
20090103 c  4  5
```

Note that the dates weren’t automatically parsed. In that case you would need to do as before:

```
In [159]: df = pd.read_csv('foo.csv', parse_dates=True)
In [160]: df.index
Out[160]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'],
  dtype='datetime64[ns]', freq=None)
```

24.1.20.2 Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

```
In [161]: 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` and `read_table` can take a list of column numbers to turn multiple columns into a MultiIndex for the index of the returned object:

```
In [162]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])
```
24.1.20.3 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.
read_csv is also able to interpret a more common format of multi-columns indices.

```
In [170]: 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 [171]: pd.read_csv('mi2.csv',header=[0,1],index_col=0)
```

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.

### 24.1.21 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 [172]: print(open('tmp2.sv').read())
:0:1:2:3
:0:0.4691122999071863:-0.2828634432386633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.861848963477999:-2.1045692188948086:-0.4949292740687813:-0.71803807037338
3:0.7215516224436669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.715002161146375:-1.0392684835147725
7:-0.3764685823644646:-1.1578922506419993:-1.344311812731667:0.8448851414284881
8:1.075769783155533:-1.090497520822223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498
```

```
In [173]: pd.read_csv('tmp2.sv', sep=None, engine='python')
```

24.1. CSV & Text files

---

pandas: powerful Python data analysis toolkit, Release 0.21.0

---

24.1. CSV & Text files

---

1051
24.1.22 Reading multiple files to create a single DataFrame

It’s best to use `concat()` to combine multiple files. See the cookbook for an example.

24.1.23 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 [174]: print(open('tmp.sv').read())
|0|1|2|3
0|0.4691122999071863|-0.282633432866331|-1.5090585031735124|-1.1356323710171934
1|1.2112102520285061|-0.17321464905330858|0.11920871129693428|-1.0442359667799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.721551622443669|-0.7067711336300845|-1.0395749851146963|0.2718598855428299
4|-0.4249723297883753|0.5670459859204836|-1.7150020161146735|-1.0392684835147725
5|-0.6736897080883706|0.1136484096883855|-1.4784265524372235|0.5249876671147047
6|0.407052186802365|0.5770459859204836|-1.1578922506419993|0.8448851414248841
7|0.105769783715553|0.1090499752802223|1.6435630703622064|-1.4693879595399115
8|0.35702056413309086|0.6746001037299882|-1.776903716971867|-0.9689138124473498
```

```
In [175]: table = pd.read_table('tmp.sv', sep='|')

In [176]: table
Out[176]:
```
     Unnamed: 0  0  1  2  3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.211210 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569  0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673689  0.113648 -1.478427  0.524988
6  0.407052  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```

By specifying a `chunksize` to `read_csv` or `read_table`, the return value will be an iterable object of type `TextFileReader`:

```
In [177]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)

In [178]: reader
Out[178]: <pandas.io.parsers.TextFileReader at 0x120341780>
```
In [179]: 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

Specifying iterator=True will also return the TextFileReader object:

In [180]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)

In [181]: reader.get_chunk(5)

Out[181]:

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
4 4 -0.424972 0.567020 0.276232 -1.087401

24.1.24 Specifying the parser engine

Under the hood pandas uses a fast and efficient parser implemented in C as well as a python implementation which is currently more feature-complete. Where possible pandas uses the C parser (specified as engine='c'), but may fall back to python if C-unsupported options are specified. Currently, C-unsupported options 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'.

24.1.25 Reading remote files

You can pass in a URL to a CSV file:

def = pd.read_csv('https://download.bls.gov/pub/time.series/cu/cu.item',
   sep='	')

S3 URLs are handled as well:
24.1.26 Writing out Data

24.1.26.1 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 StringIO
- `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
- `cols`: 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 ‘\n’)
- `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
- `tupleize_cols`: If False (default), write as a list of tuples, otherwise write in an expanded line format suitable for read_csv
- `date_format`: Format string for datetime objects

24.1.26.2 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.

### 24.2 JSON

Read and write JSON format files and strings.

#### 24.2.1 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}`

The format of the JSON string:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
</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’.
pandas: powerful Python data analysis toolkit, Release 0.21.0

• 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 NaN‘s, NaT‘s and None will be converted to null and datetime objects will be converted based on the
date_format and date_unit parameters.
In [182]: dfj = pd.DataFrame(randn(5, 2), columns=list('AB'))
In [183]: json = dfj.to_json()
In [184]: json
Out[184]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.
˓→0061535699,"4":0.8957173022},"B":{"0":0.4137381054,"1":-0.472034511,"2":-0.
˓→3625429925,"3":-0.923060654,"4":0.8052440254}}'

24.2.1.1 Orient Options
There are a number of different options for the format of the resulting JSON file / string. Consider the following
DataFrame and Series:
In [185]: dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
.....:
columns=list('ABC'), index=list('xyz'))
.....:
In [186]: dfjo
Out[186]:
A B C
x 1 4 7
y 2 5 8
z 3 6 9
In [187]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')
In [188]: sjo
Out[188]:
x
15
y
16
z
17
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:
In [189]: dfjo.to_json(orient="columns")
Out[189]: '{"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:
In [190]: dfjo.to_json(orient="index")
In [191]: sjo.to_json(orient="index")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[191]:
˓→'{"x":15,"y":16,"z":17}'

1056

Chapter 24. IO Tools (Text, CSV, HDF5, ...)


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:

```
In [192]: dfjo.to_json(orient="records")
```

Value oriented is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [194]: dfjo.to_json(orient="values")
Out[194]: '[[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 [195]: dfjo.to_json(orient="split")
Out[195]: '{"columns": ["A", "B", "C"], "index": ["x", "y", "z"], "data": [[1, 4, 7], [2, 5, 8], [3, 6, 9]]}
```

```
In [196]: sjo.to_json(orient="split")
Out[196]: '{"name": "D", "index": ["x", "y", "z"], "data": [15, 16, 17]}'
```

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.

### 24.2.1.2 Date Handling

Writing in ISO date format

```
In [197]: dfd = pd.DataFrame(randn(5, 2), columns=list('AB'))
In [198]: dfd['date'] = pd.Timestamp('20130101')
In [199]: dfd = dfd.sort_index(1, ascending=False)
In [200]: json = dfd.to_json(date_format='iso')
```

```
In [201]: json
Out[201]: '{"date": {"0": "2013-01-01T00:00:00.000Z", "1": "2013-01-01T00:00:00.000Z", "2": "2013-01-01T00:00:00.000Z", "3": "2013-01-01T00:00:00.000Z", "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": -0.8273169356, "3": 0.4108345112, "4": 0.1320031703}}'
```

Writing in ISO date format, with microseconds
In [202]: json = dfd.to_json(date_format='iso', date_unit='us')

In [203]: json
Out[203]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-01T00:00:00.000000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.8273169356,"3":0.4108345112,"4":0.1320031703}}'

Epoch timestamps, in seconds

In [204]: json = dfd.to_json(date_format='epoch', date_unit='s')
In [205]: json
Out[205]: '{"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":-1.8273169356,"3":0.4108345112,"4":0.1320031703}}'

Writing to a file, with a date index and a date column

In [206]: dfj2 = dfj.copy()
In [207]: dfj2['date'] = pd.Timestamp('20130101')
In [208]: dfj2['ints'] = list(range(5))
In [209]: dfj2['bools'] = True
In [210]: dfj2.index = pd.date_range('20130101', periods=5)
In [211]: dfj2.to_json('test.json')
In [212]: open('test.json').read()
Out[212]: '{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"1357171200000":0.8138502857,"1357257600000":-0.8273169356,"1357344000000":0.4108345112,"1357430400000":0.1320031703},"B":{"1356998400000":2.5656459463,"1357084800000":1.3403088498,"1357171200000":-0.2261692849,"1357257600000":0.8138502857,"1357344000000":-0.8273169356},"date":{"1356998400000":1356998400000,"1357084800000":1357084800000,"1357171200000":1357171200000,"1357257600000":1357257600000,"1357344000000":1357344000000},"ints":{"1356998400000":0,"1357084800000":1,"1357171200000":2,"1357257600000":3,"1357344000000":4},"bools":{"1356998400000":true,"1357084800000":true,"1357171200000":true,"1357257600000":true,"1357344000000":true}}'

24.2.1.3 Fallback Behavior

If the JSON serializer cannot handle the container contents directly it will fallback 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:

```python
In [213]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[213]: '{"0":{"0":"(1+0j)","1":"(2+0j)"},"2":"(1+2j)"}'}
```

can be dealt with by specifying a simple default_handler:

```python
In [213]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[213]: '{"0":{"0":"(1+0j)","1":"(2+0j)"},"2":"(1+2j)"}'}
```

## 24.2.2 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}

The format of the JSON string

<table>
<thead>
<tr>
<th>split</th>
<th>dict like {index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</th>
</tr>
</thead>
<tbody>
<tr>
<td>records</td>
<td>list like [[column -&gt; value], ... , [column -&gt; value]]</td>
</tr>
<tr>
<td>index</td>
<td>dict like {index -&gt; {column -&gt; value}}</td>
</tr>
<tr>
<td>columns</td>
<td>dict like {column -&gt; {index -&gt; value}}</td>
</tr>
<tr>
<td>values</td>
<td>just the values array</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 builtin functionality
- **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.

### 24.2.2.1 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 [214]: pd.read_json(json)
Out[214]:
   A         B       date
0 -1.206412  2.565646 2013-01-01
1  1.431256  1.340309 2013-01-01
2 -1.170299 -0.226169 2013-01-01
3  0.410835  0.813850 2013-01-01
4  0.132003 -0.827317 2013-01-01
```
Reading from a file:

```python
In [215]: pd.read_json('test.json')
Out[215]:
          A         B  bools  date     ints
2013-01-01 -1.294524  0.413738 True 2013-01-01 0
2013-01-02  0.276662 -0.472035 True 2013-01-01 1
2013-01-03 -0.013960 -0.362543 True 2013-01-01 2
2013-01-04 -0.006154 -0.923061 True 2013-01-01 3
2013-01-05  0.895717  0.805244 True 2013-01-01 4
```

Don’t convert any data (but still convert axes and dates):

```python
In [216]: pd.read_json('test.json', dtype=object).dtypes
Out[216]:
A     object
B     object
bools object
date   object
ints   object
dtype: object
```

Specify dtypes for conversion:

```python
In [217]: pd.read_json('test.json', dtype={'A' : 'float32', 'bools' : 'int8'}).dtypes
Out[217]:
A     float32
B     float64
bools  int8
date   datetime64[ns]
ints   int64
dtype: object
```

Preserve string indices:

```python
In [218]: si = pd.DataFrame(np.zeros((4, 4)),
                          columns=list(range(4)),
                          index=[str(i) for i in range(4)])
```

```python
In [219]: si
Out[219]:
       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
```

```python
In [220]: si.index
Index(['0', '1', '2', '3'], dtype='object')
```

```python
In [221]: si.columns
Int64Index([0, 1, 2, 3], dtype='int64')
```

```python
In [222]: json = si.to_json()
```

```python
In [223]: sij = pd.read_json(json, convert_axes=False)
```
In [224]: sij
Out[224]:
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 [225]: sij.index
Out[225]: Index(['0', '1', '2', '3'], dtype='object')

In [226]: sij.columns
Out[226]: Index(['0', '1', '2', '3'], dtype='object')

Dates written in nanoseconds need to be read back in nanoseconds:

In [227]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as milliseconds -> Won't Work
In [228]: dfju = pd.read_json(json, date_unit='ms')

In [229]: dfju
Out[229]:
A   B  bools  date   ints
0 1356998400000000000  -1.294524  0.413738  True 1356998400000000000  0
1 1357084800000000000   0.276662  -0.472035  True 1356998400000000000  1
2 1357171200000000000  -0.013960  -0.362543  True 1356998400000000000  2
3 1357257600000000000  -0.006154  -0.923061  True 1356998400000000000  3
4 1357344000000000000   0.895717   0.805244  True 1356998400000000000  4

# Let pandas detect the correct precision
In [230]: dfju = pd.read_json(json)

In [231]: dfju
Out[231]:
A   B  bools  date   ints
0 2013-01-01   -1.294524  0.413738  True 2013-01-01  0
1 2013-01-02   0.276662  -0.472035  True 2013-01-01  1
2 2013-01-03  -0.013960  -0.362543  True 2013-01-01  2
3 2013-01-04  -0.006154  -0.923061  True 2013-01-01  3
4 2013-01-05   0.895717   0.805244  True 2013-01-01  4

# Or specify that all timestamps are in nanoseconds
In [232]: dfju = pd.read_json(json, date_unit='ns')

In [233]: dfju
Out[233]:
A   B  bools  date   ints
0 2013-01-01   -1.294524  0.413738  True 2013-01-01  0
1 2013-01-02   0.276662  -0.472035  True 2013-01-01  1
2 2013-01-03  -0.013960  -0.362543  True 2013-01-01  2
3 2013-01-04  -0.006154  -0.923061  True 2013-01-01  3
4 2013-01-05   0.895717   0.805244  True 2013-01-01  4
24.2.2.2 The Numpy Parameter

Note: 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:

```
In [234]: randfloats = np.random.uniform(-100, 1000, 10000)
In [235]: randfloats.shape = (1000, 10)
In [236]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))
In [237]: jsonfloats = dffloats.to_json()

In [238]: timeit pd.read_json(jsonfloats)
7.68 ms +- 85.3 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
In [239]: timeit pd.read_json(jsonfloats, numpy=True)
5.19 ms +- 329 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [240]: jsonfloats = dffloats.head(100).to_json()
In [241]: timeit pd.read_json(jsonfloats)
4.15 ms +- 24.9 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
In [242]: timeit pd.read_json(jsonfloats, numpy=True)
3.37 ms +- 45.5 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.

24.2.3 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 [243]: from pandas.io.json import json_normalize
In [244]: data = [{"id": 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
```
.....:  {'name': {'given': 'Mose', 'family': 'Regner'}},
.....:  {'id': 2, 'name': 'Faye Raker'}
.....:

In [245]: json_normalize(data)
Out[245]:
    id   name name.family name.first name.given name.last
0  1.0  NaN     NaN      NaN     Coleen     NaN     Volk
1  NaN  NaN     Regner   NaN     Mose       NaN     NaN
2  2.0  Faye Raker  NaN     NaN     NaN       NaN     NaN

In [246]: data = [{'state': 'Florida',
.....:    'shortname': 'FL',
.....:    'info': {
.....:        'governor': 'Rick Scott',
.....:    },
.....:    'counties': [{'name': 'Dade', 'population': 12345},
.....:                   {'name': 'Broward', 'population': 40000},
.....:                   {'name': 'Palm Beach', 'population': 60000}]],

In [247]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor']])
Out[247]:
     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
3  Summit        1234 Ohio        OH       John Kasich
4 Cuyahoga       1337 Ohio        OH       John Kasich

24.2.4 Line delimited json

New in version 0.19.0.

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

New in version 0.21.0.

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 [248]: json1 = ''
.....:    "a":1,"b":2
.....:    "a":3,"b":4
.....:    ...
.....:    

In [249]: df = pd.read_json(json1, lines=True)
In [250]: df
Out[250]:
a  b
0 1 2
1 3 4

In [251]: df.to_json(orient='records', lines=True)

Out[251]: {'a':1,'b':2}
{'a':3,'b':4}'

# reader is an iterator that returns 'chunksize' lines each iteration
In [252]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [253]: reader
Out[253]: <pandas.io.json.json.JsonReader at 0x122daa9b0>

In [254]: for chunk in reader:
.....: print(chunk)
.....: Empty DataFrame
Columns: []
Index: []
a  b
0 1 2
a  b
1 3 4

24.2.5 Table Schema

New in version 0.20.0.

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 [255]: 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'))

In [256]: df
Out[256]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx</td>
<td>0</td>
<td>a</td>
<td>2016-01-01</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>b</td>
<td>2016-01-02</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>c</td>
<td>2016-01-03</td>
</tr>
</tbody>
</table>

In [257]: df.to_json(orient='table', date_format="iso")

"{schema": {"fields": [{"name": "idx", "type": "integer"}, {"name": "A", "type": "integer"},
.....:  {"name": "B", "type": "string"}, {"name": "C", "type": "datetime"}],"primaryKey": ["idx"],
.....:  "pandas_version": "0.20.0"}, "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"}]}

24.2. JSON
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:

<table>
<thead>
<tr>
<th>Pandas type</th>
<th>Table Schema type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64</td>
<td>integer</td>
</tr>
<tr>
<td>float64</td>
<td>number</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>datetime64[ns]</td>
<td>datetime</td>
</tr>
<tr>
<td>timedelta64[ns]</td>
<td>duration</td>
</tr>
<tr>
<td>categorical</td>
<td>any</td>
</tr>
<tr>
<td>object</td>
<td>str</td>
</tr>
</tbody>
</table>

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 naïve values, which are treated as UTC with an offset of 0.

```
In [258]: from pandas.io.json import build_table_schema
In [259]: s = pd.Series(pd.date_range('2016', periods=4))
In [260]: build_table_schema(s)
```

```
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values', 'type': 'datetime'}],
    'pandas_version': '0.20.0',
    'primaryKey': ['index']}
```

- Datetimes with a timezone (before serializing), include an additional field `tz` with the time zone name (e.g. 'US/Central').

```
In [261]: s_tz = pd.Series(pd.date_range('2016', periods=12, tz='US/Central'))
```

```
In [262]: build_table_schema(s_tz)
```

```
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
    'pandas_version': '0.20.0',
    'primaryKey': ['index']}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain and additional field `freq` with the period’s frequency, e.g. 'A-DEC'

```
In [263]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC', periods=4))
```
In [264]: build_table_schema(s_per)
Out[264]:
{'fields': [{'freq': 'A-DEC', 'name': 'index', 'type': 'datetime'},
  {'name': 'values', 'type': 'integer'}],
'pandas_version': '0.20.0',
'primaryKey': ['index']}

• Categoricals use the any type and an enum constraint listing the set of possible values. Additionally, an ordered field is included

In [265]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))
In [266]: build_table_schema(s_cat)
Out[266]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'constraints': {'enum': ['a', 'b']},
   'name': 'values',
   'ordered': False,
   'type': 'any'}],
'pandas_version': '0.20.0',
'primaryKey': ['index']}

• A primaryKey field, containing an array of labels, is included if the index is unique:

In [267]: s_dupe = pd.Series([1, 2], index=[1, 1])
In [268]: build_table_schema(s_dupe)
Out[268]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
'pandas_version': '0.20.0'}

• The primaryKey behavior is the same with MultiIndexes, but in this case the primaryKey is an array:

In [269]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([['a', 'b'],
                                              (0, 1)]))
In [270]: build_table_schema(s_multi)
Out[270]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
'pandas_version': '0.20.0',
'primaryKey': FrozenList(['level_0', 'level_1'])}

• 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.

_Table Schema: http://specs.frictionlessdata.io/json-table-schema/
24.3 HTML

24.3.1 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 DataFrames. 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

```
In [271]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'
In [272]: dfs = pd.read_html(url)
In [273]: dfs
Out[273]:
<table>
<thead>
<tr>
<th>Bank Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 The Farmers and Merchants State Bank of Argonia</td>
<td>Argonia</td>
</tr>
<tr>
<td>1 Fayette County Bank</td>
<td>Saint Elmo</td>
</tr>
<tr>
<td>2 Guaranty Bank, (d/b/a BestBank in Georgia &amp; M...</td>
<td>Milwaukee</td>
</tr>
<tr>
<td>3 First NBC Bank</td>
<td>New Orleans</td>
</tr>
<tr>
<td>4 Proficio Bank</td>
<td>Cottonwood Heights</td>
</tr>
<tr>
<td>5 Seaway Bank and Trust Company</td>
<td>Chicago</td>
</tr>
<tr>
<td>6 Harvest Community Bank</td>
<td>Pennsville</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>547 Hamilton Bank, NA</td>
<td>Miami</td>
</tr>
<tr>
<td>548 Sinclair National Bank</td>
<td>Gravette</td>
</tr>
<tr>
<td>549 Superior Bank, FSB</td>
<td>Hillsdale</td>
</tr>
<tr>
<td>550 Malta National Bank</td>
<td>Malta</td>
</tr>
<tr>
<td>551 First Alliance Bank &amp; Trust Co.</td>
<td>Manchester</td>
</tr>
<tr>
<td>552 National State Bank of Metropolis</td>
<td>Metropolis</td>
</tr>
<tr>
<td>553 Bank of Honolulu</td>
<td>Honolulu</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>ST</th>
<th>CERT</th>
<th>Acquiring Institution</th>
<th>Closing Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KS</td>
<td>Conway Bank</td>
<td>October 13, 2017</td>
</tr>
<tr>
<td>1</td>
<td>IL</td>
<td>United Fidelity Bank, fsb</td>
<td>May 26, 2017</td>
</tr>
<tr>
<td>2</td>
<td>WI</td>
<td>First-Citizens Bank &amp; Trust Company</td>
<td>May 5, 2017</td>
</tr>
<tr>
<td>3</td>
<td>LA</td>
<td>Whitney Bank</td>
<td>April 28, 2017</td>
</tr>
<tr>
<td>4</td>
<td>UT</td>
<td>Cache Valley Bank</td>
<td>March 3, 2017</td>
</tr>
<tr>
<td>5</td>
<td>IL</td>
<td>State Bank of Texas</td>
<td>January 27, 2017</td>
</tr>
<tr>
<td>6</td>
<td>NJ</td>
<td>First-Citizens Bank &amp; Trust Company</td>
<td>January 13, 2017</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>547</td>
<td>FL</td>
<td>Israel Discount Bank of New York</td>
<td>January 11, 2002</td>
</tr>
<tr>
<td>548</td>
<td>AR</td>
<td>Delta Trust &amp; Bank</td>
<td>September 7, 2001</td>
</tr>
<tr>
<td>549</td>
<td>IL</td>
<td>Superior Federal, FSB</td>
<td>July 27, 2001</td>
</tr>
<tr>
<td>550</td>
<td>OH</td>
<td>North Valley Bank</td>
<td>May 3, 2001</td>
</tr>
<tr>
<td>551</td>
<td>NH</td>
<td>Southern New Hampshire Bank &amp; Trust</td>
<td>February 2, 2001</td>
</tr>
<tr>
<td>552</td>
<td>IL</td>
<td>Banterra Bank of Marion</td>
<td>December 14, 2000</td>
</tr>
</tbody>
</table>
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 [274]: with open(file_path, 'r') as f:
    ...:     dfs = pd.read_html(f.read())
    ...:

In [275]: dfs
```

### Example Data

<table>
<thead>
<tr>
<th>Bank Name</th>
<th>City</th>
<th>ST</th>
<th>CERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks of Wisconsin d/b/a Bank of Kenosha</td>
<td>Kenosha</td>
<td>WI</td>
<td>35386</td>
</tr>
<tr>
<td>Central Arizona Bank</td>
<td>Scottsdale</td>
<td>AZ</td>
<td>34527</td>
</tr>
<tr>
<td>Sunrise Bank</td>
<td>Valdosta</td>
<td>GA</td>
<td>58185</td>
</tr>
<tr>
<td>Pisgah Community Bank</td>
<td>Asheville</td>
<td>NC</td>
<td>58701</td>
</tr>
<tr>
<td>Douglas County Bank</td>
<td>Douglasville</td>
<td>GA</td>
<td>21649</td>
</tr>
<tr>
<td>Parkway Bank</td>
<td>Lenoir</td>
<td>NC</td>
<td>57158</td>
</tr>
<tr>
<td>Chipola Community Bank</td>
<td>Marianna</td>
<td>FL</td>
<td>58034</td>
</tr>
<tr>
<td>Hamilton Bank, NAEn Espanol</td>
<td>Miami</td>
<td>FL</td>
<td>24382</td>
</tr>
<tr>
<td>Sinclair National Bank</td>
<td>Gravette</td>
<td>AR</td>
<td>34248</td>
</tr>
<tr>
<td>Superior Bank, FSB</td>
<td>Hinsdale</td>
<td>IL</td>
<td>32646</td>
</tr>
<tr>
<td>Malta National Bank</td>
<td>Malta</td>
<td>OH</td>
<td>6629</td>
</tr>
<tr>
<td>First Alliance Bank &amp; Trust Co.</td>
<td>Manchester</td>
<td>NH</td>
<td>34264</td>
</tr>
<tr>
<td>National State Bank of Metropolis</td>
<td>Metropolis</td>
<td>IL</td>
<td>3815</td>
</tr>
<tr>
<td>Bank of Honolulu</td>
<td>Honolulu</td>
<td>HI</td>
<td>21029</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acquiring Institution</th>
<th>Closing Date</th>
<th>Updated Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Western State Bank</td>
<td>May 14, 2013</td>
<td>May 20, 2013</td>
</tr>
<tr>
<td>Synovus Bank</td>
<td>May 10, 2013</td>
<td>May 21, 2013</td>
</tr>
<tr>
<td>Capital Bank, N.A.</td>
<td>May 10, 2013</td>
<td>May 14, 2013</td>
</tr>
<tr>
<td>Hamilton State Bank</td>
<td>April 26, 2013</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>CertusBank, National Association</td>
<td>April 26, 2013</td>
<td>May 17, 2013</td>
</tr>
</tbody>
</table>
You can even pass in an instance of `StringIO` if you so desire

```
In [276]: with open(file_path, 'r') as f:
    ....:     sio = StringIO(f.read())
    ....:

In [277]: dfs = pd.read_html(sio)
```

```
In [278]: dfs
Out[278]:
      Acquiring Institution       Closing Date     Updated Date
0                 North Shore Bank, FSB  May 31, 2013     May 31, 2013
1             Western State Bank  May 14, 2013     May 20, 2013
2                Synovus Bank  May 10, 2013     May 21, 2013
3              Capital Bank, N.A.  May 10, 2013     May 14, 2013
4           Hamilton State Bank  April 26, 2013     May 16, 2013
5     CertusBank, National Association  April 26, 2013     May 17, 2013
6   First Federal Bank of Florida  April 18, 2013     May 16, 2013
```

[506 rows x 7 columns]
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 (``xrange`` (Python 2 only) 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'])
```

New in version 0.19.

Specify whether to keep the default set of NaN values

```python
dfs = pd.read_html(url, keep_default_na=False)
```

New in version 0.19.

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})
```

New in version 0.19.

Use some combination of the above
dfs = pd.read_html(url, match='Metcalf Bank', index_col=0)

Read in pandas to_html output (with some loss of floating point precision)

df = pd.DataFrame(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)

dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])

or

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.

dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])

### 24.3.2 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.

In [279]: df = pd.DataFrame(randn(2, 2))

In [280]: df
Out[280]:
   0    1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [281]: print(df.to_html()) # raw html

In [281]: print(df.to_html()) # raw html

```html
<body>
<table border="1" class="dataframe">
  
  <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 `columns` argument will limit the columns shown:

```python
In [282]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<th>0</th>
<td>-0.184744</td>
</tr>
<tr>
<th>1</th>
<td>-0.856240</td>
</tr>
</tbody>
</table>
```

`float_format` takes a Python callable to control the precision of floating point values:

```python
In [283]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">
<th></th>
<th>0</th>
<th>1</th>
</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>
```
HTML:

bold_rows will make the row labels bold by default, but you can turn that off

```python
In [284]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">  
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.184744</td>
<td>0.496971</td>
</tr>
<tr>
<td>1</td>
<td>-0.856240</td>
<td>1.857977</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 [285]: 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>
```

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
Escaped:

```python
In [287]: print(df.to_html())
```

```
<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>&amp;</td>  
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>  
      <td>&lt;</td>  
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>  
      <td>&gt;</td>  
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```python
In [288]: 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>
```
24.3.3 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.

24.4 Excel files

The read_excel() method can read Excel 2003 (.xls) and Excel 2007+ (.xlsx) files using the xlrd Python module. 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 for some advanced strategies.
24.4.1 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
read_excel('path_to_file.xls', sheet_name='Sheet1')
```

24.4.1.1 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 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'] = read_excel(xls, 'Sheet1', index_col=None, na_values=['NA'])
    data['Sheet2'] = read_excel(xls, 'Sheet2', index_col=None, na_values=['NA'])

# equivalent using the read_excel function
data = read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'], index_col=None, na_values=['NA'])
```

New in version 0.17.

`read_excel` can take an `ExcelFile` object as input.

24.4.1.2 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
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```python
# Returns a DataFrame
read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

Using all default values:

```python
# Returns a DataFrame
read_excel('path_to_file.xls')
```

Using None to get all sheets:

```python
# Returns a dictionary of DataFrames
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.
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.

### 24.4.1.3 Reading a MultiIndex

New in version 0.17.

`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 [289]: df = pd.DataFrame({'a':[1,2,3,4], 'b':[5,6,7,8]},
    index=pd.MultiIndex.from_product([['a','b'],['c','d']]))

In [290]: df.to_excel('path_to_file.xlsx')

In [291]: df = pd.read_excel('path_to_file.xlsx', index_col=[0,1])
```
If the index has level names, they will parsed as well, using the same parameters.

```python
In [293]: df.index = df.index.set_names(['lvl1', 'lvl2'])
In [294]: df.to_excel('path_to_file.xlsx')
In [295]: df = pd.read_excel('path_to_file.xlsx', index_col=[0,1])
In [296]: df
Out[296]:
   a  b
lvl1 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 [297]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']], names=['c1', 'c2'])
In [298]: df.to_excel('path_to_file.xlsx')
In [299]: df = pd.read_excel('path_to_file.xlsx', index_col=[0,1], header=[0,1])
In [300]: df
Out[300]:
   c1  a  c2  b  d
lvl1 lvl2
  a  c  1  d  5
  b  c  3  d  7
d  4  8
```

### 24.4.1.4 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.

If `usecols` is an integer, then it is assumed to indicate the last column to be parsed.

```python
read_excel('path_to_file.xls', 'Sheet1', usecols=2)
```
If `usecols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```python
read_excel('path_to_file.xls', 'Sheet1', usecols=[0, 2, 3])
```

### 24.4.1.5 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:

```python
read_excel('path_to_file.xls', 'Sheet1', parse_dates=['date_strings'])
```

### 24.4.1.6 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
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
cfun = lambda x: int(x) if x else -1
read_excel('path_to_file.xls', 'Sheet1', converters={'MyInts': cfun})
```

### 24.4.1.7 dtype Specifications

New in version 0.20.

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
read_excel('path_to_file.xls', dtype={'MyInts': 'int64', 'MyText': str})
```

### 24.4.2 Writing Excel Files

#### 24.4.2.1 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`.
df.to_excel('path_to_file.xlsx', index_label='label', merge_cells=False)

The Panel class also has a `to_excel` instance method, which writes each DataFrame in the Panel to a separate sheet. In order to write separate DataFrames to separate sheets in a single Excel file, one can pass an `ExcelWriter`.

```python
with ExcelWriter('path_to_file.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

**Note:** Wringing a little more performance out of `read_excel` Internally, Excel stores all numeric data as floats. Because this can produce unexpected behavior when reading in data, pandas defaults to trying to convert integers to floats if it doesn’t lose information (1.0 --> 1). You can pass `convert_float=False` to disable this behavior, which may give a slight performance improvement.

### 24.4.2.2 Writing Excel Files to Memory

New in version 0.17.

Pandas supports writing Excel files to buffer-like objects such as `StringIO` or `BytesIO` using `ExcelWriter`. New in version 0.17.

Added support for Openpyxl >= 2.2

```python
# Safe import for either Python 2.x or 3.x
try:
    from io import BytesIO
except ImportError:
    from cStringIO import StringIO as BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = 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.

### 24.4.3 Excel writer engines

*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 and openpyxl for .xlsm files 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: This includes stable support for Openpyxl from 1.6.1. However, it is advised to use version 2.2 and higher, especially when working with styles.
- xlsxwriter
- xlwt

```python
# By setting the 'engine' in the DataFrame and Panel 'to_excel()' methods.
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='xlsxwriter')

# By setting the 'engine' in the ExcelWriter constructor.
writer = ExcelWriter('path_to_file.xlsx', engine='xlsxwriter')

# Or via pandas configuration.
from pandas import options
options.io.excel.xlsx.writer = 'xlsxwriter'
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

### 24.4.4 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)

#### 24.5 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_table 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()
```
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
In [302]: df = pd.DataFrame(randn(5,3))
In [303]: df
Out[303]:
     0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

```python
In [304]: df.to_clipboard()
In [305]: pd.read_clipboard()
Out[305]:
     0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

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 gtk or PyQt4 modules) on Linux to use these methods.

### 24.6 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.

```python
In [306]: df
Out[306]:
     0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

```python
In [307]: df.to_pickle('foo.pkl')
```

24.6. Pickling
The `read_pickle` function in the `pandas` namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```python
In [308]: pd.read_pickle('foo.pkl')
Out[308]:
      0    1    2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2  0.325417  1.238640 -1.210543
3 -1.525743 -0.172372 -0.734129
```

**Warning:** Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3.6/library/pickle.html

**Warning:** Several internal refactorings have been done while still preserving compatibility with pickles created with older versions of pandas. However, for such cases, pickled dataframes, series etc, must be read with `pd.read_pickle`, rather than `pickle.load`.

See here and here for some examples of compatibility-breaking changes. See this question for a detailed explanation.

### 24.6.1 Compressed pickle files

New in version 0.20.0.

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz` are supported for reading and writing. `zip` file supports read only and must contain only one data file to be read in.

The compression type can be an explicit parameter or be inferred from the file extension. If `infer`, then use `gzip`, `bz2`, `zip`, or `xz` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, or `'.xz'`, respectively.

```python
In [309]: df = pd.DataFrame({
       .....:     'A': np.random.randn(1000),
       .....:     'B': 'foo',
       .....:     'C': pd.date_range('20130101', periods=1000, freq='s'))
       .....:
In [310]: df
Out[310]:
       A       B       C
0  0.478412 foo 2013-01-01 00:00:00
1 -0.783748 foo 2013-01-01 00:00:01
2  1.403558 foo 2013-01-01 00:00:02
3 -0.539282 foo 2013-01-01 00:00:03
4 -1.651012 foo 2013-01-01 00:00:04
5  0.692072 foo 2013-01-01 00:00:05
6  1.022171 foo 2013-01-01 00:00:06
...     ...     ...     ...
993 -1.613932 foo 2013-01-01 00:16:33
994  1.088104 foo 2013-01-01 00:16:34
995 -0.632963 foo 2013-01-01 00:16:35
```
Using an explicit compression type

```
In [311]: df.to_pickle("data.pkl.compress", compression="gzip")
```

```
In [312]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")
```

```
In [313]: rt
```

```
Out[313]:
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.478412</td>
<td>foo 2013-01-01 00:00:00</td>
</tr>
<tr>
<td>1</td>
<td>-0.783748</td>
<td>foo 2013-01-01 00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>1.403558</td>
<td>foo 2013-01-01 00:00:02</td>
</tr>
<tr>
<td>3</td>
<td>-0.539282</td>
<td>foo 2013-01-01 00:00:03</td>
</tr>
<tr>
<td>4</td>
<td>-1.651012</td>
<td>foo 2013-01-01 00:00:04</td>
</tr>
<tr>
<td>5</td>
<td>0.692072</td>
<td>foo 2013-01-01 00:00:05</td>
</tr>
<tr>
<td>6</td>
<td>1.022171</td>
<td>foo 2013-01-01 00:00:06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>993</td>
<td>-1.613932</td>
<td>foo 2013-01-01 00:16:33</td>
</tr>
<tr>
<td>994</td>
<td>1.088104</td>
<td>foo 2013-01-01 00:16:34</td>
</tr>
<tr>
<td>995</td>
<td>-0.632963</td>
<td>foo 2013-01-01 00:16:35</td>
</tr>
<tr>
<td>996</td>
<td>-0.585314</td>
<td>foo 2013-01-01 00:16:36</td>
</tr>
<tr>
<td>997</td>
<td>-0.275038</td>
<td>foo 2013-01-01 00:16:37</td>
</tr>
<tr>
<td>998</td>
<td>-0.937512</td>
<td>foo 2013-01-01 00:16:38</td>
</tr>
<tr>
<td>999</td>
<td>0.632369</td>
<td>foo 2013-01-01 00:16:39</td>
</tr>
</tbody>
</table>

[1000 rows x 3 columns]

Inferring compression type from the extension

```
In [314]: df.to_pickle("data.pkl.xz", compression="infer")
```

```
In [315]: rt = pd.read_pickle("data.pkl.xz", compression="infer")
```

```
In [316]: rt
```

```
Out[316]:
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.478412</td>
<td>foo 2013-01-01 00:00:00</td>
</tr>
<tr>
<td>1</td>
<td>-0.783748</td>
<td>foo 2013-01-01 00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>1.403558</td>
<td>foo 2013-01-01 00:00:02</td>
</tr>
<tr>
<td>3</td>
<td>-0.539282</td>
<td>foo 2013-01-01 00:00:03</td>
</tr>
<tr>
<td>4</td>
<td>-1.651012</td>
<td>foo 2013-01-01 00:00:04</td>
</tr>
<tr>
<td>5</td>
<td>0.692072</td>
<td>foo 2013-01-01 00:00:05</td>
</tr>
<tr>
<td>6</td>
<td>1.022171</td>
<td>foo 2013-01-01 00:00:06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>993</td>
<td>-1.613932</td>
<td>foo 2013-01-01 00:16:33</td>
</tr>
<tr>
<td>994</td>
<td>1.088104</td>
<td>foo 2013-01-01 00:16:34</td>
</tr>
<tr>
<td>995</td>
<td>-0.632963</td>
<td>foo 2013-01-01 00:16:35</td>
</tr>
<tr>
<td>996</td>
<td>-0.585314</td>
<td>foo 2013-01-01 00:16:36</td>
</tr>
<tr>
<td>997</td>
<td>-0.275038</td>
<td>foo 2013-01-01 00:16:37</td>
</tr>
<tr>
<td>998</td>
<td>-0.937512</td>
<td>foo 2013-01-01 00:16:38</td>
</tr>
<tr>
<td>999</td>
<td>0.632369</td>
<td>foo 2013-01-01 00:16:39</td>
</tr>
</tbody>
</table>

[1000 rows x 3 columns]
The default is to ‘infer’

```
In [317]: df.to_pickle("data.pkl.gz")

In [318]: rt = pd.read_pickle("data.pkl.gz")

In [319]: rt
Out[319]:
  A  B  C
0 0.478412 foo 2013-01-01 00:00:00
1 -0.783748 foo 2013-01-01 00:00:01
2  1.403558 foo 2013-01-01 00:00:02
3 -0.539282 foo 2013-01-01 00:00:03
4  1.651012 foo 2013-01-01 00:00:04
5  0.692072 foo 2013-01-01 00:00:05
6  1.022171 foo 2013-01-01 00:00:06
...     ...     ...     ...
993 -1.613932 foo 2013-01-01 00:16:33
994  1.088104 foo 2013-01-01 00:16:34
995 -0.632963 foo 2013-01-01 00:16:35
996 -0.585314 foo 2013-01-01 00:16:36
997 -0.275038 foo 2013-01-01 00:16:37
998 -0.937512 foo 2013-01-01 00:16:38
999  0.632369 foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [320]: df["A"].to_pickle("s1.pkl.bz2")

In [321]: rt = pd.read_pickle("s1.pkl.bz2")

In [322]: rt
Out[322]:
  0  0.478412
  1 -0.783748
  2  1.403558
  3 -0.539282
  4  1.651012
  5  0.692072
  6  1.022171
  ...
  993 -1.613932
  994  1.088104
  995 -0.632963
  996 -0.585314
  997 -0.275038
  998 -0.937512
  999  0.632369

Name: A, Length: 1000, dtype: float64
24.7 msgpack

pandas supports the msgpack format for object serialization. This is a lightweight portable binary format, similar to binary JSON, that is highly space efficient, and provides good performance both on the writing (serialization), and reading (deserialization).

**Warning:** This is a very new feature of pandas. We intend to provide certain optimizations in the io of the msgpack data. Since this is marked as an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

As a result of writing format changes and other issues:

<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>

Reading (files packed by older versions) is backward-compatible, except for files packed with 0.17 in Python 2, in which case only they can only be unpacked in Python 2.

In [323]: df = pd.DataFrame(np.random.rand(5,2), columns=list('AB'))

In [324]: df.to_msgpack('foo.msg')

In [325]: pd.read_msgpack('foo.msg')
Out[325]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.170801</td>
<td>0.895366</td>
</tr>
<tr>
<td>1</td>
<td>0.838238</td>
<td>0.052592</td>
</tr>
<tr>
<td>2</td>
<td>0.664140</td>
<td>0.289750</td>
</tr>
<tr>
<td>3</td>
<td>0.449593</td>
<td>0.872087</td>
</tr>
<tr>
<td>4</td>
<td>0.983618</td>
<td>0.744359</td>
</tr>
</tbody>
</table>

In [326]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))

You can pass a list of objects and you will receive them back on deserialization.

In [327]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1,2,3]), s)

In [328]: pd.read_msgpack('foo.msg')
Out[328]:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.170801</td>
<td>0.895366</td>
<td>2013-01-01 0.548134</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.838238</td>
<td>0.052592</td>
<td>2013-01-02 0.503447</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.664140</td>
<td>0.289750</td>
<td>2013-01-03 0.348438</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.449593</td>
<td>0.872087</td>
<td>2013-01-04 0.707267</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.983618</td>
<td>0.744359</td>
<td>2013-01-05 0.261656</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You can pass `iterator=True` to iterate over the unpacked results

```python
In [329]: for o in pd.read_msgpack('foo.msg', iterator=True):
.....:     print(o)
.....:
   A     B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359

foo
[1 2 3]
2013-01-01  0.548134
2013-01-02  0.503447
2013-01-03  0.348438
2013-01-04  0.707267
2013-01-05  0.261656
Freq: D, dtype: float64
```

You can pass `append=True` to the writer to append to an existing pack

```python
In [330]: df.to_msgpack('foo.msg', append=True)
In [331]: pd.read_msgpack('foo.msg')
Out[331]:

{'dict': ({'df': A     B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359}, 'foo', array([1, 2, 3]), 2013-01-01  0.548134
2013-01-02  0.503447
2013-01-03  0.348438
2013-01-04  0.707267
2013-01-05  0.261656
Freq: D, dtype: float64, A     B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359
```

Unlike other io methods, `to_msgpack` is available on both a per-object basis, `df.to_msgpack()` and using the top-level `pd.to_msgpack(...)` where you can pack arbitrary collections of python lists, dicts, scalars, while intermixing pandas objects.

```python
In [332]: pd.to_msgpack('foo2.msg', { 'dict' : [ { 'df' : df }, { 'string' : 'foo' }, { 'scalar' : 1. }, { 's' : s } ] })
In [333]: pd.read_msgpack('foo2.msg')
Out[333]:

{'dict': ({'df': A     B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359
```

Chapter 24. IO Tools (Text, CSV, HDF5, ...
24.7.1 Read/Write API

Msgpacks can also be read from and written to strings.

```python
In [334]: df.to_msgpack()
Out[334]:

Furthermore you can concatenate the strings to produce a list of the original objects.

```python
In [335]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[335]:

```

24.8 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 requires PyTables >= 3.0.0. There is a indexing bug in PyTables < 3.2 which may appear when querying stores using an index. If you see a subset of results being returned, upgrade to PyTables >= 3.2. Stores created previously will need to be rewritten using the updated version.
Warning: As of version 0.17.0, HDFStore will not drop rows that have all missing values by default. Previously, if all values (except the index) were missing, HDFStore would not write those rows to disk.

In [336]: store = pd.HDFStore('store.h5')

In [337]: 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 [338]: np.random.seed(1234)

In [339]: index = pd.date_range('1/1/2000', periods=8)

In [340]: s = pd.Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [341]: df = pd.DataFrame(randn(8, 3), index=index,
                                 columns=['A', 'B', 'C'])

In [342]: wp = pd.Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
                             major_axis=pd.date_range('1/1/2000', periods=5),
                             minor_axis=['A', 'B', 'C', 'D'])

# store.put('s', s) is an equivalent method
In [343]: store['s'] = s

In [344]: store['df'] = df

In [345]: store['wp'] = wp

# the type of stored data
In [346]: store.root.wp._v_attrs.pandas_type
Out[346]: 'wide'

In [347]: store
Out[347]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In a current or later Python session, you can retrieve stored objects:

# store.get('df') is an equivalent method
In [348]: store['df']
Out[348]:
       A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109
# dotted (attribute) access provides get as well

```python
In [349]: store.df
```

```
  -->
       A     B     C
2000-01-01 0.887163 0.859588 -0.636524
2000-01-02 0.015696 -2.242685 1.150036
2000-01-03 0.991946 0.953324 -2.021255
2000-01-04 -0.334077 0.002118 0.405453
2000-01-05 0.289092 1.321158 -1.546906
2000-01-06 -0.202646 -0.655969 0.193421
2000-01-07 0.553439 1.318152 -0.469305
2000-01-08 0.675554 -1.817027 -0.183109
```

**Deletion of the object specified by the key**

```python
# store.remove('wp') is an equivalent method

In [350]: del store['wp']

In [351]: store
```

```
Out[351]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

**Closing a Store, Context Manager**

```python
In [352]: store.close()

In [353]: store
```

```
Out[353]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

```python
In [354]: store.is_open
```

```
Out[354]: False
```

# Working with, and automatically closing the store with the context
# manager

```python
In [355]: with pd.HDFStore('store.h5') as store:
    .....:   store.keys()
    .....:
```

## 24.8.1 Read/Write API

HDFStore supports an *top-level* API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work.

```python
In [356]: df_tl = pd.DataFrame(dict(A=list(range(5)), B=list(range(5))))

In [357]: df_tl.to_hdf('store_tl.h5', 'table', append=True)

In [358]: pd.read_hdf('store_tl.h5', 'table', where = ['index>2'])
```

```
Out[358]:
     A  B
3  3  3
4  4  4
```
As of version 0.17.0, HDFStore will no longer drop rows that are all missing by default. This behavior can be enabled by setting dropna=True.

In [359]: df_with_missing = pd.DataFrame({'col1':[0, np.nan, 2],
                              'col2':[1, np.nan, np.nan]})
In [360]: df_with_missing
Out[360]:
   col1  col2
0   0.0   1.0
1  NaN   NaN
2   2.0  NaN

In [361]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
                              format = 'table', mode='w')
In [362]: pd.read_hdf('file.h5', 'df_with_missing')
Out[362]:
   col1  col2
0   0.0   1.0
1  NaN   NaN
2   2.0  NaN

In [363]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
                              format = 'table', mode='w', dropna=True)
In [364]: pd.read_hdf('file.h5', 'df_with_missing')
Out[364]:
   col1  col2
0   0.0   1.0
2   2.0  NaN

This is also true for the major axis of a Panel:

In [365]: matrix = [[[np.nan, np.nan, np.nan], [1, np.nan, np.nan]],
                   [[np.nan, np.nan, np.nan], [np.nan, 5, 6]],
                   [[np.nan, np.nan, np.nan], [np.nan, 5, np.nan]]]
In [366]: panel_with_major_axis_all_missing = pd.Panel(matrix,
                                              items=['Item1', 'Item2', 'Item3'],
                                              major_axis=[1,2],
                                              minor_axis=['A', 'B', 'C'])
In [367]: panel_with_major_axis_all_missing
Out[367]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 1 to 2
Minor_axis axis: A to C

In [368]: panel_with_major_axis_all_missing.to_hdf('file.h5', 'panel',
                                              dropna = True,
                                              format='table',
                                              mode='w')
24.8.2 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.

```python
pd.DataFrame(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
```

24.8.3 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 & 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.

```python
In [371]: store = pd.HDFStore('store.h5')
In [372]: df1 = df[0:4]
In [373]: df2 = df[4:]

# append data (creates a table automatically)
In [374]: store.append('df', df1)
In [375]: store.append('df', df2)
In [376]: store
Out[376]: <class 'pandas.io.pytables.HDFStore'>
```
# select the entire object

```
In [377]: store.select('df')
```

```
\nOut[377]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.887163</td>
<td>0.859588</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.015696</td>
<td>-2.242685</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.991946</td>
<td>0.953324</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.334077</td>
<td>0.002118</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.289092</td>
<td>1.321158</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.202646</td>
<td>-0.655969</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.553439</td>
<td>1.318152</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.675554</td>
<td>-1.817027</td>
</tr>
</tbody>
</table>
```

# the type of stored data

```
In [378]: store.root.df._v_attrs.pandas_type
```

```
˓→'frame_table'
```

**Note:** You can also create a table by passing `format='table'` or `format='t'` to a `put` operation.

## 24.8.4 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 [379]: store.put('foo/bar/bah', df)
In [380]: store.append('food/orange', df)
In [381]: store.append('food/apple', df)
In [382]: store
Out[382]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [383]: store.keys()
```

```
\nOut[383]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']
```

# remove all nodes under this level

```
In [384]: store.remove('food')
In [385]: store
Out[385]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```
**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), ...

Instead, use explicit string based keys
```

```
In [386]: store['foo/bar/bah']
Out[386]:
   A         B         C
0  0.887163  0.859588  0.636524
1  0.015696 -2.242685  1.150036
2  0.991946  0.953324 -2.021255
3 -0.334077  0.002118  0.405453
4  0.289092  1.321158 -1.546906
5 -0.202646 -0.655969  0.193421
6  0.553439  1.318152 -0.469305
7  0.675554 -1.817027 -0.183109
```

### 24.8.5 Storing Types

#### 24.8.5.1 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 [387]: df_mixed = pd.DataFrame({ 'A' : randn(8),
   ....:     'B' : randn(8),
   ....:     'C' : np.array(randn(8),dtype='float32'),
   ....:     'string' : 'string',
   ....:     'int' : 1,
   ....:     'bool' : True,
   ....:     'datetime64' : pd.Timestamp('20010102')},
   ....:     index=list(range(8))
   ....:)

In [388]: df_mixed.loc[df_mixed.index[3:5], ['A', 'B', 'string', 'datetime64']] = np.nan

In [389]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})

In [390]: df_mixed1 = store.select('df_mixed')
```
24.8.5.2 Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.

```python
In [394]: 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 [395]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index, columns=['A', 'B', 'C'])
```
In [396]: df_mi
Out[396]:
     A    B    C
foo bar
    one  -0.584718  0.816594 -0.081947
    two  -0.344766  0.528288 -1.068989
    three -0.511881  0.291205  0.566534
bar one  0.503592  0.285296  0.484288
    two  1.363482 -0.781105 -0.468018
baz two  1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
qux one -0.203933 -0.182175  0.680656
    two  1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884
In [397]: store.append('df_mi',df_mi)
In [398]: store.select('df_mi')
Out[398]:
     A    B    C
foo bar
    one  -0.584718  0.816594 -0.081947
    two  -0.344766  0.528288 -1.068989
    three -0.511881  0.291205  0.566534
bar one  0.503592  0.285296  0.484288
    two  1.363482 -0.781105 -0.468018
baz two  1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
qux one -0.203933 -0.182175  0.680656
    two  1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

# the levels are automatically included as data columns
In [399]: store.select('df_mi', 'foo=bar')

24.8.6 Querying

24.8.6.1 Querying a Table

**Warning:** This query capabilities have changed substantially starting in 0.13.0. Queries from prior version are accepted (with a `DeprecationWarning`) printed if its not string-like.

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 a DataFrame
• major_axis, minor_axis, and items are supported indexers of the Panel
• if data_columns are specified, these can be used as additional indexers

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:
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
string = "HolyMoly"
store.select('df', 'index == string')

instead of this

string = "HolyMoly"
store.select('df', 'index == %s % 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 [400]: dfq = pd.DataFrame(randn(10,4),columns=list('ABCD'),index=pd.date_range('20130101',periods=10))
In [401]: store.append('dfq',dfq,format='table',data_columns=True)

Use boolean expressions, with in-line function evaluation.

In [402]: store.select('dfq',"index>pd.Timestamp('20130104') & columns=['A', 'B']")

Out[402]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-05</td>
<td>1.210384</td>
<td>0.797435</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-0.850346</td>
<td>1.176812</td>
</tr>
<tr>
<td>2013-01-07</td>
<td>0.984188</td>
<td>-0.121728</td>
</tr>
<tr>
<td>2013-01-08</td>
<td>0.796595</td>
<td>-0.474021</td>
</tr>
<tr>
<td>2013-01-09</td>
<td>-0.804834</td>
<td>-2.123620</td>
</tr>
<tr>
<td>2013-01-10</td>
<td>0.334198</td>
<td>0.536784</td>
</tr>
</tbody>
</table>

Use and inline column reference

In [403]: store.select('dfq',where="A>0 or C>0")

Out[403]:

<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>0.436258</td>
<td>-1.703013</td>
<td>0.393711</td>
<td>-0.479324</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.299016</td>
<td>0.694103</td>
<td>0.678630</td>
<td>0.239556</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.151227</td>
<td>0.816127</td>
<td>1.893534</td>
<td>0.639633</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.962029</td>
<td>-2.085266</td>
<td>1.930247</td>
<td>-1.735349</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>1.210384</td>
<td>0.797435</td>
<td>-0.379811</td>
<td>0.702562</td>
</tr>
<tr>
<td>2013-01-07</td>
<td>0.984188</td>
<td>-0.121728</td>
<td>2.365769</td>
<td>0.496143</td>
</tr>
<tr>
<td>2013-01-08</td>
<td>0.796595</td>
<td>-0.474021</td>
<td>-0.056696</td>
<td>1.357797</td>
</tr>
<tr>
<td>2013-01-10</td>
<td>0.334198</td>
<td>0.536784</td>
<td>-0.743830</td>
<td>-0.320204</td>
</tr>
</tbody>
</table>

Works with a Panel as well.

In [404]: store.append('wp',wp)
In [405]: store
Out[405]:
<class 'pandas.io.pytables.HDFStore'>
**In [406]:** store.select('wp', "major_axis>pd.Timestamp('20000102') & minor_axis=['A', 'B']")

```
<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
```

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 [407]:** store.select('df', "columns=['A', 'B']")

```
   A    B
2000-01-01  0.887163  0.859588
2000-01-02  0.015696 -2.242685
2000-01-03  0.991946  0.953324
2000-01-04 -0.334077  0.002118
2000-01-05  0.289092  1.321158
2000-01-06 -0.202646 -0.655969
2000-01-07  0.553439  1.318152
2000-01-08  0.675554 -1.817027
```

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.

```
# this is effectively what the storage of a Panel looks like
In [408]: wp.to_frame()
```

```
   Item1    Item2
major minor
2000-01-01  A  1.058969  0.215269
           B -0.397840  0.841009
           C  0.337438 -1.445810
           D  1.047579 -1.401973
2000-01-02  A  1.045938 -0.100918
           B  0.863717 -0.548242
           C -0.122092 -0.144620
           D  0.247792  1.033801
2000-01-04  B  0.036142  0.307969
           C -2.074978 -0.208499
           D  0.247792  1.033801
2000-01-05  A -0.897157 -2.400454
           B -0.136795  2.030604
           C  0.018289 -1.142631
           D  0.755414  0.211883
```

[20 rows x 2 columns]

```
# limiting the search
In [409]: store.select('wp',"major_axis>20000102 & minor_axis=['A', 'B']", start=0, stop=10)
```
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.

### 24.8.6.2 Using `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 [410]: from datetime import timedelta
In [411]: dftd = pd.DataFrame(dict(A = pd.Timestamp('20130101'), B = [ pd.Timestamp('20130101') + timedelta(days=i,seconds=10) for i in range(10) ]))
In [412]: dftd['C'] = dftd['A']-dftd['B']
In [413]: dftd
Out[413]:
    A            B          C
0 2013-01-01 00:00:10  00:00:00 -1 days +23:59:50
1 2013-01-01 00:00:10  00:00:00  0 days +23:59:50
2 2013-01-01 00:00:10  00:00:00  1 days +23:59:50
3 2013-01-01 00:00:10  00:00:00  2 days +23:59:50
4 2013-01-01 00:00:10  00:00:00  3 days +23:59:50
5 2013-01-01 00:00:10  00:00:00  4 days +23:59:50
6 2013-01-01 00:00:10  00:00:00  5 days +23:59:50
7 2013-01-01 00:00:10  00:00:00  6 days +23:59:50
8 2013-01-01 00:00:10  00:00:00  7 days +23:59:50
9 2013-01-01 00:00:10  00:00:00  8 days +23:59:50

In [414]: store.append('dftd',dftd,data_columns=True)
In [415]: store.select('dftd','C<'-3.5D'')
Out[415]:
    A            B          C
0 2013-01-01 00:00:10  00:00:00 -5 days +23:59:50
1 2013-01-01 00:00:10  00:00:00 -6 days +23:59:50
2 2013-01-01 00:00:10  00:00:00 -7 days +23:59:50
3 2013-01-01 00:00:10  00:00:00 -8 days +23:59:50
4 2013-01-01 00:00:10  00:00:00 -9 days +23:59:50
5 2013-01-01 00:00:10  00:00:00 -10 days +23:59:50
6 2013-01-01 00:00:10  00:00:00 -11 days +23:59:50
7 2013-01-01 00:00:10  00:00:00 -12 days +23:59:50
8 2013-01-01 00:00:10  00:00:00 -13 days +23:59:50
9 2013-01-01 00:00:10  00:00:00 -14 days +23:59:50
10 2013-01-01 00:00:10 00:00:00 -15 days +23:59:50
11 2013-01-01 00:00:10 00:00:00 -16 days +23:59:50
12 2013-01-01 00:00:10 00:00:00 -17 days +23:59:50
13 2013-01-01 00:00:10 00:00:00 -18 days +23:59:50
14 2013-01-01 00:00:10 00:00:00 -19 days +23:59:50
15 2013-01-01 00:00:10 00:00:00 -20 days +23:59:50

24.8. HDF5 (PyTables)
24.8.6.3 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 [416]: i = store.root.df.table.cols.index.index

In [417]: i.optlevel, i.kind
Out[417]: (6, 'medium')

# change an index by passing new parameters
In [418]: store.create_table_index('df', optlevel=9, kind='full')

In [419]: i = store.root.df.table.cols.index.index

In [420]: i.optlevel, i.kind
Out[420]: (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 [421]: df_1 = pd.DataFrame(randn(10,2),columns=list('AB'))

In [422]: df_2 = pd.DataFrame(randn(10,2),columns=list('AB'))

In [423]: st = pd.HDFStore('appends.h5',mode='w')

In [424]: st.append('df', df_1, data_columns=['B'], index=False)

In [425]: st.append('df', df_2, data_columns=['B'], index=False)

In [426]: st.get_storer('df').table
Out[426]:
/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.

```python
In [427]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [428]: st.get_storer('df').table
Out[428]:
/df/table (Table(20,)) ''
   description := {
       "index": Int64Col(shape=(), dflt=0, pos=0),
       "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
```
See here for how to create a completely-sorted-index (CSI) on an existing store.

### 24.8.6.4 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 [430]: df_dc = df.copy()
In [431]: df_dc['string'] = 'foo'
In [432]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan
In [433]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'
In [434]: df_dc['string2'] = 'cool'
In [435]: df_dc.loc[df_dc.index[1:3], ['B','C']] = 1.0
In [436]: df_dc
```

```
Out[436]:
          A         B         C     string  string2
2000-01-01 0.887163  0.859588 -0.636524    foo     cool
2000-01-02 0.015696  1.000000  1.000000    foo     cool
2000-01-03 0.991946  1.000000  1.000000    foo     cool
2000-01-04 -0.334077  0.002118  0.405453    foo     cool
2000-01-05 0.289092  1.321158 -1.546906 NaN     cool
2000-01-06 -0.202646 -0.655969  0.193421 NaN     cool
2000-01-07 0.553439  1.318152 -0.469305    foo     cool
2000-01-08 0.675554 -1.817027 -0.183109    bar     cool
```

# on-disk operations
```
In [437]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])
In [438]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
```

```
Out[438]:
          A         B         C     string  string2
2000-01-01 0.887163  0.859588 -0.636524    foo     cool
2000-01-02 0.015696  1.000000  1.000000    foo     cool
2000-01-03 0.991946  1.000000  1.000000    foo     cool
2000-01-04 -0.334077  0.002118  0.405453    foo     cool
2000-01-05 0.289092  1.321158 -1.546906 NaN     cool
2000-01-06 -0.202646 -0.655969  0.193421 NaN     cool
2000-01-07 0.553439  1.318152 -0.469305    foo     cool
2000-01-08 0.675554 -1.817027 -0.183109    bar     cool
```

# getting creative
```
In [439]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
```

```
Out[439]:
          A         B         C     string  string2
2000-01-01 0.887163  0.859588 -0.636524    foo     cool
2000-01-02 0.015696  1.000000  1.000000    foo     cool
2000-01-03 0.991946  1.000000  1.000000    foo     cool
2000-01-04 -0.334077  0.002118  0.405453    foo     cool
2000-01-05 0.289092  1.321158 -1.546906 NaN     cool
2000-01-06 -0.202646 -0.655969  0.193421 NaN     cool
2000-01-07 0.553439  1.318152 -0.469305    foo     cool
2000-01-08 0.675554 -1.817027 -0.183109    bar     cool
```
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!)

### 24.8.6.5 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 [442]: for df in store.select('df', chunksize=3):
    ....:     print(df)
    ....:
   A   B   C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
   A   B   C
2000-01-04 -0.334077  0.002118  0.405453
```
Note: You can also use the iterator with read_hdf which will open, then automatically close the store when finished iterating.

```python
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.

```python
In [443]: df = pd.DataFrame({'number': np.arange(1,11)})

In [444]: df
Out[444]:
   number
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10

In [445]: store.append('dfeq', df, data_columns=['number'])

In [446]: def chunks(l, n):
       ....:     return [l[i:i+n] for i in range(0, len(l), n)]
       ....:

In [447]: evens = [2,4,6,8,10]

In [448]: coordinates = store.select_as_coordinates('dfeq','number=evens')

In [449]: for c in chunks(coordinates, 2):
       ....:     print(store.select('dfeq',where=c))
       ....:
   number
1  2
3  4
   number
5  6
7  8
   number
9 10
```
24.8.6.6 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 [450]: store.select_column('df_dc', 'index')
Out[450]:
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 [451]: store.select_column('df_dc', 'string')
Out[451]:
\n0  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 [452]: df_coord = pd.DataFrame(np.random.randn(1000,2),index=pd.date_range('20000101',periods=1000))
In [453]: store.append('df_coord',df_coord)
In [454]: c = store.select_as_coordinates('df_coord','index>20020101')
In [455]: c.summary()
Out[455]: 'Int64Index: 268 entries, 732 to 999'
In [456]: store.select('df_coord',where=c)
Out[456]:
\n0  2002-01-02 -0.178266 -0.064638
1  2002-01-03 -1.204956 -3.880898
2  2002-01-04  0.974470  0.415160
3  2002-01-05  1.751967  0.485011
4  2002-01-06 -0.170894  0.748870
```

1106 Chapter 24. IO Tools (Text, CSV, HDF5, ...)
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.

```python
In [457]: df_mask = pd.DataFrame(np.random.randn(1000,2),index=pd.date_range('20000101', periods=1000))

In [458]: store.append('df_mask',df_mask)

In [459]: c = store.select_column('df_mask','index')

In [460]: where = c[pd.DatetimeIndex(c).month==5].index

In [461]: store.select('df_mask',where=where)

Out[461]:
   0         1
2000-05-01 -1.006245 -0.616759
2000-05-02  0.218940  0.717838
2000-05-03  0.013333  1.348060
2000-05-04  0.662176 -1.050645
2000-05-05 -1.034870 -0.243242
2000-05-06 -0.753366 -1.454329
2000-05-07 -1.022920 -0.476989
              ...         ...
2002-05-25 -0.509090 -0.389376
2002-05-26  0.150674  1.164337
2002-05-27 -0.332944  0.115181
2002-05-28 -1.048127 -0.605733
2002-05-29  1.418754 -0.442835
2002-05-30 -0.433200  0.835001
2002-05-31 -1.041278  1.401811

[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.
24.8.6.7 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.
24.8.7 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. This is especially true in higher dimensional objects (Panel and Panel4D). 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.

```python
# returns the number of rows deleted
In [471]: store.remove('wp', 'major_axis>20000102')
Out[471]: 12

In [472]: store.select('wp')
\class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

**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.

---

24.8. HDF5 (PyTables)
24.8.8 Notes & Caveats

24.8.8.1 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 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.

New in version 0.20.2: 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)
```
24.8.8.2 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.

```
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.

24.8.8.3 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) 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 its items (Panel) / 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.

24.8.9 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></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.
24.8.9.1 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.

```
In [473]: dfcat = pd.DataFrame({ 'A' : pd.Series(list('aabbcdba')).astype('category'),
                          'B' : np.random.randn(8) })

In [474]: dfcat
Out[474]:
   A   B
0  a  0.603273
1  a  0.262554
2  b  0.979586
3  b  2.132387
4  c  0.892485
5  d  1.996474
6  b  0.231425
7  a  0.980070

In [475]: dfcat.dtypes
Out[475]:
A    category
B    float64
dtype: object

In [476]: cstore = pd.HDFStore('cats.h5', mode='w')

In [477]: cstore.append('dfcat', dfcat, format='table', data_columns=[A'])

In [478]: result = cstore.select('dfcat', where="A in ['b','c']")

In [479]: result
Out[479]:
   A   B
2  b  -0.979586
3  b   2.132387
4  c   0.892485
6  b   0.231425

In [480]: result.dtypes
Out[480]:
A    category
B    float64
dtype: object
```

24.8.9.2 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

```
In [481]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'},index=list(range(5)))

In [482]: dfs
Out[482]:
   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 [483]: store.append('dfs', dfs, min_itemsize = 30)

In [484]: store.get_storer('dfs').table
       '/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, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [485]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })

In [486]: store.get_storer('dfs2').table
       '/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, medium, shuffle, zlib(1)).is_csi=False,
         "A": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```
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.

```python
In [487]: dfss = pd.DataFrame(dict(A = ['foo','bar','nan']))

In [488]: dfss
Out[488]:
   A
0  foo
1  bar
2  nan

In [489]: store.append('dfss', dfss)

In [490]: store.select('dfss')
Out[490]:
   A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [491]: store.append('dfss2', dfss, nan_rep='_nan_')

In [492]: store.select('dfss2')
Out[492]:
   A
0  foo
1  bar
2  nan
```

### 24.8.10 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:

```python
In [493]: np.random.seed(1)

In [494]: 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 [495]: df_for_r.head()
Out[495]:
   class  first  second
0     0  0.417022  0.326645
1     0  0.720324  0.527058
2     1  0.000114  0.885942
3     1  0.302333  0.357270
4     1  0.146756  0.908535

In [496]: store_export = pd.HDFStore('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 =负载hdf5data("transfer.hdf5")
> head(data)
  first    second  class
1  0.4170220047  0.3266449  0
2  0.7203244934  0.5270581  0
3  0.0001143748  0.8859421  1
4  0.3023325726  0.3572698  1
5  0.1467558908  0.9085352  1
6  0.0923385948  0.6233601  1
```

**Note:** The R function lists the entire HDF5 file’s contents and assembles the `data.frame` object from all matching nodes, so use this only as a starting point if you have stored multiple `DataFrame` objects to a single HDF5 file.
24.8.11 Performance

- The `tables` format comes 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 expected rows that PyTables will expected. 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.

24.9 Feather

New in version 0.20.0.

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.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.

- 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 simply `.reset_index()` in order to store the index.

- Duplicate column names and non-string columns names are not supported

- Non supported types include `Period` and actual python object types. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation

```python
In [499]: 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.Categorical(list('abc')),
    ....:     'g': pd.date_range('20130101', periods=3),
    ....:     'h': pd.date_range('20130101', periods=3, tz='US/Eastern'),
    ....:     'i': pd.date_range('20130101', periods=3, freq='ns')})

In [500]: df
Out[500]:
   a   b   c   d       e    f     g                  h          i
0  a  1  3  4.0   True  2013-01-01 00:00:00-05:00 2013-01-01
1  b  2  4  5.0  False  2013-01-02 00:00:00-05:00 2013-01-01
```
Write to a feather file.

In [502]: df.to_feather('example.feather')

Read from a feather file.

In [503]: result = pd.read_feather('example.feather')

In [504]: result

Out[504]:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
<td>3</td>
<td>4.0</td>
<td>True</td>
<td>a</td>
<td>2013-01-01 00:00:00-05:00</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>2</td>
<td>4</td>
<td>5.0</td>
<td>False</td>
<td>b</td>
<td>2013-01-02 00:00:00-05:00</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>3</td>
<td>5</td>
<td>6.0</td>
<td>True</td>
<td>c</td>
<td>2013-01-03 00:00:00-05:00</td>
</tr>
</tbody>
</table>

# we preserve dtypes

In [505]: result.dtypes

Out[505]:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>object</td>
<td>b</td>
<td>int64</td>
<td>c</td>
<td>uint8</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>dtype: object</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**24.10 Parquet**

New in version 0.21.0.

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.

- 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 simply .reset_index(drop=True) in order to store the index.
- Duplicate column names and non-string column names are not supported
- Categorical dtypes can be serialized to parquet, but will de-serialize as object dtype.
- Non supported types include Period and actual python object types. These will raise a helpful error message on an attempt at serialization.

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 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. These libraries differ by having different underlying dependencies (fastparquet by using numba, while pyarrow uses a c-library).
### 24.11 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_sql_table</code></td>
<td>Read SQL database table into a DataFrame.</td>
</tr>
<tr>
<td><code>read_sql_query</code></td>
<td>Read SQL query into a DataFrame.</td>
</tr>
<tr>
<td><code>read_sql</code></td>
<td>Read SQL query or database table into a DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.to_sql</code></td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
</tbody>
</table>

#### 24.11.1 pandas.read_sql_table

`pandas.read_sql_table(table_name, con[, schema, ...])`  
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** : string  
  Name of SQL table in database.  

- **con** : SQLAlchemy connectable (or database string URI)  
  SQLite DBAPI connection mode not supported.  

- **schema** : string, default None
Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

**index_col** : string or list of strings, optional, default: None

Column(s) to set as index(MultiIndex).

**coerce_float** : boolean, 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

**See also:**

*read_sql_query* Read SQL query into a DataFrame.

*read_sql*

**Notes**

Any datetime values with time zone information will be converted to UTC.

### 24.11.2 pandas.read_sql_query

**pandas.read_sql_query** *(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=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** : string SQL query or SQLAlchemy Selectable (select or text object)

  SQL query to be executed.

- **con** : SQLAlchemy connectable(engine/connection), database string URI, or sqlite3 DBAPI2 connection Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.
index_col : string or list of strings, optional, default: None

Column(s) to set as index(MultiIndex).

coerce_float : boolean, 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\}% 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.

Returns DataFrame

See also:

read_sql_table Read SQL database table into a DataFrame.

read_sql

Notes

Any datetime values with time zone information parsed via the parse_dates parameter will be converted to UTC.

24.11.3 pandas.read_sql

pandas.read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None)

Read SQL query or database table into a DataFrame.

Parameters sql : string or SQLAlchemy Selectable (select or text object)

SQL query to be executed.

con : SQLAlchemy connectable(engine/connection) or database string URI

or DBAPI2 connection (fallback mode) Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col : string or list of strings, optional, default: None
Column(s) to set as index(MultiIndex).

coerce_float : boolean, 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

See also:
read_sql_table Read SQL database table into a DataFrame.
read_sql_query Read SQL query into a DataFrame.

Notes

This function is a convenience wrapper around read_sql_table and read_sql_query (and for backward compatibility) and will delegate to the specific function depending on the provided input (database table name or SQL query). The delegated function might have more specific notes about their functionality not listed here.

24.11.4 pandas.DataFrame.to_sql

DataFrame.to_sql(name, con, flavor=None, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None)
Write records stored in a DataFrame to a SQL database.

Parameters name : string
Name of SQL table
con : SQLAlchemy engine or DBAPI2 connection (legacy mode)
Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

**flavor** : ‘sqlite’, default None

Deprecated since version 0.19.0: ‘sqlite’ is the only supported option if SQLAlchemy is not used.

**schema** : string, default None

Specify the schema (if database flavor supports this). If None, use default schema.

**if_exists** : {'fail', 'replace', 'append'}, default 'fail'

- fail: If table exists, do nothing.
- replace: If table exists, drop it, recreate it, and insert data.
- append: If table exists, insert data. Create if does not exist.

**index** : boolean, default True

Write DataFrame index as a column.

**index_label** : string 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, default None

If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

**dtype** : dict of column name to SQL type, default None

Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

---

**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 [514]: from sqlalchemy import create_engine

# Create your engine.
In [515]: engine = create_engine('sqlite:///::memory:)
```

If you want to manage your own connections you can pass one of those instead:

```python
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```
24.11.5 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>

```
In [516]: data.to_sql('data', engine)
```

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:

```
In [517]: data.to_sql('data_chunked', engine, chunksize=1000)
```

24.11.5.1 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:

```
In [518]: from sqlalchemy.types import String
In [519]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

**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.

24.11.6 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.

```
In [520]: pd.read_sql_table('data', engine)
Out[520]:
     index  id    Date  Col_1  Col_2  Col_3
0    0     26 2010-10-18    X  27.50   True
```
You can also specify the name of the column as the DataFrame index, and specify a subset of columns to be read.

```
In [521]: pd.read_sql_table('data', engine, index_col='id')
Out[521]:
       Date  Col_1  Col_2  Col_3
index
id
26  2010-10-18    X 27.50   True
42  2010-10-19   Y -12.50  False
63  2010-10-20    Z  5.73   True
```

```
In [522]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])

<table>
<thead>
<tr>
<th></th>
<th>Col_1</th>
<th>Col_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>27.50</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>-12.50</td>
</tr>
<tr>
<td>2</td>
<td>Z</td>
<td>5.73</td>
</tr>
</tbody>
</table>
```

And you can explicitly force columns to be parsed as dates:

```
In [523]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[523]:
       Date  id     Col_1  Col_2  Col_3
index
0  2010-10-18  26    X 27.50   True
1  2010-10-19  42    Y -12.50  False
2  2010-10-20  63    Z  5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to `pandas.to_datetime()`:

```
pd.read_sql_table('data', engine, parse_dates={'Date': '%Y-%m-%d'})
pd.read_sql_table('data', engine, parse_dates={'Date': {'format': '%Y-%m-%d %H:%M:%S'}})
```

You can check if a table exists using `has_table()`

### 24.11.7 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:

```
df.to_sql('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')
```

### 24.11.8 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.
In [524]: pd.read_sql_query('SELECT * FROM data', engine)
Out[524]:

<table>
<thead>
<tr>
<th>index</th>
<th>id</th>
<th>Date</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2010-10-18</td>
<td>X</td>
<td>27.50</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2010-10-19</td>
<td>Y</td>
<td>-12.50</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2010-10-20</td>
<td>Z</td>
<td>5.73</td>
<td>1</td>
</tr>
</tbody>
</table>

Of course, you can specify a more “complex” query.

In [525]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engine)
Out[525]:

<table>
<thead>
<tr>
<th>id</th>
<th>Col_1</th>
<th>Col_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Y</td>
<td>-12.5</td>
</tr>
</tbody>
</table>

The `read_sql_query()` function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

In [526]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [527]: df.to_sql('data_chunks', engine, index=False)

In [528]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks", engine, chunksize=5):
   ....:     print(chunk)
   ....:
   a    b    c
| 0 | 0.280665 | -0.073113 | 1.160339 |
| 1 | 0.369493 | 1.904659  | 1.111057 |
| 2 | 0.659050 | -1.627438 | 0.602319 |
| 3 | 0.420282 | 0.810952  | 1.044442 |
| 4 | -0.400878| 0.824006  | -0.562305|
   a    b    c
| 0 | 1.954878 | -1.331952 | -1.760689|
| 1 | -1.650721| -0.890556 | -1.119115|
| 2 | 1.956079 | -0.326499 | -1.342676|
| 3 | 1.114383 | -0.586524 | -1.236853|
| 4 | 0.875839 | 0.623362  | -0.434957|
   a    b    c
| 0 | 1.407540 | 0.129102  | 1.616950 |
| 1 | 0.502741 | 1.558806  | 0.109403 |
| 2 | -1.219744| 2.449369  | -0.545774|
| 3 | -0.198838| -0.700399 | -0.203394|
| 4 | 0.242669 | 0.201830  | 0.661020 |
   a    b    c
| 0 | 1.792158 | -0.120465 | -1.233121|
| 1 | -1.182318| -0.665755 | -1.674196|
| 2 | 0.825030 | -0.498214 | -0.310985|
| 3 | -0.001891| -1.396620 | -0.861316|
| 4 | 0.674712 | 0.618539  | -0.443172|

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)])

24.11.9 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

24.11.10 Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```python
In [529]: import sqlalchemy as sa

In [530]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'), engine, params={'col1': 'X'})
Out[530]:
               index  id  Date       Col_1  Col_2  Col_3
0  2010-10-18  00:00:00.000000  0  26  X  27.5  1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```python
In [531]: metadata = sa.MetaData()

In [532]: 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),  
   ....: )
```

24.11. SQL Queries
In [533]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 == True), engine)
Out[533]:
   index  Date Col_1   Col_2  Col_3
0       0  2010-10-18    X   27.50   True
1       2  2010-10-20    Z    5.73   True

You can combine SQLAlchemy expressions with parameters passed to `read_sql()` using `sqlalchemy.bindparam()`.

In [534]: import datetime as dt
In [535]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date'))
In [536]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[536]:
   index  Date Col_1   Col_2  Col_3
0       1  2010-10-19    Y  -12.50   False
1       2  2010-10-20    Z    5.73   True

### 24.11.11 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:

- `data.to_sql('data', cnx)`
- `pd.read_sql_query("SELECT * FROM data", con)`

### 24.12 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. 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.
24.13 Stata Format

24.13.1 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).

```python
In [537]: df = pd.DataFrame(randn(10, 2), columns=list('AB'))
In [538]: 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 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`.

24.13.2 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 [539]: pd.read_stata('stata.dta')
Out[539]:
    index    A          B
0        0  1.810535  -1.305727
1        1 -0.344987  -0.230840
2        2 -2.793085   1.937529
3        3  0.366332  -1.044589
4        4   2.051173  -0.585662
5        5  0.429526  -0.606998
6        6  0.106223  -1.525680
7        7  0.795026  -0.374438
8        8  0.134048   1.202055
9        9  0.284748   0.262467
```
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 [540]: reader = pd.read_stata('stata.dta', chunksize=3)
```
```python
In [541]: 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().

```python
In [542]: reader = pd.read_stata('stata.dta', iterator=True)
In [543]: chunk1 = reader.read(5)
In [544]: chunk2 = reader.read(5)
```

Currently the index is retrieved as a column.

The parameter convert_categories 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.

### 24.13.2.1 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_categories (True by default). The keyword argument order_categories (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.

### 24.14 SAS Formats

New in version 0.17.0.

The top-level function `read_sas()` can read (but not write) SAS xport (.XPT) and SAS7BDAT (.sas7bdat) format files were added in v0.18.0.

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
rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
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.

### 24.15 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.
24.15.1 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.

24.16 Performance Considerations

This is an informal comparison of various IO methods, using pandas 0.20.3. 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

Writing

In [14]: %timeit test_sql_write(df)
2.37 s ± 36.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [15]: %timeit test_hdf_fixed_write(df)
194 ms ± 65.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [26]: %timeit test_hdf_fixed_write_compress(df)
119 ms ± 2.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [16]: %timeit test_hdf_table_write(df)
623 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [27]: %timeit test_hdf_table_write_compress(df)
563 ms ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [17]: %timeit test_csv_write(df)
3.13 s ± 49.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [30]: %timeit test_feather_write(df)
103 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [31]: %timeit test_pickle_write(df)
109 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [32]: %timeit test_pickle_write_compress(df)
3.33 s ± 55.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Reading

In [18]: %timeit test_sql_read()
1.35 s ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```
And here’s the code

```python
import os
import pandas as pd
import sqlite3
from numpy.random import randn
from pandas.io import sql

sz = 1000000

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)
    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')
CHAPTER TWENTYFIVE

REMOTE DATA ACCESS

25.1 DataReader

The sub-package pandas.io.data was deprecated in v.0.17 and removed in v.0.19. Instead there has been created a separately installable pandas-datareader package. This will allow the data modules to be independently updated on your pandas installation.

For code older than < 0.19 you should replace the imports of the following:

```python
from pandas.io import data, wb
```

With:

```python
from pandas_datareader import data, wb
```
26.1 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 sizeable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in python, for example 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.

26.1.1 Pure python

We have a DataFrame to which we want to apply a function row-wise.

```python
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]:
     N     a       b     x
0  585  0.469112 -0.218470 x
1  841 -0.282863 -0.061645 x
2  251 -1.509059 -0.723780 x
3  972 -1.135632  0.551225 x
4  458 -0.173215  0.837519 x
5  159  0.119209  1.103245 x
993 190  0.131892  0.290162 x
994 931  0.342097  0.215341 x
995 374 -1.512743  0.874737 x
996 246  0.933753  1.120790 x
997 157 -0.308013  0.198768 x
998 977 -0.079915  1.757555 x
999 770 -1.010589 -1.115680 x
[1000 rows x 4 columns]
```
Here's the function in pure python:

```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)
   ...:     return s * dx
   ...
```

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:

```python
In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)

671001 function calls (665992 primitive calls) in 0.256 seconds

Ordered by: internal time
List reduced from 145 to 4 due to restriction <4>
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1000    0.136    0.000   0.200    0.000 <ipython-input-4-91e33489f136>:1(integrate_f)
552423    0.064    0.000   0.064    0.000 <ipython-input-3-bc41a25943f6>:1(f)
3000    0.006    0.000   0.038    0.000 base.py:2534(get_value)
3000    0.004    0.000   0.044    0.000 series.py:620(__getitem__)
```

By far the majority of time is spend inside either `integrate_f` or `f`, hence we’ll concentrate our efforts cythonizing these two functions.

**Note:** In python 2 replacing the `range` with its generator counterpart (`xrange`) would mean the `range` line would vanish. In python 3 `range` is already a generator.

### 26.1.2 Plain cython

First we’re going to need to import the cython magic function to ipython (for cython versions < 0.21 you can use `%load_ext cythonmagic`):

```python
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.

In [4]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 85.5 ms per loop

Already this has shaved a third off, not too bad for a simple copy and paste.

26.1.3 Adding type

We get another huge improvement simply by providing type information:

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

In [4]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 20.3 ms per loop

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:

In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
118576 function calls (113567 primitive calls) in 0.053 seconds

Ordered by: internal time
List reduced from 142 to 4 due to restriction <4>

ncalls  ttotal  percall  cumtime  percall filename:lineno(function)
3000  0.006  0.000  0.035  0.000 base.py:2534(get_value)
3000  0.004  0.000  0.040  0.000 series.py:620(__getitem__)
9027  0.003  0.000  0.007  0.000 {built-in method builtins.getattr}
1  0.003  0.003  0.052  0.052 {pandas._libs.lib.reduce}
26.1.4 Using ndarray

It’s calling series... a lot! It’s creating a Series from each row, and getting from both the index and the series (three times for each row). Function calls are expensive in python, so maybe we could minimise these by cythonizing the apply part.

Note: We are now passing ndarrays into the cython function, fortunately cython plays very nicely with numpy.

```
In [10]: %%cython
    ....: cimport numpy as np
    ....: import numpy as np
    ....: cdef double f_typed(double x) except? -2:
    ....:     return x * (x - 1)
    ....: cpdef 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
    ....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_b, np.ndarray col_N):
    ....:     assert (col_a.dtype == np.float and col_b.dtype == np.float and col_N.dtype == np.int)
    ....:     cdef Py_ssize_t 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(len(col_a)):
    ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
    ....:     return res
```

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 .values attribute of the Series. 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 .values to get the underlying ndarray
```
apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
```

**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'].values, df['b'].values, df['N'].values)
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:

```python
In [11]: %prun -l 4 apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
203 function calls in 0.001 seconds

Ordered by: internal time
List reduced from 53 to 4 due to restriction <4>

 ncalls tottime percall cumtime percall filename:lineno(function)
      1    0.001  0.001    0.001  0.001 {built-in method _cython_magic_
   ->9df59845e7ff190f442cda6f3d60e56c.apply_integrate_f}
      3    0.000    0.000    0.000    0.000 internals.py:3860(iget)
      1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
      3    0.000    0.000    0.000    0.000 generic.py:1837(_get_item_cache)
```

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.

### 26.1.5 More advanced techniques

There is still hope for improvement. Here’s an example of using some more advanced cython techniques:

```python
In [12]: %%cython
   ....: cimport cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef 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 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'].values, df['b'].values, df['N'].values)
1000 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.
26.2 Using numba

A recent alternative to statically compiling cython code, is to use a *dynamic jit-compiler*, numba. Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). 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:** You will need to install numba. This is easy with conda, by using: conda install numba, see *installing using miniconda.*

**Note:** As of numba version 0.20, pandas objects cannot be passed directly to numba-compiled functions. Instead, one must pass the numpy array underlying the pandas object to the numba-compiled function as demonstrated below.

26.2.1 Jit

Using numba to just-in-time compile your code. We simply take the plain python code from above and annotate with the @jit decorator.

```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'].values, df['b'].values, df['N'].values)
    return pd.Series(result, index=df.index, name='result')
```

Note that we directly pass numpy arrays to the numba function. compute_numba is just a wrapper that provides a nicer interface by passing/returning pandas objects.
26.2.2 Vectorize

`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 toy example of doubling each observation:

```python
import numba
def double_every_value_nonumba(x):
    return x*2
@numba.vectorize
def double_every_value_withnumba(x):
    return x*2

# Custom function without numba
In [5]: %timeit df['col1_doubled'] = df.a.apply(double_every_value_nonumba)
1000 loops, best of 3: 797 us per loop

# Standard implementation (faster than a custom function)
In [6]: %timeit df['col1_doubled'] = df.a*2
1000 loops, best of 3: 233 us per loop

# Custom function with numba
In [7]: %timeit df['col1_doubled'] = double_every_value_withnumba(df.a.values)
1000 loops, best of 3: 145 us per loop
```

26.2.3 Caveats

**Note:** `numba` will execute on any function, but can only accelerate certain classes of functions.

`numba` is best at accelerating functions that apply numerical functions to numpy arrays. When passed a function that only uses operations it knows how to accelerate, it will execute in `nopython` mode.

If `numba` is passed a function that includes something it doesn’t know how to work with – a category that currently includes sets, lists, dictionaries, or string functions – it will revert to `object` mode. In `object` mode, `numba` will execute but your code will not speed up significantly. 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. `@numba.jit(nopython=True)`). For more on troubleshooting `numba` modes, see the `numba` troubleshooting page.

Read more in the `numba` docs.

26.3 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()`.

### 26.3.1 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` and `arctan2`.

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.

26.3.2 `eval()` Examples

`pandas.eval()` works well with expressions containing large arrays.

First let’s create a few decent-sized arrays to play with:

```python
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()`:

```python
In [15]: %timeit df1 + df2 + df3 + df4
   11.9 ms +- 379 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [16]: %timeit pd.eval('df1 + df2 + df3 + df4')
   8.29 ms +- 156 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Now let’s do the same thing but with comparisons:

```python
In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
   25.9 ms +- 654 us per loop (mean +- std. dev. of 7 runs, 10 loops each)

In [18]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
   10.6 ms +- 336 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

eval() also works with unaligned pandas objects:

```python
In [19]: s = pd.Series(np.random.randn(50))
In [20]: %timeit df1 + df2 + df3 + df4 + s
   22.1 ms +- 583 us per loop (mean +- std. dev. of 7 runs, 10 loops each)

In [21]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
   9.49 ms +- 411 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.
26.3.3 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.

```python
In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=['a', 'b'])
In [23]: df.eval('a + b')
Out[23]:
     0    1
0 -0.246747  0.867786
1  0.867786 -1.626063
2 -1.626063 -1.134978
3 -1.134978  1.027798
4  1.027798 -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.

New in version 0.18.0.

The inplace keyword determines whether this assignment will performed on the original DataFrame or return a copy with the new column.

```
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
```

```
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  1  5  5 10
1  1  6  7 14
2  1  7  9 18
3  1  8 11 22
4  1  9 13 26
```

When inplace is set to False, a copy of the DataFrame with the new or modified columns is returned and the original frame is unchanged.

```
In [29]: df
Out[29]:
   a  b  c  d
0  1  5  5 10
1  1  6  7 14
```
In 

\[30\]: df.eval('e = a - c', inplace=False)

\[30\] Out:

→

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>14</td>
<td>-6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>18</td>
<td>-8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>11</td>
<td>22</td>
<td>-10</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>13</td>
<td>26</td>
<td>-12</td>
</tr>
</tbody>
</table>

In 

\[31\]: df

\[31\] Out:

→

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>13</td>
<td>26</td>
</tr>
</tbody>
</table>

New in version 0.18.0.

As a convenience, multiple assignments can be performed by using a multi-line string.

In 

\[32\]: df.eval(""
......: c = a + b
......: d = a + b + c
......: a = 1""", inplace=False)

\[32\] Out:

→

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

The equivalent in standard Python would be

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

\[37\] Out:

→

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>11</td>
<td>22</td>
</tr>
</tbody>
</table>
New in version 0.18.0.

The `query` method gained the `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
0  3  3
1  4  4

In [40]: df.query('a > 2', inplace=True)
In [41]: df
Out[41]:
   a  b
0  3  3
1  4  4
```

**Warning:** Unlike with `eval`, the default value for `inplace` for `query` is `False`. This is consistent with prior versions of pandas.

### 26.3.4 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')
   →
   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.
In [46]: a = np.random.randn()

In [47]: df.query('@a < a')
Out[47]:
   a   b
0  0.863987 -0.115998

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,

In [49]: a, b = 1, 2

In [50]: pd.eval('@a + b')
Traceback (most recent call last):
  File "/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/IPython/core/interactiveshell.py", line 2862, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-50-d778d4ee0271>"", line 1, in <module>
    pd.eval('@a + b')
  File "/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/pandas/core/computation/eval.py", line 282, in eval
    _check_for_locals(expr, level, parser)
  File "/Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/pandas/core/computation/eval.py", line 149, in _check_for_locals
    raise SyntaxError(msg)
File "<string>", line unknown
SyntaxError: The '@' prefix is not allowed in top-level eval calls, please refer to your variables by name without the '@' prefix

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

## 26.3.5 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 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)'
The same expression can be “anded” together with the word `and` as well:

```python
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` and `or` operators here have the same precedence that they would in vanilla Python.

### 26.3.6 `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
   12.4 ms +- 827 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
In [63]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
   13.5 ms +- 1.17 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

### 26.3.7 `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.
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 numpy.random.randn().

### 26.3.8 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 ndarray.astype(str) will truncate any strings that are more than 60 characters in length. Second, we can’t pass object arrays to numexpr thus string comparisons must be evaluated in Python space.

The upshot is that this only applies to object-dtype’d expressions. So, if you have an expression—for example
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.
Note: The SparsePanel class has been removed in 0.19.0

We have implemented “sparse” versions of Series and DataFrame. These are not 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) is omitted. A special SparseIndex object tracks where data has been “sparsified”. This will make much more sense in an example. All of the standard pandas data structures have a to_sparse method:

```python
In [1]: ts = pd.Series(randn(10))
In [2]: ts[2:-2] = np.nan
In [3]: sts = ts.to_sparse()
In [4]: sts
Out[4]:
   0   0.469112
   1  -0.282863
   2   NaN
   3   NaN
   4   NaN
   5   NaN
   6   NaN
   7   NaN
   8  -0.861849
   9  -2.104569
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The to_sparse method takes a kind argument (for the sparse index, see below) and a fill_value. So if we had a mostly zero Series, we could convert it to sparse with fill_value=0:

```python
In [5]: ts.fillna(0).to_sparse(fill_value=0)
Out[5]:
   0   0.469112
   1  -0.282863
   2  0.000000
   3  0.000000
   4  0.000000
   5  0.000000
```
The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA DataFrame:

```
In [6]: df = pd.DataFrame(randn(10000, 4))
In [7]: df.iloc[:9998] = np.nan
In [8]: sdf = df.to_sparse()
```

```
In [9]: sdf
Out[9]:
     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
5   NaN NaN NaN NaN
6   NaN NaN NaN NaN
... ... ... ... ...
9993  NaN  NaN  NaN  NaN
9994  NaN  NaN  NaN  NaN
9995  NaN  NaN  NaN  NaN
9996  NaN  NaN  NaN  NaN
9997  NaN  NaN  NaN  NaN
9998  0.509184 -0.774928 -1.369894 -0.382141
9999  0.280249 -1.648493 1.490865 -0.890819
```

[10000 rows x 4 columns]

```
In [10]: sdf.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. Functionally, their behavior should be nearly identical to their dense counterparts.

Any sparse object can be converted back to the standard dense form by calling `to_dense`:

```
In [11]: sts.to_dense()
Out[11]:
     0   1   2   3
0   0.469112 NaN NaN NaN
1  -0.282863 NaN NaN NaN
2   NaN NaN NaN NaN
3   NaN NaN NaN NaN
4   NaN NaN NaN NaN
5   NaN NaN NaN NaN
6   NaN NaN NaN NaN
7   NaN NaN NaN NaN
```

Chapter 27. Sparse data structures
27.1 SparseArray

SparseArray is the base layer for all of the sparse indexed data structures. It is a 1-dimensional ndarray-like object storing only values distinct from the fill_value:

```
In [12]: arr = np.random.randn(10)
In [14]: sparr = pd.SparseArray(arr)
In [15]: sparr
Out[15]:
[-1.9556635297215477, -1.6588664275960427, nan, nan, nan, 1.1589328886422277, 0.
   → 14529711373305043, nan, 0.6060271905134522, 1.3342113401317768]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

Like the indexed objects (SparseSeries, SparseDataFrame), a SparseArray can be converted back to a regular ndarray by calling to_dense:

```
In [16]: sparr.to_dense()
Out[16]:
array([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
       nan, 0.606 , 1.3342])
```

27.2 SparseList

The SparseList class has been deprecated and will be removed in a future version. See the docs of a previous version for documentation on SparseList.

27.3 SparseIndex objects

Two kinds of SparseIndex are implemented, block and integer. We recommend using block as it’s more memory efficient. The integer format keeps an arrays of all of the locations where the data are not equal to the fill value. The block format tracks only the locations and sizes of blocks of data.

27.4 Sparse Dtypes

Sparse data should have the same dtype as its dense representation. Currently, float64, int64 and bool dtypes are supported. Depending on the original dtype, fill_value default changes:

- float64: np.nan
In [17]: s = pd.Series([1, np.nan, np.nan])

In [18]: s
Out[18]:
      0  1.0
      1  NaN
      2  NaN
dtype: float64

In [19]: s.to_sparse()
Out[19]:
      0  1.0
      1  NaN
      2  NaN
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [20]: s = pd.Series([1, 0, 0])

In [21]: s
Out[21]:
     0  1
     1  0
     2  0
dtype: int64

In [22]: s.to_sparse()
Out[22]:
     0  1
     1  0
     2  0
dtype: int64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [23]: s = pd.Series([True, False, True])

In [24]: s
Out[24]:
     0   True
     1   False
     2   True
dtype: bool

In [25]: s.to_sparse()
Out[25]:
     0   True
     1   False
     2   True
dtype: bool
BlockIndex
Block locations: array([0, 2], dtype=int32)
You can change the dtype using `.astype()`, the result is also sparse. Note that `.astype()` also affects to the `fill_value` to keep its dense representation.

```python
In [26]: s = pd.Series([1, 0, 0, 0, 0])

In [27]: s
Out[27]:
0    1
1    0
2    0
3    0
4    0
dtype: int64

In [28]: ss = s.to_sparse()

In [29]: ss
Out[29]:
0    1
1    0
2    0
3    0
4    0
dtype: int64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [30]: ss.astype(np.float64)
Out[30]:
0    1.0
1    0.0
2    0.0
3    0.0
4    0.0
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)
```

It raises if any value cannot be coerced to specified dtype.

```python
In [1]: ss = pd.Series([1, np.nan, np.nan]).to_sparse()

In [2]: ss.astype(np.int64)
ValueError: unable to coerce current fill_value nan to int64 dtype
```
27.5 Sparse Calculation

You can apply NumPy ufuncs to SparseArray and get a SparseArray as a result.

```
In [31]: arr = pd.SparseArray([1., np.nan, np.nan, -2., np.nan])
In [32]: np.abs(arr)
Out[32]:
[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 [33]: arr = pd.SparseArray([1., -1, -1, -2., -1], fill_value=-1)
In [34]: np.abs(arr)
Out[34]:
[1.0, 1.0, 1.0, 2.0, 1.0]
Fill: 1
IntIndex
Indices: array([0, 3], dtype=int32)
```

```
In [35]: np.abs(arr).to_dense()
Out[35]:
array([1., 1., 1., 2., 1.])
```

27.6 Interaction with scipy.sparse

27.6.1 SparseDataFrame

New in version 0.20.0.

Pandas supports creating sparse dataframes directly from scipy.sparse matrices.

```
In [36]: from scipy.sparse import csr_matrix
In [37]: arr = np.random.random(size=(1000, 5))
In [38]: arr[arr < .9] = 0
In [39]: sp_arr = csr_matrix(arr)
In [40]: sp_arr
Out[40]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
 with 517 stored elements in Compressed Sparse Row format>
In [41]: sdf = pd.SparseDataFrame(sp_arr)
In [42]: sdf
Out[42]:
   0   1    2   3   4
0  0.956380 NaN  NaN  NaN  NaN
```
All sparse formats are supported, but matrices that are not in COOrdinate format will be converted, copying data as needed. To convert a SparseDataFrame back to sparse SciPy matrix in COO format, you can use the SparseDataFrame.to_coo() method:

```python
In [43]: sdf.to_coo()
```

```python
Out[43]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
      with 517 stored elements in COOrdinate format>
```

### 27.6.2 SparseSeries

A SparseSeries.to_coo() method is implemented for transforming a SparseSeries indexed by a MultiIndex to a scipy.sparse.coo_matrix.

The method requires a MultiIndex with two or more levels.

```python
In [44]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [45]: 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 [46]: s
```

```python
Out[46]:
A  B  C  D
1  2  a  0  3.0
   1   NaN
1  1  b  0  1.0
   1   3.0
2  1  b  0  NaN
   1   NaN
dtype: float64
```

# SparseSeries
In [47]: ss = s.to_sparse()

In [48]: ss
Out[48]:
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
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 2], dtype=int32)

In the example below, we transform the SparseSeries 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 [49]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
....: column_levels=['C', 'D'],
....: sort_labels=True)
....:

In [50]: A
Out[50]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [51]: A.todense()
→ matrix([[0., 0., 1., 3.],
          [3., 0., 0., 0.],
          [0., 0., 0., 0.]])

In [52]: rows
→ [(1, 1), (1, 2), (2, 1)]

In [53]: columns
→ [('a', 0), ('a', 1), ('b', 0), ('b', 1)]

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

In [54]: A, rows, columns = ss.to_coo(row_levels=['A', 'B', 'C'],
....: column_levels=['D'],
....: sort_labels=False)
....:

In [55]: A
Out[55]:
<3x2 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [56]: A.todense()
A convenience method `SparseSeries.from_coo()` is implemented for creating a `SparseSeries` from a scipy.sparse.coo_matrix.

```python
In [59]: from scipy import sparse

In [60]: A = sparse.coo_matrix(((3.0, 1.0, 2.0), ([1, 0, 0], [0, 2, 3])),
        shape=(3, 4))

In [61]: A
Out[61]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [62]: A.todense()
matrix([[0. , 0. , 1. , 2. ],
        [3.0, 0. , 0. , 0. ],
        [0. , 0. , 0. , 0. ]])
```

The default behaviour (with `dense_index=False`) simply returns a `SparseSeries` containing only the non-null entries.

```python
In [63]: ss = pd.SparseSeries.from_coo(A)

In [64]: ss
Out[64]:
0   3 2.0
     1 0
1   0 3.0
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

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.

```python
In [65]: ss_dense = pd.SparseSeries.from_coo(A, dense_index=True)

In [66]: ss_dense
Out[66]:
```

27.6. Interaction with scipy.sparse
<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>NaN</td>
<td>NaN</td>
<td>1.0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

```
dtype: float64
BlockIndex
Block locations: array([2], dtype=int32)
Block lengths: array([3], dtype=int32)
```
28.1 DataFrame memory usage

The memory usage of a dataframe (including the index) is shown when accessing the `info` method of a dataframe. A configuration option, `display.memory_usage` (see Options and Settings), 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 `df.info()`:

```
In [1]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
   ...:   'complex128', 'object', 'bool']
   ...
In [2]: n = 5000
In [3]: data = dict((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):
bool     5000 non-null bool
complex128 5000 non-null complex128
datetime64[ns] 5000 non-null datetime64[ns]
float64   5000 non-null float64
int64     5000 non-null int64
object    5000 non-null object
timedelta64[ns] 5000 non-null timedelta64[ns]
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: 289.1+ 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`.

New in version 0.17.1.
Passing `memory_usage='deep'` will enable a more accurate memory usage report, that accounts for the full usage of the contained objects. This is optional as it can be expensive to do this deeper introspection.

```python
In [7]: df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
bool 5000 non-null bool
complex128 5000 non-null complex128
datetime64[ns] 5000 non-null datetime64[ns]
float64 5000 non-null float64
int64 5000 non-null int64
object 5000 non-null object
timedelta64[ns] 5000 non-null timedelta64[ns]
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: 425.6 KB
```

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 of the dataframe can be found with the `memory_usage` method:

```python
In [8]: df.memory_usage()
Out[8]:
Index    80
bool     5000
complex128     80000
datetime64[ns]  40000
float64     40000
int64    40000
object   40000
timedelta64[ns]  40000
categorical 10920
dtype: int64
```

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]:
bool     5000
complex128     80000
datetime64[ns]  40000
float64     40000
int64    40000
object   40000
timedelta64[ns]  40000
categorical 10920
```

# total memory usage of dataframe
```
In [9]: df.memory_usage().sum()
```

```
296000
```

1164 Chapter 28. Frequently Asked Questions (FAQ)
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`.

### 28.2 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 a `if` or when using the boolean operations, `and`, or, or `not`. It is not clear what the result of

```python
>>> if pd.Series([False, True, False]):
... print("I was true")
```

should be. Should it be `True` because it’s not zero-length? `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().
```

If you see that, you need to explicitly choose what you want to do with it (e.g., use `any()`, `all()` or `empty`). or, 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
```

or return if any value is `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
```
28.2.1 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.

28.2.2 Using the `in` operator

Using the Python `in` operator on a Series tests for membership in the index, not membership among the values.

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()`:

For DataFrames, likewise, `in` applies to the column axis, testing for membership in the list of column names.

28.3 NaN, Integer NA values and NA type promotions

28.3.1 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.

28.3.2 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:

```python
In [15]: s = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
In [16]: s
Out[16]:
a    1
b    2
c    3
```
This trade-off is made largely for memory and performance reasons, and also so that the resulting Series continues to be “numeric”. One possibility is to use dtype=object arrays instead.

### 28.3.3 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. These are summarized by 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.

### 28.3.4 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.

### 28.4 Differences with NumPy

For Series and DataFrame objects, var normalizes by N−1 to produce unbiased estimates of the sample variance, while NumPy’s var normalizes by N, which measures the variance of the sample. Note that cov normalizes by N−1 in both pandas and NumPy.

### 28.5 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the DataFrame.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.

### 28.6 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/DataFrame/Panel constructors using something similar to the following:

```
In [21]: x = np.array(list(range(10)), 'i4')  # big endian
In [22]: newx = x.byteswap().newbyteorder()  # force native byteorder
In [23]: s = pd.Series(newx)
```

See the NumPy documentation on byte order for more details.
**Warning:** Up to pandas 0.19, a pandas.rpy module existed with functionality to convert between pandas and rpy2 objects. This functionality now lives in the rpy2 project itself. See the updating section of the previous documentation for a guide to port your code from the removed pandas.rpy to rpy2 functions.

rpy2 is an interface to R running embedded in a Python process, and also includes functionality to deal with pandas DataFrames. Converting data frames back and forth between rpy2 and pandas should be largely automated (no need to convert explicitly, it will be done on the fly in most rpy2 functions). To convert explicitly, the functions are pandas2ri.py2ri() and pandas2ri.ri2py().

See also the documentation of the rpy2 project: https://rpy2.readthedocs.io.

In the remainder of this page, a few examples of explicit conversion is given. The pandas conversion of rpy2 needs first to be activated:

```python
In [1]: from rpy2.robjects import r, pandas2ri
In [2]: pandas2ri.activate()
```

### 29.1 Transferring R data sets into Python

Once the pandas conversion is activated (pandas2ri.activate()), many conversions of R to pandas objects will be done automatically. For example, to obtain the ‘iris’ dataset as a pandas DataFrame:

```python
In [3]: r.data('iris')
Out[3]:
R object with classes: ('character',) mapped to:
<StrVector - Python:0x132e375c8 / R:0x7ff8b921f338>
['iris']

In [4]: r['iris'].head()
```

```
  Sepal.Length  Sepal.Width  Petal.Length  Petal.Width  Species
   5.1          3.5          1.4          0.2         setosa
   4.9          3.0          1.4          0.2         setosa
   4.7          3.2          1.3          0.2         setosa
   4.6          3.1          1.5          0.2         setosa
   5.0          3.6          1.4          0.2         setosa
```
If the pandas conversion was not activated, the above could also be accomplished by explicitly converting it with the `pandas2ri.ri2py` function.

### 29.2 Converting DataFrames into R objects

The `pandas2ri.py2ri` function support the reverse operation to convert DataFrames into the equivalent R object (that is, `data.frame`):

```python
In [5]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]},
                          index=['one', 'two', 'three'])

In [6]: r_dataframe = pandas2ri.py2ri(df)

In [7]: print(type(r_dataframe))
<type 'rpy2.robjects.vectors.DataFrame'>

In [8]: print(r_dataframe)
   A  B  C
one 1  4  7
two 2  5  8
three 3  6  9
```

The DataFrame’s index is stored as the `rownames` attribute of the `data.frame` instance.
Increasingly, packages are being built on top of pandas to address specific needs in data preparation, analysis and visualization. This is encouraging because it means pandas is not only helping users to handle their data tasks but also that it provides a better starting point for developers to build powerful and more focused data tools. The creation of libraries that complement pandas’ functionality also allows pandas development to remain focused around it’s original requirements.

This is an in-exhaustive list of projects that build on pandas in order to provide tools in the PyData space.

We’d like to make it easier for users to find these project, if you know of other substantial projects that you feel should be on this list, please let us know.

### 30.1 Statistics and Machine Learning

#### 30.1.1 Statsmodels

Statsmodels is the prominent python “statistics and econometrics library” and it has a long-standing special relationship with pandas. Statsmodels provides powerful statistics, econometrics, analysis and modeling functionality that is out of pandas’ scope. Statsmodels leverages pandas objects as the underlying data container for computation.

#### 30.1.2 sklearn-pandas

Use pandas DataFrames in your scikit-learn ML pipeline.

### 30.2 Visualization

#### 30.2.1 Bokeh

Bokeh is a Python interactive visualization library for large datasets that natively uses the latest web technologies. Its goal is to provide elegant, concise construction of novel graphics in the style of Protovis/D3, while delivering high-performance interactivity over large data to thin clients.

#### 30.2.2 yhat/ggplot

Hadley Wickham’s ggplot2 is a foundational exploratory visualization package for the R language. Based on “The Grammar of Graphics” it provides a powerful, declarative and extremely general way to generate bespoke plots of any kind of data. It’s really quite incredible. Various implementations to other languages are available, but a faithful
implementation for python users has long been missing. Although still young (as of Jan-2014), the yhat/ggplot project has been progressing quickly in that direction.

### 30.2.3 Seaborn

Although pandas has quite a bit of “just plot it” functionality built-in, visualization and in particular statistical graphics is a vast field with a long tradition and lots of ground to cover. The Seaborn project builds on top of pandas and matplotlib to provide easy plotting of data which extends to more advanced types of plots then those offered by pandas.

### 30.2.4 Vincent

The Vincent project leverages Vega (that in turn, leverages d3) to create plots. Although functional, as of Summer 2016 the Vincent project has not been updated in over two years and is unlikely to receive further updates.

### 30.2.5 IPython Vega

Like Vincent, the IPython Vega project leverages Vega to create plots, but primarily targets the IPython Notebook environment.

### 30.2.6 Plotly

Plotly’s Python API enables interactive figures and web shareability. Maps, 2D, 3D, and live-streaming graphs are rendered with WebGL and D3.js. The library supports plotting directly from a pandas DataFrame and cloud-based collaboration. Users of matplotlib, ggplot for Python, and Seaborn can convert figures into interactive web-based plots. Plots can be drawn in IPython Notebooks, edited with R or MATLAB, modified in a GUI, or embedded in apps and dashboards. Plotly is free for unlimited sharing, and has cloud, offline, or on-premise accounts for private use.

### 30.2.7 QtPandas

Spun off from the main pandas library, the qtpandas library enables DataFrame visualization and manipulation in PyQt4 and PySide applications.

### 30.3 IDE

#### 30.3.1 IPython

IPython is an interactive command shell and distributed computing environment. IPython Notebook is a web application for creating IPython notebooks. An IPython notebook is a JSON document containing an ordered list of input/output cells which can contain code, text, mathematics, plots and rich media. IPython notebooks can be converted to a number of open standard output formats (HTML, HTML presentation slides, LaTeX, PDF, ReStructuredText, Markdown, Python) through ‘Download As’ in the web interface and ipython nbconvert in a shell.

Pandas DataFrames implement _repr_html_ methods which are utilized by IPython Notebook for displaying (abbreviated) HTML tables. (Note: HTML tables may or may not be compatible with non-HTML IPython output formats.)
30.3.2 quantopian/qgrid

qgrid is “an interactive grid for sorting and filtering DataFrames in IPython Notebook” built with SlickGrid.

30.3.3 Spyder

Spyder is a cross-platform Qt-based open-source Python IDE with editing, testing, debugging, and introspection features. Spyder can now introspect and display Pandas DataFrames and show both “column wise min/max and global min/max coloring.”

30.4 API

30.4.1 pandas-datareader

pandas-datareader is a remote data access library for pandas. pandas.io from pandas < 0.17.0 is now refactored/split-off to and importable from pandas_datareader (PyPI:pandas-datareader). Many/most of the supported APIs have at least a documentation paragraph in the pandas-datareader docs:

The following data feeds are available:

- Yahoo! Finance
- Google Finance
- FRED
- Fama/French
- World Bank
- OECD
- Eurostat
- EDGAR Index

30.4.2 quandl/Python

Quandl API for Python wraps the Quandl REST API to return Pandas DataFrames with timeseries indexes.

30.4.3 pydatastream

PyDatastream is a Python interface to the Thomson Dataworks Enterprise (DWE/Datastream) SOAP API to return indexed Pandas DataFrames or Panels with financial data. This package requires valid credentials for this API (non free).

30.4.4 pandaSDMX

pandaSDMX is a library to retrieve and acquire statistical data and metadata disseminated in SDMX 2.1, an ISO-standard widely used by institutions such as statistics offices, central banks, and international organisations. pandaSDMX can expose datasets and related structural metadata including dataflows, code-lists, and datastructure definitions as pandas Series or multi-indexed DataFrames.
30.4.5 fredapi

fredapi is a Python interface to the Federal Reserve Economic Data (FRED) provided by the Federal Reserve Bank of St. Louis. It works with both the FRED database and ALFRED database that contains point-in-time data (i.e. historic data revisions). fredapi provides a wrapper in python to the FRED HTTP API, and also provides several convenient methods for parsing and analyzing point-in-time data from ALFRED. fredapi makes use of pandas and returns data in a Series or DataFrame. This module requires a FRED API key that you can obtain for free on the FRED website.

30.5 Domain Specific

30.5.1 Geopandas

Geopandas extends pandas data objects to include geographic information which support geometric operations. If your work entails maps and geographical coordinates, and you love pandas, you should take a close look at Geopandas.

30.5.2 xarray

xarray brings the labeled data power of pandas to the physical sciences by providing N-dimensional variants of the core pandas data structures. It aims to provide a pandas-like and pandas-compatible toolkit for analytics on multi-dimensional arrays, rather than the tabular data for which pandas excels.

30.6 Out-of-core

30.6.1 Dask

Dask is a flexible parallel computing library for analytics. Dask allow a familiar DataFrame interface to out-of-core, parallel and distributed computing.

30.6.2 Blaze

Blaze provides a standard API for doing computations with various in-memory and on-disk backends: NumPy, Pandas, SQLAlchemy, MongoDB, PyTables, PySpark.

30.6.3 Odo

Odo provides a uniform API for moving data between different formats. It uses pandas own read_csv for CSV IO and leverages many existing packages such as PyTables, h5py, and pymongo to move data between non pandas formats. Its graph based approach is also extensible by end users for custom formats that may be too specific for the core of odo.

30.7 Data validation

30.7.1 Engarde

Engarde is a lightweight library used to explicitly state your assumptions about your datasets and check that they’re actually true.
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.

### 31.1 Quick Reference

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

#### 31.1.1 Querying, Filtering, Sampling

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

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

### 31.1.3 Transforming

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

### 31.1.4 Grouping and Summarizing

<table>
<thead>
<tr>
<th><strong>R</strong></th>
<th><strong>pandas</strong></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>

### 31.2 Base R

#### 31.2.1 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

```R
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list('abc'))
In [2]: df[['a', 'c']]  
Out[2]:
a    c
0  1.039575 -0.424972
1  0.567020 -1.087401
2 -0.673690 -1.478427
3  0.524988  0.577046
4 -1.715002 -0.370647
5 -1.157892  0.844885
```
In [3]: df.loc[:, ['a', 'c']]

→

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1.039575</td>
<td>-0.424972</td>
</tr>
<tr>
<td>1</td>
<td>0.567020</td>
<td>-1.087401</td>
</tr>
<tr>
<td>2</td>
<td>-0.673690</td>
<td>-1.478427</td>
</tr>
<tr>
<td>3</td>
<td>0.524988</td>
<td>0.577046</td>
</tr>
<tr>
<td>4</td>
<td>-1.715002</td>
<td>-0.370647</td>
</tr>
<tr>
<td>5</td>
<td>-1.157892</td>
<td>0.848885</td>
</tr>
<tr>
<td>6</td>
<td>1.075770</td>
<td>1.643563</td>
</tr>
<tr>
<td>7</td>
<td>-1.469388</td>
<td>-0.674600</td>
</tr>
<tr>
<td>8</td>
<td>-1.776904</td>
<td>-1.294524</td>
</tr>
<tr>
<td>9</td>
<td>0.413738</td>
<td>-0.472035</td>
</tr>
</tbody>
</table>

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_[:10, 24:30]]

Out[8]:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.013960</td>
<td>-0.362543</td>
<td>-0.006154</td>
<td>-0.923061</td>
<td>0.895717</td>
<td>0.805244</td>
</tr>
<tr>
<td>1</td>
<td>0.545952</td>
<td>-1.219217</td>
<td>-1.226825</td>
<td>0.769804</td>
<td>-1.281247</td>
<td>-0.727707</td>
</tr>
<tr>
<td>2</td>
<td>2.396780</td>
<td>0.014871</td>
<td>3.357427</td>
<td>-0.317441</td>
<td>-1.236269</td>
<td>0.896171</td>
</tr>
<tr>
<td>3</td>
<td>-0.988387</td>
<td>0.094055</td>
<td>1.262731</td>
<td>1.289997</td>
<td>0.082423</td>
<td>-0.055758</td>
</tr>
<tr>
<td>4</td>
<td>-1.340896</td>
<td>1.846883</td>
<td>-1.328865</td>
<td>1.682706</td>
<td>-1.717693</td>
<td>0.888782</td>
</tr>
<tr>
<td>5</td>
<td>0.464000</td>
<td>0.227371</td>
<td>-0.496222</td>
<td>0.306389</td>
<td>-2.290613</td>
<td>-1.134623</td>
</tr>
<tr>
<td>6</td>
<td>-0.507516</td>
<td>-0.230096</td>
<td>0.394500</td>
<td>-1.934370</td>
<td>-1.652499</td>
<td>1.488753</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>-0.083272</td>
<td>-0.273955</td>
<td>-0.772369</td>
<td>-1.242807</td>
<td>-0.386336</td>
<td>-0.182486</td>
</tr>
<tr>
<td>24</td>
<td>2.071413</td>
<td>-1.364763</td>
<td>1.122066</td>
<td>0.066847</td>
<td>1.751987</td>
<td>0.419071</td>
</tr>
<tr>
<td>25</td>
<td>0.036609</td>
<td>0.359986</td>
<td>1.211905</td>
<td>0.850427</td>
<td>1.554957</td>
<td>-0.88463</td>
</tr>
<tr>
<td>26</td>
<td>-1.179240</td>
<td>0.238923</td>
<td>1.756671</td>
<td>-0.747571</td>
<td>0.543625</td>
<td>-0.159609</td>
</tr>
<tr>
<td>27</td>
<td>0.025645</td>
<td>0.932436</td>
<td>-1.694531</td>
<td>-0.182236</td>
<td>-1.072710</td>
<td>0.466764</td>
</tr>
<tr>
<td>28</td>
<td>0.439086</td>
<td>0.812684</td>
<td>-0.128932</td>
<td>-0.142506</td>
<td>-1.137207</td>
<td>0.462001</td>
</tr>
<tr>
<td>29</td>
<td>-0.909806</td>
<td>-0.312066</td>
<td>0.383630</td>
<td>-0.631606</td>
<td>1.321415</td>
<td>-0.004799</td>
</tr>
<tr>
<td>30</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>2.565646</td>
<td>1.431256</td>
<td>1.340309</td>
<td>0.875906</td>
<td>-2.211372</td>
<td>0.974466</td>
</tr>
<tr>
<td>1</td>
<td>-0.097883</td>
<td>0.695775</td>
<td>0.341734</td>
<td>-1.743161</td>
<td>-0.826591</td>
<td>-0.345352</td>
</tr>
<tr>
<td>2</td>
<td>-0.082240</td>
<td>-2.182937</td>
<td>0.380396</td>
<td>1.266143</td>
<td>0.299368</td>
<td>-0.863838</td>
</tr>
<tr>
<td>3</td>
<td>-0.489682</td>
<td>0.369374</td>
<td>-0.034571</td>
<td>0.221471</td>
<td>-0.744471</td>
<td>0.758527</td>
</tr>
<tr>
<td>4</td>
<td>0.901805</td>
<td>1.171216</td>
<td>0.520260</td>
<td>0.650776</td>
<td>-1.461665</td>
<td>-1.137707</td>
</tr>
<tr>
<td>5</td>
<td>-0.260838</td>
<td>0.281957</td>
<td>1.523962</td>
<td>-0.008434</td>
<td>1.952541</td>
<td>-1.056652</td>
</tr>
<tr>
<td>6</td>
<td>0.576897</td>
<td>1.146000</td>
<td>1.487349</td>
<td>2.015523</td>
<td>-1.833722</td>
<td>1.771740</td>
</tr>
</tbody>
</table>
23  0.065624  0.307665  -1.898358  1.389045  -0.873585  -0.699862  0.812477
24  1.010694  0.877138  -0.611561  -1.040389  -0.796211  0.241596  0.385922
25  -0.617855  0.536164  2.175585  1.872601  -2.513465  -0.139184  0.810491
26  0.937882  0.617547  0.287918  -1.584814  0.307941  1.809049  0.296237
27  -0.026233  -0.051744  0.001402  0.150664  -3.060395  0.040268  0.066091
28  -1.788308  0.753604  0.918071  0.922729  0.869610  0.364726  -0.226101
29  -0.481634  -2.056211  -2.106095  0.039227  0.211283  1.440190  -0.989193
[30 rows x 16 columns]

31.2.2 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]:
    v1    v2
28  0.410001 -0.078638
29  0.690579  0.995761
```

1178 Chapter 31. Comparison with R / R libraries
For more details and examples see the `groupby documentation`.

### 31.2.3 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:

```r
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`.

### 31.2.4 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
tapply(baseball$batting.average, baseball.example$team, max)
```

In pandas we may use `pivot_table()` method to handle this:
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(.200, .400, 25)
   })

In [17]:
   baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)

Out[17]:

```
         team  team 1  team 2  team 3  team 4  team 5
batting avg 0.394457 0.39573 0.343015 0.388863 0.377379
```

For more details and examples see the reshaping documentation.

### 31.2.5 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:

```r
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
```

```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
0 -1.003455 -0.990738
1  0.083515  0.548796
3 -0.524392  0.904400
4 -0.837804  0.746374
8 -0.507219  0.245479
```

```python
In [20]: df[df.a <= df.b]
```

```
a   b
0 -1.003455 -0.990738
1  0.083515  0.548796
3 -0.524392  0.904400
4 -0.837804  0.746374
8 -0.507219  0.245479
```

```python
In [21]: df.loc[df.a <= df.b]
```

```
a   b
0 -1.003455 -0.990738
```

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:
For more details and examples see the query documentation.

### 31.2.6 with

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  -0.920205
1  -0.860236
2   1.154370
3   0.188140
4  -1.163718
5   0.001397
6  -0.825694
7  -1.138198
8  -1.708034
9   1.148616
dtype: float64

In [24]: df.a + df.b    # same as the previous expression
```

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

### 31.3 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>

### 31.3.1 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., 168., 120),
         ....: 'y': np.random.uniform(7., 334., 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]:
           mean       std
    month week
   5     1  71.840596  52.886392
        2  71.904794  55.786805
        3  89.845632  49.892367
   6     1  97.730877  52.442172
        2  93.369836  47.178389
        3  96.592088  58.773744
   7     1  59.255715  43.442336
        2  69.634012  28.607369
        3  84.510992  59.761096
   8     1 104.787666  31.745437
        2  69.717872  53.747188
        3  79.892221  52.950459
```

For more details and examples see the `groupby documentation`. 
31.4 reshape / reshape2

31.4.1 melt.array

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))
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]:
   0 1 2 3
0 0 0 0 1.0
1 0 0 1 2.0
2 0 0 2 3.0
3 0 0 3 4.0
4 0 1 0 5.0
5 0 1 1 6.0
6 0 1 2 7.0
... ... ... ...
17 1 1 1 18.0
18 1 1 2 19.0
19 1 1 3 20.0
20 1 2 0 21.0
21 1 2 1 22.0
22 1 2 2 23.0
23 1 2 3 NaN
[24 rows x 4 columns]
```

31.4.2 melt.list

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.

```python
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.
31.4.3 melt.data.frame

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

```r
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:

```python
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
```

```python
In [34]: cheese.set_index(['first', 'last']).stack()  # alternative way
```

```
\rightarrow
    first last
  John  Doe    height  5.5
         weight 130.0
  Mary  Bo    height  6.0
         weight 150.0
dtype: float64
```

For more details and examples see the reshaping documentation.

31.4.4 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()`:
In [35]: df = pd.DataFrame({
    ....:     'x': np.random.uniform(1., 168., 12),
    ....:     'y': np.random.uniform(7., 334., 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>month</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>114.001700</td>
<td>132.227290</td>
<td>65.808204</td>
</tr>
<tr>
<td></td>
<td>124.669553</td>
<td>147.495706</td>
<td>82.882820</td>
</tr>
<tr>
<td>y</td>
<td>225.636630</td>
<td>301.864228</td>
<td>91.706834</td>
</tr>
<tr>
<td></td>
<td>57.692665</td>
<td>215.851669</td>
<td>218.004383</td>
</tr>
<tr>
<td>z</td>
<td>17.793871</td>
<td>7.124644</td>
<td>17.679823</td>
</tr>
<tr>
<td></td>
<td>15.068355</td>
<td>13.873974</td>
<td>9.394966</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({
    ....:     'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
    ....:                    'Animal2', 'Animal3'],
    ....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
    ....:     'Amount': [10, 7, 4, 2, 5, 6, 2],
    ....: })

In [39]: df.pivot_table(values='Amount', index='Animal', columns='FeedType', aggfunc='sum')

Out[39]:

<table>
<thead>
<tr>
<th>FeedType</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal1</td>
<td>10.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Animal2</td>
<td>2.0</td>
<td>13.0</td>
</tr>
<tr>
<td>Animal3</td>
<td>6.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

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

31.4.5 factor

pandas has a data type for categorical data.

cut(c(1,2,3,4,5,6), 3)
factor(c(1,2,3,2,2,3))

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

In [41]: pd.cut(pd.Series([1,2,3,4,5,6]), 3)
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]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]

In [42]: pd.Series([1,2,3,2,2,3]).astype("category")

→
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:

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

```
In [3]: url = 'https://raw.github.com/pandas-dev/pandas/master/pandas/tests/data/tips.csv'
In [4]: tips = pd.read_csv(url)
In [5]: tips.head()
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
```

### 32.1 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
LIMIT 5;
```

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

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
Out[6]:
   total_bill  tip  smoker  time
0       16.99  1.01  Female  None
1       10.34  1.66     Male  None
2       21.01  3.50     Male  None
3       23.68  3.31     Male  None
4       24.59  3.61  Female  None
```
Calling the DataFrame without the list of column names would display all columns (akin to SQL’s `*`).

### 32.2 WHERE

Filtering in SQL is done via a WHERE clause.

```sql
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

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

```python
In [7]: tips[tips['time'] == 'Dinner'].head(5)
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
```

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

```python
In [8]: is_dinner = tips['time'] == 'Dinner'
In [9]: is_dinner.value_counts()
Out[9]:
True    176
False    68
Name: time, dtype: int64
In [10]: tips[is_dinner].head(5)
```

Just like SQL’s OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

```sql
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```
# tips of more than $5.00 at Dinner meals

```python
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
```

```
Out[11]:
   total_bill  tip   sex  smoker  day  time  size
0    39.42  7.58  Male  No  Sat  Dinner  4
1    30.40  5.60  Male  No  Sun  Dinner  4
2    32.40  6.00  Male  No  Sun  Dinner  4
3    34.81  5.20  Female  No  Sun  Dinner  4
4    48.27  6.73  Male  No  Sat  Dinner  4
5   29.93  5.07  Male  No  Sun  Dinner  4
6   29.85  5.14  Female  No  Sun  Dinner  5
7   50.81 10.00  Male  Yes  Sat  Dinner  3
8    7.25  5.15  Male  Yes  Sun  Dinner  2
9   23.33  5.65  Male  Yes  Sun  Dinner  2
10  23.17  6.50  Male  Yes  Sun  Dinner  4
11  25.89  5.16  Male  Yes  Sat  Dinner  4
12  48.33  9.00  Male  No  Sat  Dinner  4
13  28.17  6.50  Male  Yes  Sun  Dinner  4
14  23.17  6.50  Male  Yes  Sun  Dinner  4
15  25.89  5.16  Male  Yes  Sat  Dinner  4
16  48.33  9.00  Male  No  Sat  Dinner  4
17  28.17  6.50  Male  Yes  Sun  Dinner  4
```

```sql
-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

# tips by parties of at least 5 diners OR bill total was more than $45

```python
In [12]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
```

```
Out[12]:
   total_bill  tip   sex  smoker  day  time  size
4    48.27  6.73  Male  No  Sat  Dinner  4
7    29.80  4.20  Female  No  Thur  Lunch  6
10  34.30  6.70  Male  No  Thur  Lunch  6
11  41.19  5.00  Male  No  Thur  Lunch  5
12  27.05  5.16  Male  Yes  Thur  Lunch  6
13  29.85  5.14  Female  No  Sun  Dinner  5
14  48.17  5.00  Male  No  Sun  Dinner  6
15  50.81 10.00  Male  Yes  Sat  Dinner  3
16  45.35  3.50  Male  Yes  Sun  Dinner  3
17  20.69  5.00  Male  No  Sun  Dinner  5
18  30.46  2.00  Male  Yes  Sun  Dinner  5
19  48.33  9.00  Male  No  Sat  Dinner  4
20  28.15  3.00  Male  Yes  Sat  Dinner  5
```

NULL checking is done using the `notna()` and `isna()` methods.

```python
```

```python
In [14]: frame
```

```
Out[14]:
   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 \texttt{col2} IS NULL with the following query:

\begin{verbatim}
SELECT * 
FROM frame 
WHERE \texttt{col2} IS NULL;
\end{verbatim}

\texttt{In [15]: frame[frame[\texttt{col2}].isna()]
Out [15]:
   col1  col2
0   A    F
1   B    NaN
3   C    H
4   D    I
}

Getting items where \texttt{col1} IS NOT NULL can be done with \texttt{notna()}.

\begin{verbatim}
SELECT * 
FROM frame 
WHERE \texttt{col1} IS NOT NULL;
\end{verbatim}

\texttt{In [16]: frame[frame[\texttt{col1}].notna()]
Out [16]:
   col1  col2
0   A    F
1   B    NaN
3   C    H
4   D    I
}

### 32.3 GROUP BY

In pandas, SQL's GROUP BY operations are performed using the similarly named \texttt{groupby()} method. \texttt{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:

\begin{verbatim}
SELECT sex, \texttt{count(*)} 
FROM tips 
GROUP BY sex;
/*
Female  87
Male    157
*/
\end{verbatim}

The pandas equivalent would be:

\begin{verbatim}
In [17]: tips.groupby('sex').size()
Out [17]:
   sex    
Female  87
Male    157
dtype: int64
\end{verbatim}

Notice that in the pandas code we used \texttt{size()} and not \texttt{count()}. This is because \texttt{count()} applies the function to each column, returning the number of not null records within each.
Alternatively, we could have applied the `count()` method to an individual column:

```
In [19]: tips.groupby('sex')['total_bill'].count()
Out[19]:
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
Thur 2.771452 62
*/
```

```
In [20]: tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
Out[20]:
   tip  day
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 [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})

Out[21]:

<table>
<thead>
<tr>
<th>smoker</th>
<th>day</th>
<th>tip</th>
<th>size</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Fri</td>
<td>tip</td>
<td>4.0</td>
<td>2.812500</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td></td>
<td>45.0</td>
<td>3.102889</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td></td>
<td>57.0</td>
<td>3.167895</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td></td>
<td>45.0</td>
<td>2.673778</td>
</tr>
<tr>
<td>Yes</td>
<td>Fri</td>
<td></td>
<td>15.0</td>
<td>2.714000</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td></td>
<td>42.0</td>
<td>2.875476</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td></td>
<td>19.0</td>
<td>3.516842</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td></td>
<td>17.0</td>
<td>3.030000</td>
</tr>
</tbody>
</table>

32.4 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).

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.

32.4.1 INNER JOIN

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

```
# merge performs an INNER JOIN by default
In [24]: pd.merge(df1, df2, on='key')

Out[24]:

<table>
<thead>
<tr>
<th>key</th>
<th>value_x</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.318214</td>
<td>0.543581</td>
</tr>
<tr>
<td>1</td>
<td>2.169960</td>
<td>-0.426067</td>
</tr>
<tr>
<td>2</td>
<td>2.169960</td>
<td>1.138079</td>
</tr>
</tbody>
</table>
```

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

```
In [25]: indexed_df2 = df2.set_index('key')
In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)
```
32.4.2 LEFT OUTER JOIN

```python
-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
ON df1.key = df2.key;
```

```
In [27]: pd.merge(df1, df2, on='key', how='left')
Out[27]:
   key  value_x  value_y
0    A  0.116174   NaN
1    B -0.318214  0.543581
2    C  0.285261   NaN
3    D  2.169960 -0.426067
4    D  2.169960  1.138079
```

32.4.3 RIGHT JOIN

```python
-- show all records from df2
SELECT *
FROM df1
RIGHT OUTER JOIN df2
ON df1.key = df2.key;
```

```
In [28]: pd.merge(df1, df2, on='key', how='right')
Out[28]:
   key  value_x  value_y
0    B -0.318214  0.543581
1    D  2.169960 -0.426067
2    D  2.169960  1.138079
3    E   NaN    0.086073
```

32.4.4 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).

```python
-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
ON df1.key = df2.key;
```

32.4. JOIN
# show all records from both frames

```python
In [29]: pd.merge(df1, df2, on='key', how='outer')
```

```
Out[29]:

<table>
<thead>
<tr>
<th>key</th>
<th>value_x</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 A</td>
<td>0.116174</td>
<td>NaN</td>
</tr>
<tr>
<td>1 B</td>
<td>-0.318214</td>
<td>0.543581</td>
</tr>
<tr>
<td>2 C</td>
<td>0.285261</td>
<td>NaN</td>
</tr>
<tr>
<td>3 D</td>
<td>2.169960</td>
<td>-0.426067</td>
</tr>
<tr>
<td>4 D</td>
<td>2.169960</td>
<td>1.138079</td>
</tr>
<tr>
<td>5 E</td>
<td>NaN</td>
<td>0.086073</td>
</tr>
</tbody>
</table>
```

## 32.5 UNION

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

```python
In [30]: df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],
                      'rank': range(1, 4)})

In [31]: df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],
                      'rank': [1, 4, 5]})

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 [32]: pd.concat([df1, df2])
```

```
Out[32]:

<table>
<thead>
<tr>
<th>city</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>1</td>
</tr>
<tr>
<td>San Francisco</td>
<td>2</td>
</tr>
<tr>
<td>New York City</td>
<td>3</td>
</tr>
<tr>
<td>Chicago</td>
<td>1</td>
</tr>
<tr>
<td>Boston</td>
<td>4</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>5</td>
</tr>
</tbody>
</table>
```

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 [33]: pd.concat([df1, df2]).drop_duplicates()
Out[33]:
   city  rank
0    Chicago   1
1        San Francisco   2
2     New York City   3
1        Boston       4
2    Los Angeles    5
```

### 32.6 Pandas equivalents for some SQL analytic and aggregate functions

#### 32.6.1 Top N rows with offset

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

```
In [34]: tips.nlargest(10+5, columns='tip').tail(10)
Out[34]:
      total_bill  sex  smoker  day  time  size
   183  23.17        Male    Yes  Sun  Dinner   4
   214  28.17  Female    Yes  Sat  Dinner   3
    47  32.40        Male     No  Sun  Dinner   4
   239  29.03        Male     No  Sat  Dinner   3
    88  24.71        Male     No Thur  Lunch   2
   181  23.33        Male    Yes  Sun  Dinner   2
    44  30.40        Male     No  Sun  Dinner   4
    52  34.81  Female     No  Sun  Dinner   4
    85  34.83  Female     No Thur  Lunch   4
   211  25.89        Male    Yes  Sat  Dinner   4
```

#### 32.6.2 Top N rows per group

```
-- Oracle’s ROW_NUMBER() analytic function
SELECT * FROM (SELECT
    t.*,
```

32.6. Pandas equivalents for some SQL analytic and aggregate functions
```python
ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn
FROM tips t
)
WHERE rn < 3
ORDER BY day, rn;
```

```plaintext
In [35]: (tips.assign(rn=tips.sort_values(['total_bill'], ascending=False)
....: .groupby(['day'])
....: .cumcount() + 1)
....: .query('rn < 3')
....: .sort_values(['day','rn'])
....: )

Out[35]:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>40.17</td>
<td>4.73</td>
<td>Male</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>90</td>
<td>28.97</td>
<td>3.00</td>
<td>Male</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>170</td>
<td>50.81</td>
<td>10.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>212</td>
<td>48.33</td>
<td>9.00</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>156</td>
<td>48.17</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>6</td>
</tr>
<tr>
<td>182</td>
<td>45.35</td>
<td>3.50</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>197</td>
<td>43.11</td>
<td>5.00</td>
<td>Female</td>
<td>Yes</td>
<td>Thur</td>
<td>Lunch</td>
<td>4</td>
</tr>
<tr>
<td>142</td>
<td>41.19</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>5</td>
</tr>
</tbody>
</table>
```

the same using `rank(method='first')` function

```plaintext
In [36]: (tips.assign(rnk=tips.groupby(['day'])['total_bill']
....: .rank(method='first', ascending=False))
....: .query('rnk < 3')
....: .sort_values(['day','rnk'])
....: )

Out[36]:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>40.17</td>
<td>4.73</td>
<td>Male</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>90</td>
<td>28.97</td>
<td>3.00</td>
<td>Male</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>170</td>
<td>50.81</td>
<td>10.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>212</td>
<td>48.33</td>
<td>9.00</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>156</td>
<td>48.17</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>6</td>
</tr>
<tr>
<td>182</td>
<td>45.35</td>
<td>3.50</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>197</td>
<td>43.11</td>
<td>5.00</td>
<td>Female</td>
<td>Yes</td>
<td>Thur</td>
<td>Lunch</td>
<td>4</td>
</tr>
<tr>
<td>142</td>
<td>41.19</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>5</td>
</tr>
</tbody>
</table>
```

-- Oracle's RANK() analytic function

```sql
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 [37]:
   
   (tips[tips['tip'] < 2]
   ....:   .assign(rnk_min=tips.groupby(['sex'])['tip']
   ....:   ....:       .rank(method='min'))
   ....:   ....:       .query('rnk_min < 3')
   ....:   ....:       .sort_values(['sex','rnk_min'])
   ....:   )
   
   Out[37]:
   
<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>rnk_min</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>3.07</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>92</td>
<td>5.75</td>
<td>Female</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>111</td>
<td>7.25</td>
<td>Female</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>236</td>
<td>12.60</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>237</td>
<td>32.83</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>2.0</td>
</tr>
</tbody>
</table>

### 32.7 UPDATE

**UPDATE tips**

SET tip = tip*2

WHERE tip < 2;

In [38]:
   
   tips.loc[tips['tip'] < 2, 'tip'] *= 2

### 32.8 DELETE

**DELETE FROM tips**

WHERE tip > 9;

In pandas we select the rows that should remain, instead of deleting them

In [39]:
   
   tips = tips.loc[tips['tip'] <= 9]
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:

```
In [1]: import pandas as pd
In [2]: import numpy as np
```

**Note:** Throughout this tutorial, the pandas DataFrame will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the DataFrame. This is often used in interactive work (e.g. Jupyter notebook or terminal) - the equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

### 33.1 Data Structures

#### 33.1.1 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>

#### 33.1.2 DataFrame / Series

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.
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.

### 33.1.3 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 row 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.

### 33.2 Data Input / Output

#### 33.2.1 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 [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
```

#### 33.2.2 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.
The pandas method is `read_csv()`, which works similarly.

```python
In [5]: url = 'https://raw.github.com/pandas-dev/pandas/master/pandas/tests/data/tips.csv'

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

In [7]: tips.head()
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    3
4      24.59  3.61 Female  No  Sun  Dinner    4
```

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.

### 33.2.3 Exporting Data

The inverse of PROC IMPORT in SAS is PROC EXPORT

```bash
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.

```python
tips.to_csv('tips2.csv')
```
pandas provides similar vectorized operations by specifying the individual Series in the DataFrame. New columns can be assigned in the same way.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2
In [9]: tips['new_bill'] = tips['total_bill'] / 2.0
In [10]: tips.head()
Out[10]:
          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
```

### 33.3.2 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 [11]: tips[tips['total_bill'] > 10].head()
Out[11]:
          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
```

### 33.3.3 If/Then Logic

In SAS, if/then logic can be used to create new columns.

```
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 [12]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')
```

```python
In [13]: tips.head()
Out[13]:
   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
```

### 33.3.4 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;
   date1_next = intnx('MONTH', date1, 1);
   * count intervals between dates;
   months_between = intck('MONTH', date1, 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 and custom offsets) - see the `timeseries documentation` for more details.

```python
In [14]: tips['date1'] = pd.Timestamp('2013-01-15')
In [15]: tips['date2'] = pd.Timestamp('2015-02-15')
In [16]: tips['date1_year'] = tips['date1'].dt.year
In [17]: tips['date2_month'] = tips['date2'].dt.month
In [18]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()
In [19]: tips['months_between'] = (tips['date2'].dt.to_period('M') -
                                tips['date1'].dt.to_period('M'))
In [20]: tips[['date1', 'date2', 'date1_year', 'date2_month',
                   'date1_next', 'months_between']].head()
```

```
       date1    date2  date1_year  date2_month date1_next  months_between
```
33.3.5 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;
  set tips;
  rename total_bill=total_bill_2;
run;
```

The same operations are expressed in pandas below.

```python
# keep
In [21]: tips[['sex', 'total_bill', 'tip']].head()
Out[21]:
   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

# drop
In [22]: tips.drop('sex', axis=1).head()
Out[22]:
   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

# rename
In [23]: tips.rename(columns={'total_bill':'total_bill_2'}).head()
Out[23]:
   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
```
33.3.6 Sorting by Values

Sorting in SAS is accomplished via **PROC SORT**

```sas
proc sort data=tips;
    by sex total_bill;
run;
```

Pandas objects have a **sort_values()** method, which takes a list of columns to sort by.

```python
In [24]: tips = tips.sort_values(['sex', 'total_bill'])

In [25]: tips.head()
Out[25]:
   total_bill  tip  sex  smoker  day  time  size
0        67  1.07 Female  Yes  Sat  Dinner  1
1        92  3.75 Female  Yes  Fri  Dinner  2
2       111  5.25 Female   No  Sat  Dinner  1
3       145  6.35 Female   No  Thur  Lunch  2
4       135  6.51 Female   No  Thur  Lunch  2
```

33.4 String Processing

33.4.1 Length

SAS determines the length of a character string with the **LENGTHN** and **LENGTHC** functions. **LENGTHN** excludes trailing blanks and **LENGTHC** includes trailing blanks.

```sas
data _null_;  
set tips;  
put(LENGTHN(time));  
put(LENGTHC(time));  
run;
```

Python determines the length of a character string with the **len** function. **len** includes trailing blanks. Use **len** and **rstrip** to exclude trailing blanks.

```python
In [26]: tips['time'].str.len().head()
Out[26]:
    0    6
   92    6
  111    6
  145    5
  135    5
Name: time, dtype: int64

In [27]: tips['time'].str.rstrip().str.len().head()
```

33.4 String Processing
### 33.4.2 Find

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.

```plaintext
data _null_;  
set tips;  
put(FINDW(sex,'ale'));  
run;
```

Python determines the position of a character in a string with the find function. find searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```plaintext
In [28]: tips['sex'].str.find("ale").head()
Out[28]:
       0
67   3
92   3
111  3
145  3
135  3
Name: sex, dtype: int64
```

### 33.4.3 Substring

SAS extracts a substring from a string based on its position with the SUBSTR function.

```plaintext
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.

```plaintext
In [29]: tips['sex'].str[0:1].head()
Out[29]:
       0
67   F
92   F
111  F
145  F
135  F
Name: sex, dtype: object
```

### 33.4.4 Scan

The SAS 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.

```plaintext
data firstlast;  
input String $60.;  
First_Name = scan(string, 1);  
Last_Name = scan(string, -1);  
datalines2;
```
Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```python
In [30]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})
In [31]: firstlast['First_Name'] = firstlast['String'].str.split(" ", expand=True)[0]
In [32]: firstlast['Last_Name'] = firstlast['String'].str.rsplit(" ", expand=True)[0]
In [33]: firstlast
Out[33]:
       String First_Name Last_Name
0   John Smith       John       John
1    Jane Cook       Jane       Jane
```

### 33.4.5 Upcase, Lowcase, and Propcase

The SAS `UPCASE LOWCASE` and `PROPCASE` functions change the case of the argument.

```sas
data firstlast;
input String $60.;
string_up = UPCASE(string);
string_low = LOWCASE(string);
string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
;
run;
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```python
In [34]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})
In [35]: firstlast['string_up'] = firstlast['String'].str.upper()
In [36]: firstlast['string_low'] = firstlast['String'].str.lower()
In [37]: firstlast['string_prop'] = firstlast['String'].str.title()
In [38]: firstlast
Out[38]:
     String  string_up  string_low  string_prop
   0  John Smith       JOHN SMITH          John Smith
   1    Jane Cook       JANE COOK          Jane Cook
```

### 33.5 Merging

The following tables will be used in the merge examples
In [39]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
....:                     'value': np.random.randn(4)})
....:

In [40]: df1
Out[40]:
   key  value
0   A  -0.857326
1   B   1.075416
2   C   0.371727
3   D   1.065735

In [41]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
....:                     'value': np.random.randn(4)})
....:

In [42]: df2
Out[42]:
   key  value
0   B  -0.227314
1   D   2.102726
2   D  -0.092796
3   E   0.094694

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.

```
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);
  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. Note that the data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

In [43]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [44]: inner_join
Out[44]:
   key  value_x  value_y
0   B  1.075416 -0.227314
1   D  1.065735  2.102726
2   D  1.065735 -0.092796

In [45]: left_join = df1.merge(df2, on=['key'], how='left')
In [46]: left_join
Out[46]:
   key  value_x  value_y
0   A  -0.857326   NaN
1   B   1.075416 -0.227314
2   C   0.371727   NaN
3   D   1.065735  2.102726
4   D   1.065735 -0.092796

In [47]: right_join = df1.merge(df2, on=['key'], how='right')

In [48]: right_join
Out[48]:
   key  value_x  value_y
0   B   1.075416 -0.227314
1   D   1.065735  2.102726
2   D   1.065735 -0.092796
3   E    NaN    0.094694

In [49]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [50]: outer_join
Out[50]:
   key  value_x  value_y
0   A  -0.857326   NaN
1   B   1.075416 -0.227314
2   C   0.371727   NaN
3   D   1.065735  2.102726
4   D   1.065735 -0.092796
5   E    NaN    0.094694

33.6 Missing Data

Like SAS, pandas has a representation for missing data - which is 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 [51]: outer_join
Out[51]:
   key  value_x  value_y
0   A  -0.857326   NaN
1   B   1.075416 -0.227314
2   C   0.371727   NaN
3   D   1.065735  2.102726
4   D   1.065735 -0.092796
5   E    NaN    0.094694

In [52]: outer_join['value_x'] + outer_join['value_y']

       0    NaN
1  0.8481016  NaN
2  3.1684611  NaN
3  0.9729392  NaN
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:

```sas
data outer_join_nulls;
  set outer_join;
  if value_x = .;
run;

data outer_join_no_nulls;
  set outer_join;
  if value_x ^= .;
run;
```

Which doesn’t work in pandas. Instead, the `pd.isna` or `pd.notna` functions should be used for comparisons.

```python
In [54]: outer_join[pd.isna(outer_join['value_x'])]
Out[54]:
   key  value_x  value_y
0    E    NaN     0.094694

In [55]: outer_join[pd.notna(outer_join['value_x'])]
Out[55]:
   key  value_x  value_y
0    A -0.857326    NaN
1    B  1.075416 -0.227314
2    C  0.371727    NaN
3    D  1.065735  2.102726
4    D  1.065735 -0.092796
5    E  1.065735   0.094694
```

pandas also provides a variety of methods to work with missing data - some of which would be challenging to express in SAS. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the `missing data documentation` for more.

```python
In [56]: outer_join.dropna()
Out[56]:
   key  value_x  value_y
1    B  1.075416 -0.227314
3    D  1.065735  2.102726
4    D  1.065735 -0.092796

In [57]: outer_join.fillna(method='ffill')
```

```python
->
   key  value_x  value_y
0    A -0.857326    NaN
1    B  1.075416 -0.227314
2    C  0.371727 -0.227314
3    D  1.065735  2.102726
4    D  1.065735 -0.092796
5    E  1.065735   0.094694
```
33.7 GroupBy

33.7.1 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.

```python
In [59]: tips_summed = tips.groupby(['sex', 'smoker'])[['total_bill', 'tip']].sum()
```

```console
Out[60]:
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
```

33.7.2 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;
```

```plaintext
proc sort data=tips;
    by smoker;
run;
```
pandas groupby provides a transform mechanism that allows these type of operations to be succinctly expressed in one operation.

```r
In [61]: gb = tips.groupby('smoker')['total_bill']
In [62]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')
In [63]: tips.head()
Out[63]:
   total_bill  tip  sex  smoker  day  time  size  adj_total_bill
0   1.0700  1.00 Female  Yes  Sat  Dinner  1  -17.686344
1   3.7500  1.00 Female  Yes  Fri  Dinner  2  -15.006344
2   5.2500  1.00 Female  No   Sat  Dinner  1   -11.938278
3   5.2500  1.00 Female  No  Thur  Lunch  2  -10.838278
4   6.5100  1.25 Female  No  Thur  Lunch  2  -10.678278
```

### 33.7.3 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.

```r
data tips_first;
set tips;
by sex smoker;
if FIRST.sex or FIRST.smoker then output;
run;
```

In pandas this would be written as:

```r
In [64]: tips.groupby(['sex','smoker']).first()
Out[64]:
   total_bill  tip  day  time  size  adj_total_bill
  sex  smoker
Female No  5.25  1.00  Sat  Dinner  1  -11.938278
  Yes  1.07  1.00  Sat  Dinner  1  -17.686344
Male No  5.51  2.00  Thur  Lunch  2  -11.678278
  Yes  5.25  5.15  Sun  Dinner  2  -13.506344
```

1212 Chapter 33. Comparison with SAS
33.8 Other Considerations

33.8.1 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.

33.8.2 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.

```python
# version 0.17, 10M rows
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
```
This page gives an overview of all public pandas objects, functions and methods. In general, all classes and functions exposed in the top-level `pandas.*` namespace are regarded as public.

Further some of the subpackages are public, including `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Certain functions in the the `pandas.io` and `pandas.tseries` submodules are public as well (those mentioned in the documentation). Further, the `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 considered to be PRIVATE. Stability of functionality in those modules is not guaranteed.

### 34.1 Input/Output

#### 34.1.1 Pickling

<table>
<thead>
<tr>
<th><code>read_pickle(path[, compression])</code></th>
<th>Load pickled pandas object (or any other pickled object) from the specified file path</th>
</tr>
</thead>
</table>

**34.1.1.1 pandas.read_pickle**

```python
pandas.read_pickle (path, compression='infer')
```

Load pickled pandas object (or any other pickled object) from the specified file path

**Warning:** Loading pickled data received from untrusted sources can be unsafe. See: [http://docs.python.org/2.7/library/pickle.html](http://docs.python.org/2.7/library/pickle.html)

**Parameters**

- `path` : string
  - File path

- `compression` : ['infer', 'gzip', 'bz2', 'xz', 'zip', None], default `infer`
  - For on-the-fly decompression of on-disk data. If `infer`, then use gzip, bz2, xz or zip if path ends in `.gz`, `.bz2`, `.xz`, or `.zip` respectively, and no decompression otherwise. Set to None for no decompression.

  New in version 0.20.0.

**Returns**

- `unpickled` : type of object stored in file
34.1.2 Flat File

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_table</td>
<td>Read general delimited file into DataFrame</td>
</tr>
<tr>
<td>read_csv</td>
<td>Read CSV (comma-separated) file into DataFrame</td>
</tr>
<tr>
<td>read_fwf</td>
<td>Read a table of fixed-width formatted lines into DataFrame</td>
</tr>
<tr>
<td>read_msgpack</td>
<td>Load msgpack pandas object from the specified path</td>
</tr>
</tbody>
</table>

34.1.2.1 pandas.read_table

**pandas.read_table** (filepath_or_buffer[, sep, ...,])

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, pathlib.Path, py._path.local.LocalPath or any object with a `read()` method (such as a file handle or 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://local-host/path/to/table.csv

- **sep**: str, default None (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

  Alternative argument name for sep.

- **delim_whitespace**: boolean, 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.

  New in version 0.18.1: support for the Python parser.

- **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 as if set to 0 if no names passed, otherwise 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, 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 will cause a UserWarning to be issued.

index_col : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

usecols : array-like or callable, default None

Return a subset of the columns. If array-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). For example, a valid array-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

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.

as_recarray : boolean, default False

Deprecated since version 0.19.0: Please call pd.read_csv(...).to_records() instead.

Return a NumPy recarray instead of a DataFrame after parsing the data. If set to True, this option takes precedence over the squeeze parameter. In addition, as row indices are not available in such a format, the index_col parameter will be ignored.

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

prefix : str, default None

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, ...

mangle_dupe_cols : boolean, default True

Duplicate columns will be specified as ‘X.0’...’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, default None

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use str or object to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine : {'c', 'python'}, optional

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.
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 or callable, 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. 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')

skip_footer : int, default 0

Deprecated since version 0.19.0: Use the `skipfooter` parameter instead, as they are identical

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

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’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

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.

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

parse_dates : boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
• list of ints 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 unparseable date, the entire 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

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format : boolean, default False

If True and parse_dates is enabled, pandas will attempt to infer the format of the date-time 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 : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

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

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 the IO Tools docs for more information on iterator and chunksize.

compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, 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', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

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.

**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**: 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 '#emptyna,b,cn1,2,3' 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 (ex. ‘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.

**tupleize_cols**: boolean, default False

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns)

**error_bad_lines**: boolean, default True

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.

**warn_bad_lines**: boolean, default True
If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

**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)

**buffer_lines** : int, default None

Deprecated since version 0.19.0: This argument is not respected by the parser

**compact_ints** : boolean, default False

Deprecated since version 0.19.0: Argument moved to `pd.to_numeric`

If `compact_ints` is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the `use_unsigned` parameter.

**use_unsigned** : boolean, default False

Deprecated since version 0.19.0: Argument moved to `pd.to_numeric`

If integer columns are being compacted (i.e. `compact_ints=True`), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

**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.

**Returns** `result` : DataFrame or TextParser

34.1.2.2 pandas.read_csv

```python
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=True, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, 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, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar=None, quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, doublequote=True, delimiterspace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
```

Read CSV (comma-separated) 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, pathlib.Path, py._path.local.LocalPath or any object with a read() method (such as a file handle or 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.csv

**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: '\r\t'

**delimiter**: str, default None

Alternative argument name for sep.

**delim_whitespace**: boolean, 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.

New in version 0.18.1: support for the Python parser.

**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 as if set to 0 if no names passed, otherwise 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, 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 will cause a UserWarning to be issued.

**index_col**: int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

**usecols**: array-like or callable, default None

Return a subset of the columns. If array-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). For example, a valid array-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

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.

**as_recarray**: boolean, default False

Deprecated since version 0.19.0: Please call pd.read_csv(...).to_records() instead.
Return a NumPy recarray instead of a DataFrame after parsing the data. If set to True, this option takes precedence over the `squeeze` parameter. In addition, as row indices are not available in such a format, the `index_col` parameter will be ignored.

**squeeze**: boolean, default False

If the parsed data only contains one column then return a Series

**prefix**: str, default None

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, ...

**mangle_dupe_cols**: boolean, default True

Duplicate columns will be specified as ‘X.0’...'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, default None

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use `str` or `object` to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**engine**: {'c', 'python'}, optional

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

**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 or callable, 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. 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')

**skip_footer**: int, default 0

Deprecated since version 0.19.0: Use the `skipfooter` parameter instead, as they are identical

**nrows**: int, default None

Number of rows of file to read. Useful for reading pieces of large files

**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’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are over-ridden, otherwise they’re appended to.

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

parse_dates : boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
- list of ints 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. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format : boolean, default False

If True and parse_dates is enabled, pandas will attempt to infer the format of the date-time 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 : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

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

**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 the IO Tools docs for more information on iterator and chunksize.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, 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`, or `.xz` (otherwise no decompression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for ‘zip’ and ‘xz’ compression.

**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.

**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** : 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 '#emptyna,b,cn1,2,3' 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 (ex. ‘utf-8’). See 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.

tupleize_cols : boolean, default False

   Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex
   Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns)

error_bad_lines : boolean, default True

   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.

warn_bad_lines : boolean, default True

   If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

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)

buffer_lines : int, default None

   Deprecated since version 0.19.0: This argument is not respected by the parser

compact_ints : boolean, default False

   Deprecated since version 0.19.0: Argument moved to pd.to_numeric

   If compact_ints is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the use_unsigned parameter.

use_unsigned : boolean, default False

   Deprecated since version 0.19.0: Argument moved to pd.to_numeric

   If integer columns are being compacted (i.e. compact_ints=True), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

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.

Returns result : DataFrame or TextParser
34.1.2.3 pandas.read_fwf

```
pandas.read_fwf(filepath_or_buffer, colspecs='infer', widths=None, **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, pathlib.Path, py._path.local.LocalPath or any object with a read() method (such as a file handle or 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.csv

- **colspecs**: list of pairs (int, int) or 'infer'. optional
  
  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 which are not being skipped via skiprows (default='infer').

- **widths**: list of ints. optional
  
  A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.

- **delimiter**: str, default ' ' + ' '
  
  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., '~').

- **delim_whitespace**: boolean, 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.
  
  New in version 0.18.1: support for the Python parser.

- **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 as if set to 0 if no names passed, otherwise 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, 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 will cause a UserWarning to be issued.

- **index_col**: int or sequence or False, default None
  
  Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

- **usecols**: array-like or callable, default None
Return a subset of the columns. If array-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). For example, a valid array-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

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.

**as_recarray** : boolean, default False

Deprecated since version 0.19.0: Please call pd.read_csv(...).to_records() instead.

Return a NumPy recarray instead of a DataFrame after parsing the data. If set to True, this option takes precedence over the squeeze parameter. In addition, as row indices are not available in such a format, the index_col parameter will be ignored.

**squeeze** : boolean, default False

If the parsed data only contains one column then return a Series

**prefix** : str, default None

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, ...

**mangle_dupe_cols** : boolean, default True

Duplicate columns will be specified as ‘X.0’...’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, default None

Data type for data or columns. E.g. `{‘a’: np.float64, ‘b’: np.int32}` Use str or object to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**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 or callable, 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. 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')

**skip_footer** : int, default 0
pandas: powerful Python data analysis toolkit, Release 0.21.0

Deprecated since version 0.19.0: Use the `skipfooter` parameter instead, as they are identical

**nrows**: int, default None

Number of rows of file to read. Useful for reading pieces of large files

**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’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

**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.

**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

**parse_dates**: boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
- list of ints 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 unparseable date, the entire 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`

Note: A fast-path exists for iso8601-formatted dates.

**infer_datetime_format**: boolean, 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**: boolean, default False

If True and `parse_dates` specifies combining multiple columns then keep the original columns.

**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

**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 the IO Tools docs for more information on `iterator` and `chunksize`.

**compression**: [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}], 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', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

**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.

**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**: 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 ‘#emptyna,b,cn1,2,3’ 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 (ex. ‘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.

**tupleize_cols :** boolean, default False

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns)

**error_bad_lines :** boolean, default True

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.

**warn_bad_lines :** boolean, default True

If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

**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)

**buffer_lines :** int, default None

Deprecated since version 0.19.0: This argument is not respected by the parser

**compact_ints :** boolean, default False

Deprecated since version 0.19.0: Argument moved to pd.to_numeric

If compact_ints is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the use_unsigned parameter.

**use_unsigned :** boolean, default False

Deprecated since version 0.19.0: Argument moved to pd.to_numeric

If integer columns are being compacted (i.e. compact_ints=True), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.
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.

Returns result : DataFrame or TextParser

34.1.2.4 pandas.read_msgpack

pandas.read_msgpack (path_or_buf, encoding='utf-8', iterator=False, **kwargs)

Load msgpack pandas object from the specified file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters path_or_buf : string File path, BytesIO like or string

encoding: Encoding for decoding msgpack str type

iterator : boolean, if True, return an iterator to the unpacker
(default is False)

Returns obj : type of object stored in file

34.1.3 Clipboard

_read_clipboard(sep)

Read text from clipboard and pass to read_table.

34.1.3.1 pandas.read_clipboard

pandas.read_clipboard(sep='\s+', **kwargs)

Read text from clipboard and pass to read_table. See read_table for the full argument list

Parameters sep : str, default ‘s+’.

A string or regex delimiter. The default of ‘s+’ denotes one or more whitespace characters.

Returns parsed : DataFrame

34.1.4 Excel

read_excel(io[, sheet_name, header, ...])

Read an Excel table into a pandas DataFrame

ExcelFile.parse([sheet_name, header, ...])

Parse specified sheet(s) into a DataFrame

34.1.4.1 pandas.read_excel

pandas.read_excel(io, sheet_name=0, header=0, skiprows=None, skip_footer=0, index_col=None, names=None, usecols=None, parse_dates=False, date_parser=None, na_values=None, thousands=None, converters=None, dtype=None, true_values=None, false_values=None, engine=None, squeeze=False, **kwds)

Read an Excel table into a pandas DataFrame
Parameters

io : string, path object (pathlib.Path or py.path.local.LocalPath),
    file-like object, pandas ExcelFile, or xlrd workbook. 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/workbook.xlsx

sheet_name : string, int, mixed list of strings/int, or None, default 0
    Strings are used for sheet names, Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets.

sheetname : string, int, mixed list of strings/int, or None, default 0
    Deprecated since version 0.21.0: Use sheet_name instead

case : int, list of ints, 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.

skiprows : list-like
    Rows to skip at the beginning (0-indexed)

skip_footer : int, default 0
    Rows at the end to skip (0-indexed)

index_col : int, list of ints, 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.

names : array-like, default None
    List of column names to use. If file contains no header row, then you should explicitly pass header=None

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.

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.

New in version 0.20.0.

**true_values** : list, default None

Values to consider as True

New in version 0.19.0.

**false_values** : list, default None

Values to consider as False

New in version 0.19.0.

**parse_cols** : int or list, default None

- If None then parse all columns,
- If int then indicates last column to be parsed
- If list of ints then indicates list of column numbers to be parsed
- If string 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.

**squeeze** : boolean, default False

If the parsed data only contains one column then return a Series

**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’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

**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.

**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.

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**engine** : string, default None

- If io is not a buffer or path, this must be set to identify io. Acceptable values are None or xlrd
- If converters are specified, they will be applied INSTEAD of dtype conversion.

New in version 0.20.0.

**convert_float** : boolean, 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

**Returns parsed**: 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.

### 34.1.4.2 pandas.ExcelFile.parse

```
pandas.ExcelFile.parse(
    sheet_name=0,
    header=0,
    skiprows=None,
    skip_footer=0,
    index_col=None,
    usecols=None,
    parse_dates=False,
    date_parser=None,
    na_values=None,
    thousands=None,
    converters=None,
    index_col=none,
    parse_dates=False,
    date_parser=None,
    na_values=None,
    thousands=None,
    converters=None,
    **kwds
)
```

Parse specified sheet(s) into a DataFrame

Equivalent to read_excel(ExcelFile, ...) See the read_excel docstring for more info on accepted parameters

### 34.1.5 JSON

#### 34.1.5.1 pandas.read_json

```
pandas.read_json(
    path_or_buf=None,
    orient=None,
    typ='frame',
    dtype=True,
    convert_axes=True,
    convert_dates=True,
    keep_default_dates=True,
    numpy=False,
    precise_float=False,
    date_unit=None,
    encoding=None,
    lines=False,
    chunksize=None,
    compression='infer'
)
```

Convert a JSON string to pandas object

**Parameters path_or_buf**: a valid JSON string or file-like, default: None

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

**orient**: string.

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'}
- 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** : type of object to recover (series or frame), default 'frame'

**dtype** : boolean or dict, default True

  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.

**convert_axes** : boolean, default True

  Try to convert the axes to the proper dtypes.

**convert_dates** : boolean, default True

  List of columns to parse for dates; If True, then try to parse datelike columns default is True; 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'

**keep_default_dates** : boolean, default True

  If parsing dates, then parse the default datelike columns

**numpy** : boolean, 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.

**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 builtin functionality.

**date_unit** : string, 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.

**lines** : boolean, default False

  Read the file as a json object per line.

  New in version 0.19.0.
encoding: str, default is ‘utf-8’

The encoding to use to decode py3 bytes.
New in version 0.19.0.

chunksize: integer, default None

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.
New in version 0.21.0.

compression: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default ‘infer’

For on-the-fly decompression of on-disk data. If ‘infer’, then use gzip, bz2, zip or xz if path_or_buf is a string ending in ‘.gz’, ‘.bz2’, ‘.zip’, or ‘.xz’, 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.
New in version 0.21.0.

Returns result: Series or DataFrame, depending on the value of typ.

See also:

DataFrame.to_json

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"]}')
```

```python
>>> 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"})
```

```python
>>> 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.
pandas: powerful Python data analysis toolkit, Release 0.21.0

```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

```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": "0.20.0"},
    "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
              {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

---

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>json_normalize(data[, record_path, meta, ...])</code></td>
<td>“Normalize” semi-structured JSON data into a flat table</td>
</tr>
<tr>
<td><code>build_table_schema(data[, index, ...])</code></td>
<td>Create a Table schema from data.</td>
</tr>
</tbody>
</table>

---

### 34.1.5.2 pandas.io.json.json_normalize

**pandas.io.json.json_normalize**

```
pandas.io.json.json_normalize(data, record_path=None, meta=None, meta_prefix=None, record_prefix=None, errors='raise', sep='.')
```

“Normalize” semi-structured JSON data into a flat table

**Parameters**

- **data**: dict or list of dicts
  - Unserialized JSON objects
- **record_path**: string or list of strings, 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 (string or list of strings), default None
  - Fields to use as metadata for each record in resulting table
- **record_prefix**: string, default None
  - If True, prefix records with dotted (?) path, e.g. foo.bar.field if path to records is ['foo', 'bar']
- **meta_prefix**: string, default None
- **errors**: {'raise', 'ignore'}, default ‘raise’
  - ‘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
  - New in version 0.20.0.
- **sep**: string, default ‘.’
  - Nested records will generate names separated by sep, e.g., for sep=’.’, { ‘foo’ : { ‘bar’ :
0 } } -> foo.bar
  - New in version 0.20.0.
Returns frame: DataFrame

Examples

```python
>>> from pandas.io.json import json_normalize
>>> data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
          ...
          {'id': 2, 'name': 'Faye Raker'}]
>>> json_normalize(data)
     id  name  name.family  name.first  name.given  name.last
0  1.0  NaN      NaN        Coleen      NaN      Volk
1 NaN  NaN      Regner      NaN        Mose      NaN
2  2.0  NaN      NaN        NaN         NaN       NaN
```

```python
>>> data = [{'state': 'Florida',
          ...
          'info': {'governor': 'Rick Scott'},
          ...
          'counties': [{'name': 'Dade', 'population': 12345},
                        ...
                        {'name': 'Palm Beach', 'population': 60000}],
          ...
          'state': 'Ohio',
          ...
          'info': {'governor': 'John Kasich'},
          ...
          'counties': [{'name': 'Summit', 'population': 1234},
                        ...
                        {'name': 'Cuyahoga', 'population': 1337}]
          }
>>> result = json_normalize(data, 'counties', ['state', 'shortname', 'info.governor'])
>>> result
   name  population  info.governor         state  shortname
0     Dade         12345   Rick Scott       Florida      FL
1   Broward         40000   Rick Scott       Florida      FL
2  Palm Beach       60000   Rick Scott       Florida      FL
3    Summit          1234   John Kasich      Ohio          OH
4  Cuyahoga          1337   John Kasich      Ohio          OH
```

34.1.5.3 pandas.io.json.build_table_schema

pandas.io.json.build_table_schema(data, index=True, primary_key=None, version=True)

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 generated the schema.

Returns `schema : dict`

Notes

See `_as_json_table_type` 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'}],
'pandas_version': '0.20.0',
'primaryKey': ['idx']}
```

34.1.6 HTML

`read_html(io[, match, flavor, header, ...])` Read HTML tables into a list of DataFrame objects.

34.1.6.1 pandas.read_html

`pandas.read_html(io, match='.+', flavor=None, header=None, index_col=None, skiprows=None, attrs=None, parse_dates=False, tupleize_cols=None, thousands='.', encoding=None, decimal='.', converters=None, na_values=None, keep_default_na=True)` Read HTML tables into a list of DataFrame objects.

Parameters

- `io` : str or file-like

  A URL, a file-like object, or a raw string containing HTML. 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 or None, container of strings

  Options are `lxml` and `bs4`. `bs4` performs better on pages with more complex HTML, `lxml` is faster for more simple pages.
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 or None, optional

The row (or list of rows for a MultiIndex) to use to make the columns headers.

index_col : int or list-like or None, optional

The column (or list of columns) to use to create the index.

skiprows : int or list-like or slice or None, optional

0-based. Number of rows to skip after parsing the column integer. 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 or None, 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.

tupleize_cols : bool, optional

If False try to parse multiple header rows into a MultiIndex, otherwise return raw tuples. Defaults to False.

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

thousands : str, optional

Separator to use to parse thousands. Defaults to ‘, ’.

encoding : str or None, 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).

decimal : str, default ‘.’

Character to recognize as decimal point (e.g. use ‘,’ for European data).

New in version 0.19.0.
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 cell (not column)
    content, and return the transformed content.
    New in version 0.19.0.

na_values : iterable, default None
    Custom NA values
    New in version 0.19.0.

keep_default_na : bool, default True
    If na_values are specified and keep_default_na is False the default NaN values are over-
    ridden, otherwise they’re appended to
    New in version 0.19.0.

Returns  dfs : list of DataFrames

See also:
Pandas.read_csv

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”.

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.

34.1.7 HDFStore: PyTables (HDF5)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_hdf(path_or_buf, key, mode)</td>
<td>read from the store, close it if we opened it</td>
</tr>
<tr>
<td>HDFStore.put(key, value[, format, append])</td>
<td>Store object in HDFStore</td>
</tr>
<tr>
<td>HDFStore.append(key, value[, format, ...])</td>
<td>Append to Table in file.</td>
</tr>
<tr>
<td>HDFStore.get(key)</td>
<td>Retrieve pandas object stored in file</td>
</tr>
<tr>
<td>HDFStore.select(key[, where, start, stop, ...])</td>
<td>Retrieve pandas object stored in file, optionally based on where</td>
</tr>
<tr>
<td>HDFStore.info()</td>
<td>print detailed information on the store</td>
</tr>
</tbody>
</table>
34.1.7.1 pandas.read_hdf

```python
pandas.read_hdf(path_or_buf, key=None, mode='r', **kwargs)
```

Read from the store, close it if we opened it

Parameters:
- **path_or_buf**: path (string), buffer or path object (pathlib.Path or py._path.local.LocalPath) designating the file to open, or an already opened pd.HDFStore object
  
  New in version 0.19.0: support for pathlib, py.path.

- **key**: group identifier in the store. Can be omitted if the HDF file contains a single pandas object.

- **mode**: string, {'r', 'r+', 'a'}, default ‘r’. Mode to use when opening the file. Ignored if path_or_buf is a pd.HDFStore.

- **where**: list of Term (or convertable) objects, optional

- **start**: optional, integer (defaults to None), row number to start selection

- **stop**: optional, integer (defaults to None), row number to stop selection

- **columns**: optional, a list of columns that if not None, will limit the return columns

- **iterator**: optional, boolean, return an iterator, default False

- **chunksize**: optional, nrows to include in iteration, return an iterator

Returns: The selected object

34.1.7.2 pandas.HDFStore.put

```python
HDFStore.put(key, value, format=None, append=False, **kwargs)
```

Store object in HDFStore

Parameters:
- **key**: object

- **value**: {Series, DataFrame, Panel}

- **format**: ‘fixed(f)|table(t)’, default is ‘fixed’
  
  - **fixed(f)** [Fixed format] Fast writing/reading. Not-appendable, nor searchable
  
  - **table(t)** [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**: boolean, default False
  
  This will force Table format, append the input data to the existing.

- **data_columns**: list of columns to create as data columns, or True to use all columns. See here # noqa
**34.1.7.3 pandas.HDFStore.append**

```
HDFStore.append(key, value, format=None, append=True, columns=None, dropna=None, **kwargs)
```

Append to Table in file. Node must already exist and be Table format.

**Parameters**

- **key**: object
- **value**: {Series, DataFrame, Panel, Panel4D}
- **format**: ‘table’ is the default
  - Use `table(t)` for PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data
- **append**: boolean, 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 string sizes
- **nan_rep**: string to use as string nan representation
- **chunksize**: size to chunk the writing
- **expectedrows**: expected TOTAL row size of this table
- **encoding**: default None, provide an encoding for strings
- **dropna**: boolean, 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

---

**34.1.7.4 pandas.HDFStore.get**

```
HDFStore.get(key)
```

Retrieve pandas object stored in file

**Parameters**

- **key**: object

**Returns**

- **obj**: type of object stored in file
34.1.7.5 pandas.HDFStore.select

HDFStore.select(key, where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, auto_close=False, **kwargs)

Retrieve pandas object stored in file, optionally based on where criteria

Parameters

key : object

where : list of Term (or convertable) objects, optional

start : integer (defaults to None), row number to start selection

stop : integer (defaults to None), row number to stop selection

columns : a list of columns that if not None, will limit the return columns

iterator : boolean, return an iterator, default False

chunksize : nrows to include in iteration, return an iterator

auto_close : boolean, should automatically close the store when finished, default is False

Returns

The selected object

34.1.7.6 pandas.HDFStore.info

HDFStore.info()

print detailed information on the store

New in version 0.21.0.

34.1.8 Feather

read_feather(path[, nthreads])

Load a feather-format object from the file path

34.1.8.1 pandas.read_feather

pandas.read_feather(path, nthreads=1)

Load a feather-format object from the file path

Parameters

path : string file path, or file-like object

nthreads : int, default 1

Number of CPU threads to use when reading to pandas.DataFrame

Returns

type of object stored in file

34.1.9 Parquet

read_parquet(path[, engine])

Load a parquet object from the file path, returning a DataFrame.
34.1.9.1 pandas.read_parquet

```
pandas.read_parquet(path, engine='auto', **kwargs)

Load a parquet object from the file path, returning a DataFrame.
```

**Parameters**

- **path** : string
  File path
- **engine** : {'auto', 'pyarrow', 'fastparquet'}, default 'auto'
  Parquet reader library to use. If 'auto', then the option `io.parquet.engine` is used. If 'auto', then the first library to be installed is used.
- **kwargs** are passed to the engine

**Returns**

DataFrame

34.1.10 SAS

```
read_sas(filepath_or_buffer[, format, ...])

Read SAS files stored as either XPORT or SAS7BDAT format files.
```

34.1.10.1 pandas.read_sas

```
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** : string or file-like object
  Path to the SAS file.
- **format** : string {'xport', 'sas7bdat'} or None
  If None, file format is inferred. 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** : string, 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.
- **iterator** : bool, defaults to False
  If True, returns an iterator for reading the file incrementally.

**Returns**

DataFrame if iterator=False and chunksize=None, else SAS7BDATReader or XportReader

34.1.11 SQL

```
read_sql_table(table_name, con[, schema, ...])

Read SQL database table into a DataFrame.
```

Continued on next page
Table 34.12 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_sql_query</code></td>
<td>Read SQL query into a DataFrame.</td>
</tr>
<tr>
<td><code>read_sql</code></td>
<td>Read SQL query or database table into a DataFrame.</td>
</tr>
</tbody>
</table>

### 34.1.12 Google BigQuery

`read_gbq`(`query`, `project_id=None`, `index_col=None`, `col_order=None`, `reauth=False`, `verbose=True`, `private_key=None`, `dialect='legacy'`, **kwargs)

Load data from Google BigQuery.

The main method a user calls to execute a Query in Google BigQuery and read results into a pandas DataFrame. Google BigQuery API Client Library v2 for Python is used. Documentation is available [here](#).

Authentication to the Google BigQuery service is via OAuth 2.0.

- If “private_key” is not provided:
  - By default “application default credentials” are used.
  - If default application credentials are not found or are restrictive, user account credentials are used. In this case, you will be asked to grant permissions for product name ‘pandas GBQ’.
- If “private_key” is provided:
  - Service account credentials will be used to authenticate.

**Parameters**

- **`query`**: str
  - SQL-Like Query to return data values
- **`project_id`**: str
  - Google BigQuery Account project ID.
- **`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`**: boolean (default False)
  - Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.
- **`verbose`**: boolean (default True)
  - Verbos output
- **`private_key`**: str (optional)
  - Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (e.g. jupyter iPython notebook on remote host)
- **`dialect`**: (‘legacy’, ‘standard’), default ‘legacy’
‘legacy’ : Use BigQuery’s legacy SQL dialect. ‘standard’ : Use BigQuery’s standard SQL (beta), which is compliant with the SQL 2011 standard. For more information see BigQuery SQL Reference

**kwargs : Arbitrary keyword arguments

customization (dict): query config parameters for job processing. For example:

```python
configuration = {'query': {'useQueryCache': False}}
```

For more information see BigQuery SQL Reference

Returns df: DataFrame

DataFrame representing results of query

### 34.1.13 STATA

**read_stata**(filepath_or_buffer[, ...]) Read Stata file into DataFrame

#### 34.1.13.1 pandas.read_stata

**pandas.read_stata** (filepath_or_buffer, convert_dates=True, convert_categoricals=True, encoding=None, index_col=None, convert_missing=False, preserve_dtypes=True, columns=None, order_categoricals=True, chunksize=None, iterator=False)

Read Stata file into DataFrame

**Parameters**

filepath_or_buffer : string or file-like object

Path to .dta file or object implementing a binary read() functions

convert_dates : boolean, defaults to True

Convert date variables to DataFrame time values

convert_categoricals : boolean, defaults to True

Read value labels and convert columns to Categorical/Factor variables

encoding : string, None or encoding

Encoding used to parse the files. None defaults to latin-1.

index_col : string, optional, default: None

Column to set as index

convert_missing : boolean, defaults to 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 : boolean, defaults to 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**: boolean, defaults to 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**: boolean, default False
Return StataReader object

**Returns** DataFrame or StataReader

**Examples**

Read a Stata dta file:

```python
>>> df = pandas.read_stata('filename.dta')
```

Read a Stata dta file in 10,000 line chunks:

```python
>>> itr = pandas.read_stata('filename.dta', chunksize=10000)
>>> for chunk in itr:
...    do_something(chunk)
```

<table>
<thead>
<tr>
<th>StataReader.data(**kwargs)</th>
<th>Reads observations from Stata file, converting them into a dataframe</th>
</tr>
</thead>
<tbody>
<tr>
<td>StataReader.data_label()</td>
<td>Returns data label of Stata file</td>
</tr>
<tr>
<td>StataReader.value_labels()</td>
<td>Returns a dict, associating each variable name a dict, associating</td>
</tr>
<tr>
<td>StataReader.variable_labels()</td>
<td>Returns variable labels as a dict, associating each variable name</td>
</tr>
</tbody>
</table>

### 34.1.13.2 pandas.io.stata.StataReader.data

StataReader.data(**kwargs)
Reads observations from Stata file, converting them into a dataframe

Depreciated since version This: is a legacy method. Use read in new code.

**Parameters**

**convert_dates**: boolean, defaults to True
Convert date variables to DataFrame time values

**convert_categoricals**: boolean, defaults to True
Read value labels and convert columns to Categorical/Factor variables

**index_col**: string, optional, default: None
Column to set as index

**convert_missing**: boolean, defaults to 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 re-
turned with object data types and missing values are represented by StataMissingValue objects.

**preserve_dtypes** : boolean, defaults to 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** : boolean, defaults to True

Flag indicating whether converted categorical data are ordered.

**Returns** DataFrame

### 34.1.13.3 pandas.io.stata.StataReader.data_label

StataReader.data_label()

Returns data label of Stata file

### 34.1.13.4 pandas.io.stata.StataReader.value_labels

StataReader.value_labels()

Returns a dict, associating each variable name a dict, associating each value its corresponding label

### 34.1.13.5 pandas.io.stata.StataReader.variable_labels

StataReader.variable_labels()

Returns variable labels as a dict, associating each variable name with corresponding label

### 34.1.13.6 pandas.io.stata.StataWriter.write_file

StataWriter.write_file()

### 34.2 General functions

#### 34.2.1 Data manipulations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>melt(frame[, id_vars, value_vars, var_name, ...])</code></td>
<td>“Unpivots” a DataFrame from wide format to long format, optionally</td>
</tr>
<tr>
<td><code>pivot(index, columns, values)</code></td>
<td>Produce ‘pivot’ table based on 3 columns of this DataFrame.</td>
</tr>
<tr>
<td><code>pivot_table(data[, values, index, columns, ...])</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
</tr>
<tr>
<td><code>crosstab(index, columns[, values, rownames, ...])</code></td>
<td>Compute a simple cross-tabulation of two (or more) factors.</td>
</tr>
<tr>
<td><code>cut(x, bins[, right, labels, retbins, ...])</code></td>
<td>Return indices of half-open bins to which each value of x belongs.</td>
</tr>
</tbody>
</table>
### Table 34.16 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qcut(x, q[, labels, retbins, precision, ...])</code></td>
<td>Quantile-based discretization function.</td>
</tr>
<tr>
<td><code>merge(left, right[, how, on, left_on, ...])</code></td>
<td>Merge DataFrame objects by performing a database-style join operation by columns or indexes.</td>
</tr>
<tr>
<td><code>merge_ordered(left, right[, on, left_on, ...])</code></td>
<td>Perform merge with optional filling/interpolation designed for ordered data like time series data.</td>
</tr>
<tr>
<td><code>merge_asof(left, right[, on, left_on, ...])</code></td>
<td>Perform an asof merge.</td>
</tr>
<tr>
<td><code>concat(objs[, axis, join, join_axes, ...])</code></td>
<td>Concatenate pandas objects along a particular axis with optional set logic along the other axes.</td>
</tr>
<tr>
<td><code>get_dummies(data[, prefix, prefix_sep, ...])</code></td>
<td>Convert categorical variable into dummy/indicator variables</td>
</tr>
<tr>
<td><code>factorize(values[, sort, order, ...])</code></td>
<td>Encode input values as an enumerated type or categorical variable</td>
</tr>
<tr>
<td><code>unique(values)</code></td>
<td>Hash table-based unique.</td>
</tr>
<tr>
<td><code>wide_to_long(df, stubnames, i, j[, sep, suffix])</code></td>
<td>Wide panel to long format.</td>
</tr>
</tbody>
</table>

#### 34.2.1.1 pandas.melt

The `pandas.melt` function unpivots a DataFrame from wide format to long format, optionally leaving identifier variables 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**

- **frame**: DataFrame
  - `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 string, optional
    - If columns are a MultiIndex then use this level to melt.

**Examples**

```python
generate_examples(pandas.melt, [frame, id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None])
```

**See also:**

- `DataFrame.melt`, `pivot_table`, `DataFrame.pivot`
```python
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6

>>> pd.melt(df, 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'], value_vars=['B', 'C'])
   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')
   A       myVarname  myValname
0  a         B     1
1  b         B     3
2  c         B     5
```

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
```
### 34.2.1.2 pandas.pivot

**pandas.pivot** *(index, columns, values)*  
Produce ‘pivot’ table based on 3 columns of this DataFrame. Uses unique values from index / columns and fills with values.

**Parameters**

- **index**: ndarray
  - Labels to use to make new frame’s index

- **columns**: ndarray
  - Labels to use to make new frame’s columns

- **values**: ndarray
  - Values to use for populating new frame’s values

**Returns**

- DataFrame

**See also:**

- `DataFrame.pivot_table`: generalization of pivot that can handle duplicate values for one index/column pair

**Notes**

- Obviously, all 3 of the input arguments must have the same length

### 34.2.1.3 pandas.pivot_table

**pandas.pivot_table** *(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')*

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 or list of functions, 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)

- **fill_value**: scalar, default `None`
  - Value to replace missing values with

- **margins**: boolean, default `False`
Add all row / columns (e.g. for subtotal / grand totals)

**dropna**: boolean, default True

Do not include columns whose entries are all NaN

**margins_name**: string, default ‘All’

Name of the row / column that will contain the totals when margins is True.

**Returns**

```
DataFrame
```

See also:

```
DataFrame.pivot
```

pivot without aggregation that can handle non-numeric data

**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', ...
... 'small', 'large', 'small', 'large'], ...
... 'D': [1, 2, 2, 3, 3, 4, 5, 6, 7]})
```

```python
>>> df
   A   B   C   D
0 foo one small 1
1 foo one large 2
2 foo one large 2
3 foo two small 3
4 foo two small 3
5 bar one large 4
6 bar one small 5
7 bar two small 6
8 bar two large 7
```

```python
>>> table = pivot_table(df, values='D', index=['A', 'B'], ...
... columns=['C'], aggfunc=np.sum)
```

```python
>>> table
   C
A   B
bar one 4.0 5.0
   two 7.0 6.0
foo one 4.0 1.0
   two NaN 6.0
```

34.2.1.4 pandas.crosstab

```
pandas.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)
```

Compute 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.

**aggfunc**: function, optional

If specified, requires *values* be specified as well

**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

**margins**: boolean, default False

Add row/column margins (subtotals)

**margins_name**: string, default ‘All’

Name of the row / column that will contain the totals when margins is True.

New in version 0.21.0.

**dropna**: boolean, default True

Do not include columns whose entries are all NaN

**normalize**: boolean, {'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.

New in version 0.18.1.

**Returns** crosstab : 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.
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", "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'])

   b   one  two
c  dull  shiny  dull  shiny
a
bar  1  2  1  0
foo  2  2  1  2
```

```python
>>> foo = pd.Categorical(['a', 'b'])
>>> bar = pd.Categorical(['d', 'e'])

>>> pd.crosstab(foo, bar)

# 'c' and 'f' are not represented in the data,
# but they still will be counted in the output

   col_0  d  e  f
row_0
a   1  0  0
b   0  1  0
```

### 34.2.1.5 pandas.cut

```
 34.2.1.5 pandas.cut

pandas.cut (x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False)
```

Return indices of half-open bins to which each value of `x` belongs.

**Parameters**

- **x**: array-like
  - Input array to be binned. It has to be 1-dimensional.

- **bins**: int, sequence of scalars, or IntervalIndex
  - If `bins` is an int, it defines the number of equal-width bins in the range of `x`. However, in this case, the range of `x` is extended by .1% on each side to include the min or max values of `x`. If `bins` is a sequence it defines the bin edges allowing for non-uniform bin width. No extension of the range of `x` is done in this case.

- **right**: bool, optional
  - Indicates whether the bins include the rightmost edge or not. If `right == True` (the default), then the bins [1,2,3,4] indicate (1,2], (2,3], (3,4].

- **labels**: array or boolean, 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.

- **retbins**: bool, optional
  - Whether to return the bins 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

**include_lowest** : bool, optional
Whether the first interval should be left-inclusive or not.

**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**

The *cut* function can be useful for going from a continuous variable to a categorical variable. For example, *cut* could convert ages to groups of age ranges.

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Categorical object

**Examples**

```python
>>> pd.cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, retbins=True)
...
([0.19, 3.367], [0.19, 3.367], [0.19, 3.367], [3.367, 6.533], ...
Categories (3, interval[float64]): [0.19, 3.367] < [3.367, 6.533] ...

>>> pd.cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, labels=['good', 'medium', 'bad'])
...
[good, good, good, medium, bad, good]
Categories (3, object): [good < medium < bad]

>>> pd.cut(np.ones(5), 4, labels=False)
array([1, 1, 1, 1, 1])
```

### 34.2.1.6 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**: ndarray or Series
- **q**: integer or array of quantiles
  - 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 boolean, 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.

**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.

New in version 0.20.0.

**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, 1], (1, 2], (2, 3], (3, 4]
Categories (4, interval[<float64>): [0 < 1 < 2 < 3 < 4] ...
```

```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, 1, 2, 3])
```

### 34.2.1.7 pandas.merge

**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 objects by performing a database-style join operation by 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.
Parameters

left : DataFrame
right : DataFrame

how : {'left', 'right', 'outer', 'inner'}, default 'inner'
- 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

on : label or list
Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

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_index : boolean, 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 : boolean, default False
Use the index from the right DataFrame as the join key. Same caveats as left_index

sort : boolean, 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 : 2-length sequence (tuple, list, ...)
Suffix to apply to overlapping column names in the left and right side, respectively

copy : boolean, default True
If False, do not copy data unnecessarily

indicator : boolean or string, default False
If True, adds a column to output DataFrame called “_merge” with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of “left_only” for observations whose merge key only appears in ‘left’ DataFrame, “right_only” for observations whose merge key only appears in ‘right’ DataFrame, and “both” if the observation’s merge key is found in both.

New in version 0.17.0.

validate : string, default None
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.

New in version 0.21.0.

Returns merged : DataFrame

The output type will be the same as ‘left’, if it is a subclass of DataFrame.

See also:
merge_ordered, merge_asof

Examples

```python
>>> A
lkey value
0 foo 1
1 bar 2
2 baz 3
3 foo 4

>>> B
rkey value
0 foo 5
1 bar 6
2 qux 7
3 bar 8

>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
lkey value_x rkey value_y
0 foo 1 foo 5
1 foo 4 foo 5
2 bar 2 bar 6
3 bar 2 bar 8
4 baz 3 NaN NaN
5 NaN NaN qux 7
```

34.2.1.8 pandas.merge_ordered

pandas.merge_ordered(left, right, on=None, left_on=None, right_on=None, left_by=None, right_by=None, fill_method=None, suffixes=('_x', '_y'), how='outer')

Perform merge 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 : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

how : {'left', 'right', 'outer', 'inner'}, default 'outer'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

New in version 0.19.0.

Returns merged : DataFrame

The output type will the be same as ‘left’, if it is a subclass of DataFrame.

See also:

merge, merge_asof

Examples

```
>>> A
   key  lvalue group
0  a    1     a
1  c    2     a
2  e    2     a
3  a    1     b
4  c    2     b
5  e    3     b

>>> B
   key  rvalue
0  a    b
1  c    c
2  e    d
3  a    b
4  c    b
5  e    b

>>> ordered_merge(A, B, fill_method='ffill', left_by='group')
   key  lvalue group  rvalue
0  a    1     a    NaN
1  b    1     a    1
2  c    2     a    2
3  d    2     a    3
4  e    3     a    3
5  f    3     a    4
6  a    1     b    NaN
7  b    1     b    1
8  c    2     b    2
9  d    2     b    3
10 e    3     b    3
11 f    3     b    4
```
34.2.1.9 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 an asof merge. 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’.

New in version 0.19.0.

Parameters

left : DataFrame

right : DataFrame

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 : boolean

Use the index of the left DataFrame as the join key.

New in version 0.19.2.

right_index : boolean

Use the index of the right DataFrame as the join key.

New in version 0.19.2.

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.

New in version 0.19.2.

right_by : column name
Field names to match on in the right DataFrame.

New in version 0.19.2.

**suffixes**: 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively.

**tolerance**: integer or Timedelta, optional, default None

Select asof tolerance within this range; must be compatible with the merge index.

**allow_exact_matches**: boolean, 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.

New in version 0.20.0.

**Returns merged**: DataFrame

**See also**:

merge, merge_ordered

**Examples**

```python
def pd.DataFrame(df, **kwargs):
    return DataFrame(df, **kwargs)
```

```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
```

```python
>>> 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
```

```python
>>> pd.merge_asof(left, right, on='a')
  a  left_val  right_val
0  1     a      1
1  5     b      3
2 10    c      7
```

```python
>>> 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.

>>> 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
1    a      1
5    b      3
10   c      7

Here is a real-world times-series example

>>> 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

>>> trades
   time   ticker  price  quantity
0 2016-05-25 13:30:00.023 MSFT  51.95          75
1 2016-05-25 13:30:00.038 MSFT  51.95          155
2 2016-05-25 13:30:00.048 GOOG  720.77         100
3 2016-05-25 13:30:00.048 GOOG  720.92         100
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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>51.97</td>
<td>51.98</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>51.96</td>
<td></td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</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'),
... 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>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2016-05-25</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

### 34.2.1.10 pandas.concat

```python
pandas.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=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, DataFrame, or Panel 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/’index’, 1/’columns’}, default 0`
The axis to concatenate along

**join**: {'inner', 'outer'}, default 'outer'

How to handle indexes on other axis(es)

**join_axes**: list of Index objects

Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic

**ignore_index**: boolean, 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**: 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

Returns **concatenated**: 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, DataFrame.append, DataFrame.join, DataFrame.merge*

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
         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
```

```python
>>> 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
```

---

34.2. General functions
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
>>> pd.concat([df1, df3])
animal letter number
0   NaN     a        1
1   NaN     b        2
0   cat      c        3
1   dog      d        4
```

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`.

```python
>>> 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.

```python
>>> 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']
```

### 34.2.11 pandas.get_dummies

`pandas.get_dummies(data, prefix=None, prefix_sep='__', dummy_na=False, columns=None, sparse=False, drop_first=False)`

Convert categorical variable into dummy/indicator variables

- **Parameters**
  - `data`: array-like, Series, or DataFrame
**prefix** : string, list of strings, or dict of strings, 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** : string, 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 columns should be sparse or not. Returns SparseDataFrame if `data` is a Series or if all columns are included. Otherwise returns a DataFrame with some SparseBlocks.

**drop_first** : bool, default False

Whether to get k-1 dummies out of k categorical levels by removing the first level.

New in version 0.18.0.

**Returns**

---

**dummies** : DataFrame or SparseDataFrame

See also:

* `Series.str.get_dummies`

**Examples**

```python
>>> import pandas as pd
>>> s = pd.Series(list('abca'))

>>> pd.get_dummies(s)
   a  b  c
0 0 1 0
1 1 0 1
2 0 0 1
3 1 0 0

>>> s1 = ['a', 'b', np.nan]

>>> pd.get_dummies(s1)
   a  b
0 0 1
1 1 0
2 0 0
```

34.2. General functions 1269
```python
>>> 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       1       0
1  2       0       1       1       0       0
2  3       1       0       0       0       1

>>> pd.get_dummies(pd.Series(list('abcaa')))
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0

>>> pd.get_dummies(pd.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

### 34.2.1.12 pandas.factorize

```python
pandas.factorize(values, sort=False, order=None, na_sentinel=-1, size_hint=None)
```

Encode input values as an enumerated type or categorical variable

**Parameters**

- **values**: ndarray (1-d)
  - Sequence
  - **sort**: boolean, default False
    - Sort by values
  - **na_sentinel**: int, default -1
    - Value to mark “not found”
  - **size_hint**: hint to the hashtable sizer

**Returns**

- **labels**: the indexer to the original array
  - **uniques**: ndarray (1-d) or Index
    - the unique values. Index is returned when passed values is Index or Series
  - note: an array of Periods will ignore sort as it returns an always sorted PeriodIndex
34.2.13  pandas.unique

**pandas.unique(values)**
Hash table-based unique. Uniques are returned in order of appearance. This does NOT sort.

Significantly faster than numpy.unique. Includes NA values.

**Parameters** values : 1d array-like

**Returns** unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

**See also:**

*pandas.Index.unique*, *pandas.Series.unique*

**Examples**

```python
>>> pd.unique(pd.Series([2, 1, 3, 3]))
array([2, 1, 3])
```

```python
>>> pd.unique(pd.Series([2] + [1] * 5))
array([2, 1])
```

```python
>>> 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')])))
array([Timestamp('2016-01-01 00:00:00-05:00', tz='US/Eastern')],
dtype=object)
```

```python
>>> 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)
```

```python
>>> pd.unique(list('baabc'))
array(['b', 'a', 'c'], dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```python
>>> pd.unique(Series(pd.Categorical(list('baabc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

```python
>>> pd.unique(Series(pd.Categorical(list('baabc'),
                             ... categories=list('abc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```
An ordered Categorical preserves the category ordering.

```python
>>> pd.unique(Series(pd.Categorical(list('baabc'),
...                          categories=list('abc'),
...                          ordered=True)))
[b, a, c]
```

Categories (3, object): [a < b < c]

An array of tuples

```python
>>> pd.unique([('a', 'b'), ('b', 'a'), ('a', 'c'), ('b', 'a')])
array([(a', 'b'), ('b', 'a'), ('a', 'c')], dtype=object)
```

### 34.2.14 pandas.wide_to_long

**pandas.wide_to_long**

```python
pandas.wide_to_long(df, stubnames, i, j, sep='', suffix='\d+')
```

Wide panel 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 subobservation 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 hypen by specifying `sep='-'`
  - New in version 0.20.0.
- **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 Aone, Btwo,..., and you have an unrelated column Arating, you can ignore the last one by specifying `suffix='(?!onetwo)'`
  - New in version 0.20.0.

#### Returns

DataFrame
A DataFrame that contains each stub name as a variable, with new index (i, j)

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 typicaly case.

Examples

```python
>>> import pandas as pd
>>> import numpy as np
>>> 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")
    id  A  B
0   1970  a  2.5
1   1970  b  1.2
2   1970  c  0.7
3   1980  d  3.2
4   1980  e  1.3
5   1980  f  0.1
With multiple id columns

>>> 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, 2.0, 2.1, 2.4, 2.1, 2.3, 2.9],
... 'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
  birth  famid  ht1  ht2
0     1     1     2.8  3.4
1     2     1     2.9  3.8
2     3     1     2.2  2.9
3     1     2     2.0  3.2
4     2     1     2.2  2.9
5     3     2     1.9  2.4
6     1     3     2.2  3.3
7     2     3     2.3  3.4
8     3     3     2.1  2.9
>>> l = pd.wide_to_long(df, stubnames="ht", i=['famid', 'birth'], j='age')
```
Going from long back to wide just takes some creative use of *unstack*:

```python
>>> w = l.reset_index().set_index(['famid', 'birth', 'age']).unstack()
>>> w.columns = pd.Index(w.columns).str.join('')
>>> w.reset_index()

<table>
<thead>
<tr>
<th>famid</th>
<th>birth</th>
<th>ht1</th>
<th>ht2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2.8</td>
<td>3.4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2.9</td>
<td>3.8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.9</td>
<td>3.8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.8</td>
<td>2.8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1.9</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2.2</td>
<td>2.4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.8</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2.3</td>
<td>3.3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2.1</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Less wieldy column names are also handled:

```python
>>> np.random.seed(0)

>>> df = pd.DataFrame({'A(quarterly)-2010': np.random.rand(3),
                     'A(quarterly)-2011': np.random.rand(3),
                     'B(quarterly)-2010': np.random.rand(3),
                     'B(quarterly)-2011': np.random.rand(3),
                     'X': np.random.randint(3, size=3)})

>>> df['id'] = df.index
```
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(quarterly)', 'B(quarterly)']
```

### 34.2.2 Top-level missing data

<table>
<thead>
<tr>
<th>isna(obj)</th>
<th>Detect missing values (NaN in numeric arrays, None/NaN in object arrays)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isnull(obj)</td>
<td>Detect missing values (NaN in numeric arrays, None/NaN in object arrays)</td>
</tr>
<tr>
<td>notna(obj)</td>
<td>Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.</td>
</tr>
<tr>
<td>notnull(obj)</td>
<td>Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.</td>
</tr>
</tbody>
</table>

#### 34.2.2.1 pandas.isna

`pandas.isna(obj)`

Detect missing values (NaN in numeric arrays, None/NaN in object arrays)

**Parameters**

- `arr`: ndarray or object value

Object to check for null-ness

**Returns**

- `isna`: array-like of bool or bool

Array or bool indicating whether an object is null or if an array is given which of the element is null.

**See also:**

- `pandas.notna` boolean inverse of pandas.isna
- `pandas.isnull` alias of isna
34.2.2.2 pandas.isnull

```
pandas.isnull(obj)
```
Detect missing values (NaN in numeric arrays, None/NaN in object arrays)

**Parameters**

- `arr`: ndarray or object value
  
  Object to check for null-ness

**Returns**

- `isna`: array-like of bool or bool
  
  Array or bool indicating whether an object is null or if an array is given which of the element is null.

**See also:**

- `pandas.notna` boolean inverse of pandas.isna
- `pandas.isnull` alias of isnull

34.2.2.3 pandas.notna

```
pandas.notna(obj)
```
Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

**Parameters**

- `arr`: ndarray or object value
  
  Object to check for not-null-ness

**Returns**

- `notna`: array-like of bool or bool
  
  Array or bool indicating whether an object is not null or if an array is given which of the element is not null.

**See also:**

- `pandas.isna` boolean inverse of pandas.notna
- `pandas.notnull` alias of notna

34.2.2.4 pandas.notnull

```
pandas.notnull(obj)
```
Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

**Parameters**

- `arr`: ndarray or object value
  
  Object to check for not-null-ness

**Returns**

- `notnull`: array-like of bool or bool
  
  Array or bool indicating whether an object is not null or if an array is given which of the element is not null.

**See also:**

- `pandas.isna` boolean inverse of pandas.notna
- `pandas.notnull` alias of notna
34.2.3 Top-level conversions

`to_numeric(arg[, errors, downcast])` Convert argument to a numeric type.

34.2.3.1 pandas.to_numeric

`pandas.to_numeric(arg, errors=’raise’, downcast=None)` Convert argument to a numeric type.

**Parameters**

- `arg`: list, tuple, 1-d array, or Series
- `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`: {‘integer’, ‘signed’, ‘unsigned’, ‘float’}, default None
  - 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:
    - ‘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.

New in version 0.19.0.

**Returns**

- `ret`: numeric if parsing succeeded.
  - Return type depends on input. Series if Series, otherwise ndarray

See also:

- `pandas.DataFrame.astype` Cast argument to a specified dtype.
- `pandas.to_datetime` Convert argument to datetime.
- `pandas.to_timedelta` Convert argument to timedelta.
- `numpy.ndarray.astype` Cast a numpy array to a specified type.

Examples

Take separate series and convert to numeric, coercing when told to
>>> import pandas as pd
>>> 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')
    0    apple
    1    1.0
    2    2
    3    -3
dtype: object
>>> pd.to_numeric(s, errors='coerce')
    0    NaN
    1    1.0
    2    2.0
    3    -3.0
dtype: float64

34.2.4 Top-level dealing with datetimelike

<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, with day (calendar) as the default</td>
</tr>
<tr>
<td><code>bdate_range</code></td>
<td>Return a fixed frequency DatetimeIndex, with business day as the default</td>
</tr>
<tr>
<td><code>period_range</code></td>
<td>Return a fixed frequency PeriodIndex, with day (calendar) as the default</td>
</tr>
<tr>
<td><code>timedelta_range</code></td>
<td>Return a fixed frequency TimedeltaIndex, with day as the default</td>
</tr>
<tr>
<td><code>infer_freq</code></td>
<td>Infer the most likely frequency given the input index.</td>
</tr>
</tbody>
</table>

34.2.4.1 pandas.to_datetime

*pandas.to_datetime* *(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, box=True, format=None, exact=True, unit=None, infer_datetime_format=False, origin='unix')*

Convert argument to datetime.

*Parameters* arg : integer, float, string, datetime, list, tuple, 1-d array, Series
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 : boolean, default False

Specify a date parse order if arg is str or its list-likes. If True, parses dates with the day first, eg 10/11/12 is parsed as 2012-11-10. Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug, based on dateutil behavior).

yearfirst : boolean, default False

Specify a date parse order if arg is str or its list-likes.

- If True parses dates with the year first, eg 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 (this is a known bug, based on dateutil behavior).

utc : boolean, default None

Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well).

box : boolean, default True

- If True returns a DatetimeIndex
- If False returns ndarray of values.

format : string, default None

strftime to parse time, eg "%d/%m/%Y", note that "%f" will parse all the way up to nanoseconds.

exact : boolean, True by default

- If True, require an exact format match.
- If False, allow the format to match anywhere in the target string.

unit : string, default ‘ns’

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 units='ms' and origin='unix' (the default), this would calculate the number of milliseconds to the unix epoch start.

infer_datetime_format : boolean, default False

If True and no format is given, attempt to infer the format of the datetime strings, 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 is ‘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.

Returns

Returns **ret** : datetime if parsing succeeded.

Return type depends on input:

• list-like: DatetimeIndex
• Series: Series of datetime64 dtype
• scalar: Timestamp

In case when it is not possible to return designated types (e.g. when any element of input is before Timestamp.min or after Timestamp.max) return will have datetime.datetime type (or corresponding array/Series).

See also:

**pandas.DataFrame.astype** Cast argument to a specified dtype.

**pandas.to_timedelta** Convert argument to timedelta.

Examples

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]
```

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-parsable 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
```

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

>>> %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'))
0  1960-01-02
1  1960-01-03
2  1960-01-04
```

### 34.2.4.2 pandas.to_timedelta

def pandas.to_timedelta(arg, unit='ns', box=True, errors='raise')

Convert argument to timedelta

**Parameters**

- **arg**: string, timedelta, list, tuple, 1-d array, or Series
- **unit**: unit of the arg (D,h,m,s,ms,us,ns) denote the unit, which is an integer/float number
- **box**: boolean, default True
  - If True returns a Timedelta/TimedeltaIndex of the results
  - If False returns a np.timedelta64 or ndarray of values of dtype timedelta64[ns]
- **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**

- **ret**: timedelta64/arrays of timedelta64 if parsing succeeded

**See also**

- `pandas.DataFrame.astype` Cast argument to a specified dtype.
- `pandas.to_datetime` Convert argument to datetime.
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')
```

```python
>>> pd.to_timedelta('15.5us')
Timedelta('0 days 00:00:00.000015')
```

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 00:00:00.000015', NaT],
               dtype='timedelta64[ns]', freq=None)
```

Converting numbers by specifying the `unit` keyword argument:

```python
>>> pd.to_timedelta(np.arange(5), unit='s')
TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02',
                '00:00:03', '00:00:04'],
               dtype='timedelta64[ns]', freq=None)
```

```python
>>> pd.to_timedelta(np.arange(5), unit='d')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

### 34.2.4.3 pandas.date_range

**pandas.date_range**

```
pandas.date_range(start=None, end=None, periods=None, freq='D', tz=None, normalize=False, name=None, closed=None, **kwargs)
```

Return a fixed frequency DatetimeIndex, with day (calendar) as the default frequency

**Parameters**

- **start**: string or datetime-like, default None
  - Left bound for generating dates
- **end**: string or datetime-like, default None
  - Right bound for generating dates
- **periods**: integer, default None
  - Number of periods to generate
- **freq**: string or DateOffset, default ‘D’ (calendar daily)
  - Frequency strings can have multiples, e.g. ‘5H’
- **tz**: string, default None
  - Time zone name for returning localized DatetimeIndex, for example Asia/Hong_Kong
- **normalize**: bool, default False
  - Normalize start/end dates to midnight before generating date range
- **name**: string, default None
  - Name of the resulting DatetimeIndex
- **closed**: string, default None
  -
Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

**Returns**  
rng : DatetimeIndex

**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.

### 34.2.4.4 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=None, **kwargs)

Return a fixed frequency DatetimeIndex, with business day as the default frequency

**Parameters**

- **start**: string or datetime-like, default None  
  Left bound for generating dates

- **end**: string or datetime-like, default None  
  Right bound for generating dates

- **periods**: integer, default None  
  Number of periods to generate

- **freq**: string or DateOffset, default ‘B’ (business daily)  
  Frequency strings can have multiples, e.g. ‘5H’

- **tz**: string 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**: string, default None  
  Name of the resulting DatetimeIndex

- **weekmask**: string 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’

  New in version 0.21.0.

- **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

  New in version 0.21.0.

- **closed**: string, default None  
  Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)
Returns `rng` : DatetimeIndex

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.

34.2.4.5 pandas.period_range

```
pandas.period_range(start=None, end=None, periods=None, freq='D', name=None)
```

Returns a fixed frequency PeriodIndex, with day (calendar) as the default frequency

Parameters `start` : string or period-like, default None
  - Left bound for generating periods

`end` : string or period-like, default None
  - Right bound for generating periods

`periods` : integer, default None
  - Number of periods to generate

`freq` : string or DateOffset, default ‘D’ (calendar daily)
  - Frequency alias

`name` : string, default None
  - Name of the resulting PeriodIndex

Returns `prng` : 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

```
>>> 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]', freq='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.

```
>>> 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]', freq='M')
```
34.2.4.6 pandas.timedelta_range

pandas.timedelta_range(start=None, end=None, periods=None, freq='D', name=None, closed=None)

Return a fixed frequency TimedeltaIndex, with day as the default frequency.

Parameters

start : string or timedelta-like, default None
Left bound for generating timedeltas

date
: string or timedelta-like, default None
Right bound for generating timedeltas

periods : integer, default None
Number of periods to generate

freq : string or DateOffset, default ‘D’ (calendar daily)
Frequency strings can have multiples, e.g. ‘5H’

name : string, default None
Name of the resulting TimedeltaIndex

closed : string, default None
Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

Returns

go : TimedeltaIndex

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.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.

```python
>>> 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')
```
34.2.4.7 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`: boolean, default True

- **Returns**
  - `freq`: string or None
    - None if no discernible frequency
    - TypeError if the index is not datetime-like
    - ValueError if there are less than three values.

34.2.5 Top-level dealing with intervals

interval_range([start, end, periods, freq, ...]) Return a fixed frequency IntervalIndex

34.2.5.1 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`: integer, default None
    - Number of periods to generate
  - `freq`: numeric, string, 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’ (calendar daily) for datetime-like.
  - `name`: string, default None
    - Name of the resulting IntervalIndex
  - `closed`: string, default ‘right’
    - options are: ‘left’, ‘right’, ‘both’, ‘neither’

- **Returns**
  - `rng`: IntervalIndex

See also:

- **IntervalIndex** an Index of intervals that are all closed on the same side.
Notes

Of the three parameters: `start`, `end`, and `periods`, exactly two must be specified.

Examples

Numeric `start` and `end` is supported.

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]
              closed='right', dtype='interval[int64]')
```

Additionally, datetime-like input is also supported.

```
>>> 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]
              closed='right', dtype='interval[datetime64[ns]]')
```

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.

```
>>> 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]
              closed='right', dtype='interval[float64]')
```

Similarly, for datetime-like `start` and `end`, the frequency must be convertible to a DateOffset.

```
>>> 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]
              closed='right', dtype='interval[datetime64[ns]]')
```

The `closed` parameter specifies which endpoints of the individual intervals within the `IntervalIndex` are closed.

```
>>> pd.interval_range(end=5, periods=4, closed='both')
IntervalIndex([(1, 2], [2, 3], [3, 4], [4, 5]
              closed='both', dtype='interval[int64]')
```

34.2.6 Top-level evaluation

```
eval(expr[, parser, engine, truediv, ...])
```
Evaluate a Python expression as a string using various backends.

34.2.6.1 pandas.eval

```
pandas.eval(expr[, parser='pandas', engine=None, truediv=True, 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 or unicode

The expression to evaluate. This string cannot contain any Python statements, only Python expressions.

**parser**: string, default ‘pandas’, {'pandas', 'python'}

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](https://pandas.pydata.org/pandas-docs/stable/optimization.html) documentation for more details.

**engine**: string or None, default ‘numexpr’, {'pandas', 'python'}

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

**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 index and 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

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:
  pandas.DataFrame.query, pandas.DataFrame.eval

Notes

The dtype of any objects involved in an arithmetic % operation are recursively cast to float64.
See the enhancing performance documentation for more details.

34.2.7 Testing

test([extra_args])

34.2.7.1 pandas.test

pandas.test(extra_args=None)

34.3 Series

34.3.1 Constructor

Series([data, index, dtype, name, copy, ...])

One-dimensional ndarray with axis labels (including time series).

34.3.1.1 pandas.Series

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, dict, or scalar value
  
  Contains data stored in Series

- **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(len(data))` if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

- **dtype**: `numpy.dtype` or None
  
  If None, dtype will be inferred

- **copy**: bool, default False
  
  Copy input data

**Attributes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td><strong>asobject</strong></td>
<td>return object Series which contains boxed values</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>Return a list of the row axis labels</td>
</tr>
<tr>
<td><strong>base</strong></td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td><strong>blocks</strong></td>
<td>Internal property, property synonym for as_blocks()</td>
</tr>
<tr>
<td><strong>data</strong></td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td><strong>dtypes</strong></td>
<td>return the dtypes object of the underlying data</td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>return the dtypes object of the underlying data</td>
</tr>
<tr>
<td><strong>flags</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ftype</strong></td>
<td>return if the data is sparse</td>
</tr>
<tr>
<td><strong>ftypes</strong></td>
<td>return if the data is sparse</td>
</tr>
<tr>
<td><strong>hasnans</strong></td>
<td></td>
</tr>
<tr>
<td><strong>iat</strong></td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td><strong>iloc</strong></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><strong>imag</strong></td>
<td></td>
</tr>
<tr>
<td><strong>is_copy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>is_monotonic</strong></td>
<td>Return boolean if values in the object are</td>
</tr>
<tr>
<td><strong>is_monotonic_decreasing</strong></td>
<td>Return boolean if values in the object are</td>
</tr>
<tr>
<td><strong>is_monotonic_increasing</strong></td>
<td>Return boolean if values in the object are</td>
</tr>
<tr>
<td><strong>is_unique</strong></td>
<td>Return boolean if values in the object are unique</td>
</tr>
<tr>
<td><strong>itemsize</strong></td>
<td>return the size of the dtypes of the item of the underlying data</td>
</tr>
<tr>
<td><strong>ix</strong></td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.24 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>nbytes</td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td>ndim</td>
<td>return the number of dimensions of the underlying data,</td>
</tr>
<tr>
<td>real</td>
<td></td>
</tr>
<tr>
<td>shape</td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td>size</td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td>strides</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>values</td>
<td>Return Series as ndarray or ndarray-like</td>
</tr>
</tbody>
</table>

**pandas.Series.T**

`Series.T`
- return the transpose, which is by definition self

**pandas.Series.asobject**

`Series.asobject`
- return object Series which contains boxed values
  - *this is an internal non-public method*

**pandas.Series.at**

`Series.at`
- Fast label-based scalar accessor
  - Similarly to `loc`, `at` provides `label` based scalar lookups. You can also set using these indexers.

**pandas.Series.axes**

`Series.axes`
- Return a list of the row axis labels

**pandas.Series.base**

`Series.base`
- return the base object if the memory of the underlying data is shared

**pandas.Series.blocks**

`Series.blocks`
- Internal property, property synonym for `as_blocks()`
  - Deprecated since version 0.21.0.
pandas.Series.data

Series.data
return the data pointer of the underlying data

pandas.Series.dtype

Series.dtype
return the dtype object of the underlying data

pandas.Series.dtypes

Series.dtypes
return the dtype object of the underlying data

pandas.Series.empty

Series.empty

pandas.Series.flags

Series.flags

pandas.Series.ftype

Series.ftype
return if the data is sparse|dense

pandas.Series.ftypes

Series.ftypes
return if the data is sparse|dense

pandas.Series.hasnans

Series.hasnans = None

pandas.Series.iat

Series.iat
Fast integer location scalar accessor.
Similarly to iloc, iat provides integer based lookups. You can also set using these indexers.
pandas.Series.iloc

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, DataFrame or Panel) and that returns valid output for indexing (one of the above).

.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

pandas.Series.imag

Series.imag

pandas.Series.is_copy

Series.is_copy = None

pandas.Series.is_monotonic

Series.is_monotonic
Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns is_monotonic : boolean

pandas.Series.is_monotonic_decreasing

Series.is_monotonic_decreasing
Return boolean if values in the object are monotonic_decreasing

New in version 0.19.0.

Returns is_monotonic_decreasing : boolean
pandas.Series.is_monotonic_increasing

Series.is_monotonic_increasing
Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns is_monotonic : boolean

pandas.Series.is_unique

Series.is_unique
Return boolean if values in the object are unique

Returns is_unique : boolean

pandas.Series.itemsize

Series.itemsize
return the size of the dtype of the item of the underlying data

pandas.Series.ix

Series.ix
A primarily label-location based indexer, with integer position fallback.

.ix[] supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

.ix is the most general indexer and will support any of the inputs in .loc and .iloc. .ix also supports floating point label schemes. .ix is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it’s usually better to be explicit and use .iloc or .loc.

See more at Advanced Indexing.

pandas.Series.loc

Series.loc
Purely label-location based indexer for selection by label.

.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'].

• A slice object with labels, e.g. 'a'::'f' (note that contrary to usual python slices, both the start and the stop are included!).

• A boolean array.
• A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above).

.loc will raise a KeyError when the items are not found.

See more at Selection by Label

**pandas.Series.name**

Series.name

**pandas.Series.nbytes**

Series.nbytes

return the number of bytes in the underlying data

**pandas.Series.ndim**

Series.ndim

return the number of dimensions of the underlying data, by definition 1

**pandas.Series.real**

Series.real

**pandas.Series.shape**

Series.shape

return a tuple of the shape of the underlying data

**pandas.Series.size**

Series.size

return the number of elements in the underlying data

**pandas.Series.strides**

Series.strides

return the strides of the underlying data

**pandas.Series.values**

Series.values

Return Series as ndarray or ndarray-like depending on the dtype

Returns arr : numpy.ndarray or ndarray-like
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='datetime64[ns]')
```

Methods

- `abs()`: Return an object with absolute value taken—only applicable to objects that are all numeric.
- `add(other[, level, fill_value, axis])`: Addition of series and other, element-wise (binary operator `add`).
- `add_prefix(prefix)`: Concatenate prefix string with panel items names.
- `add_suffix(suffix)`: Concatenate suffix string with panel items names.
- `agg(func[, axis])`: Aggregate using callable, string, dict, or list of string/callables.
- `aggregate(func[, axis])`: Aggregate using callable, string, dict, or list of string/callables.
- `align(other[, join, axis, level, copy, ...])`: Align two objects on their axes with the
- `all([axis, bool_only, skipna, level])`: Return whether all elements are True over requested axis.
- `any([axis, bool_only, skipna, level])`: Return whether any element is True over requested axis.
- `append(to_append[, ignore_index, ...])`: Concatenate two or more Series.
- `apply(func[, convert_dtype, args])`: Invoke function on values of Series.
- `argmax(*args, **kwargs)`: 
- `argmin(*args, **kwargs)`: 
- `argsort([axis, kind, order])`: Overrides ndarray.argsort.
- `as_blocks([copy])`: Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
- `as_matrix([columns])`: Convert the frame to its Numpy-array representation.
- `asfreq(freq[, method, how, normalize, ...])`: Convert TimeSeries to specified frequency.
- `asof(where[, subset])`: The last row without any NaN is taken (or the last row without
- `astype(dtype[, copy, errors])`: Cast a pandas object to a specified dtype `dtype`.
- `at_time(time[, asof])`: Select values at particular time of day (e.g.
- `autocorr([lag])`: Lag-N autocorrelation

Continued on next page
### Table 34.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 DataFrame.fillna(method='bfill').</td>
</tr>
<tr>
<td><code>bool()</code></td>
<td>Return the bool of a single element PandasObject.</td>
</tr>
<tr>
<td><code>cat</code></td>
<td>alias of CategoricalAccessor.</td>
</tr>
<tr>
<td><code>clip([lower, upper, axis, inplace])</code></td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><code>clip_lower(threshold[, axis, inplace])</code></td>
<td>Return copy of the input with values below given value(s) truncated.</td>
</tr>
<tr>
<td><code>clip_upper(threshold[, axis, inplace])</code></td>
<td>Return copy of input with values above given value(s) truncated.</td>
</tr>
<tr>
<td><code>combine(other, func[, fill_value])</code></td>
<td>Perform elementwise binary operation on two Series using given function.</td>
</tr>
<tr>
<td><code>combine_first(other)</code></td>
<td>Combine Series values, choosing the calling Series’s values first.</td>
</tr>
<tr>
<td><code>compound([axis, skipna, level])</code></td>
<td>Return the compound percentage of the values for the requested axis.</td>
</tr>
<tr>
<td><code>compress(condition, *args, **kwargs)</code></td>
<td>Return selected slices of an array along given axis as a Series.</td>
</tr>
<tr>
<td><code>consolidate([inplace])</code></td>
<td>DEPRECATED: consolidate will be an internal implementation only.</td>
</tr>
<tr>
<td><code>convert_objects([convert_dates, ...])</code></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Make a copy of this objects 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])</code></td>
<td>Compute covariance with Series, excluding missing values.</td>
</tr>
<tr>
<td><code>cummax([axis, skipna])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>cummin([axis, skipna])</code></td>
<td>Return cumulative minimum over requested axis.</td>
</tr>
<tr>
<td><code>cumprod([axis, skipna])</code></td>
<td>Return cumulative product over requested axis.</td>
</tr>
<tr>
<td><code>cumsum([axis, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td><code>describe([percentiles, include, exclude])</code></td>
<td>Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.</td>
</tr>
<tr>
<td><code>diff([periods])</code></td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td><code>div(other[, level, fill_value, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td><code>divide(other[, level, fill_value, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td><code>dot(other)</code></td>
<td>Matrix multiplication with DataFrame or inner-product with Series.</td>
</tr>
<tr>
<td><code>drop([labels, axis, index, columns, level, ...])</code></td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td><code>drop_duplicates([keep, inplace])</code></td>
<td>Return Series with duplicate values removed.</td>
</tr>
<tr>
<td><code>dropna([axis, inplace])</code></td>
<td>Return Series without null values</td>
</tr>
<tr>
<td><code>dt</code></td>
<td>alias of CombinedDatetimelikeProperties</td>
</tr>
<tr>
<td><code>duplicated([keep])</code></td>
<td>Return boolean Series denoting duplicate values.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq(other[, level, fill_value, axis])</td>
<td>Equal to of series and other, element-wise (binary operator eq).</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td>ewm([com, span, halflife, alpha, ...])</td>
<td>Provides exponential weighted functions.</td>
</tr>
<tr>
<td>expanding([min_periods, freq, center, axis])</td>
<td>Provides expanding transformations.</td>
</tr>
<tr>
<td>factorize([sort, na_sentinel])</td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td>ffill([axis, inplace, limit, downcast])</td>
<td>Synonym for DataFrame.fillna(method='ffill').</td>
</tr>
<tr>
<td>fillna([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td>filter([items, like, regex, axis])</td>
<td>Subset rows or columns of dataframe according to labels in the specified index.</td>
</tr>
<tr>
<td>first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>first_valid_index()</td>
<td>Return index for first non-NA/null value.</td>
</tr>
<tr>
<td>floordiv(other[, level, fill_value, axis])</td>
<td>Integer division of series and other, element-wise (binary operator floordiv).</td>
</tr>
<tr>
<td>from_array(arr[, index, name, dtype, copy, ...])</td>
<td>Read CSV file (DEPRECATED, please use pandas.read_csv() instead).</td>
</tr>
<tr>
<td>from_csv(path[, sep, parse_dates, header, ...])</td>
<td>Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.</td>
</tr>
<tr>
<td>get(other[, level, fill_value, axis])</td>
<td>Greater than or equal to of series and other, element-wise (binary operator ge).</td>
</tr>
<tr>
<td>get(key[, default])</td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.).</td>
</tr>
<tr>
<td>get_dtypes()</td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>get_dtypes()</td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td>get_value(label[, takeable])</td>
<td>Quickly retrieve single value at passed index label same as values (but handles sparseness conversions); is a view</td>
</tr>
<tr>
<td>groupby([by, axis, level, as_index, sort, ...])</td>
<td>Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.</td>
</tr>
<tr>
<td>gt(other[, level, fill_value, axis])</td>
<td>Greater than of series and other, element-wise (binary operator gt).</td>
</tr>
<tr>
<td>head([n])</td>
<td>Return the first n rows.</td>
</tr>
<tr>
<td>hist([by, ax, grid, xlabels, xrot, ...])</td>
<td>Draw histogram of the input series using matplotlib.</td>
</tr>
<tr>
<td>idxmax([axis, skipna])</td>
<td>Index label of the first occurrence of maximum of values.</td>
</tr>
<tr>
<td>idxmin([axis, skipna])</td>
<td>Index label of the first occurrence of minimum of values.</td>
</tr>
<tr>
<td>infer_objects()</td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td>interpolate([method, axis, limit, inplace, ...])</td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td>isin(values)</td>
<td>Return a boolean Series showing whether each element in the Series is exactly contained in the passed sequence of values.</td>
</tr>
<tr>
<td>isna()</td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td>isnull()</td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item()</td>
<td>return the first element of the underlying data as a python</td>
</tr>
<tr>
<td>items()</td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
<tr>
<td>iteritems()</td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
<tr>
<td>keys()</td>
<td>Alias for index</td>
</tr>
<tr>
<td>kurt([axis, skipna, level, numeric_only])</td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td>kurtosis([axis, skipna, level, numeric_only])</td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td>last(offset)</td>
<td>Convenience method for subsetting 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/null value.</td>
</tr>
<tr>
<td>le(other[, level, fill_value, axis])</td>
<td>Less than or equal to of series and other, element-wise (binary operator le).</td>
</tr>
<tr>
<td>lt(other[, level, fill_value, axis])</td>
<td>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 for the requested axis</td>
</tr>
<tr>
<td>map(arg[, na_action])</td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td>mask(cond[, other, inplace, axis, level, ...])</td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.</td>
</tr>
<tr>
<td>max([axis, skipna, level, numeric_only])</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>mean([axis, skipna, level, numeric_only])</td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td>median([axis, skipna, level, numeric_only])</td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td>memory_usage([index, deep])</td>
<td>Memory usage of the Series</td>
</tr>
<tr>
<td>min([axis, skipna, level, numeric_only])</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td>mod(other[, level, fill_value, axis])</td>
<td>Modulo of series and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td>mode()</td>
<td>Return the mode(s) of the dataset.</td>
</tr>
<tr>
<td>mul(other[, level, fill_value, axis])</td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td>multiply(other[, level, fill_value, axis])</td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td>ne(other[, level, fill_value, axis])</td>
<td>Not equal to of series and other, element-wise (binary operator ne).</td>
</tr>
<tr>
<td>nlargest([n, keep])</td>
<td>Return the largest n elements.</td>
</tr>
<tr>
<td>nonzero()</td>
<td>Return the indices of the elements that are non-zero</td>
</tr>
<tr>
<td>notna()</td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
<tr>
<td>notnull()</td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
<tr>
<td>nsmallest([n, keep])</td>
<td>Return the smallest n elements.</td>
</tr>
<tr>
<td>nunique([dropna])</td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td>pct_change([periods, fill_method, limit, freq])</td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td>pipe(func, *args, **kwargs)</td>
<td>Apply func(self, *args, **kwargs)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot</code></td>
<td>Alias of <code>SeriesPlotMethods</code></td>
</tr>
<tr>
<td><code>pop(item)</code></td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td><code>pow(other[, level, fill_value, axis])</code></td>
<td>Exponential power of series 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 for the requested axis.</td>
</tr>
<tr>
<td><code>product([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for the requested axis.</td>
</tr>
<tr>
<td><code>ptp([axis, skipna, level, numeric_only])</code></td>
<td>Returns the difference between the maximum value and the minimum value in the object.</td>
</tr>
<tr>
<td><code>put(*args, **kwargs)</code></td>
<td>Applies the put method to its <code>values</code> attribute if it has one.</td>
</tr>
<tr>
<td><code>quantile([q, interpolation])</code></td>
<td>Return value at the given quantile, a la <code>numpy.percentile</code>.</td>
</tr>
<tr>
<td><code>radd(other[, level, fill_value, axis])</code></td>
<td>Addition of series and other, element-wise (binary operator radd).</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>Floating division of series and other, element-wise (binary operator rtruediv).</td>
</tr>
<tr>
<td><code>reindex([index])</code></td>
<td>Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><code>reindex_axis(labels[, axis])</code></td>
<td>for compatibility with higher dims</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit, ...])</code></td>
<td>Return an object with matching indices to myself.</td>
</tr>
<tr>
<td><code>rename([index])</code></td>
<td>Alter Series index labels or name</td>
</tr>
<tr>
<td><code>rename_axis(mapper[, axis, copy, inplace])</code></td>
<td>Alter the name of 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, *args, **kwargs)</code></td>
<td>Repeat elements of an Series.</td>
</tr>
<tr>
<td><code>replace(ilo_replace, value, inplace, limit, ...)</code></td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td><code>resample(rule[, how, axis, fill_method, ...])</code></td>
<td>Convenience method for frequency conversion and resampling of time series.</td>
</tr>
<tr>
<td><code>reset_index([level, drop, name, inplace])</code></td>
<td>Analogous to the <code>pandas.DataFrame.reset_index()</code> function, see docstring there.</td>
</tr>
<tr>
<td><code>reshape(*args, **kwargs)</code></td>
<td>Deprecated since version 0.19.0.</td>
</tr>
<tr>
<td><code>rfloordiv(other[, level, fill_value, axis])</code></td>
<td>Integer division of series and other, element-wise (binary operator rfloordiv).</td>
</tr>
<tr>
<td><code>rmod(other[, level, fill_value, axis])</code></td>
<td>Modulo of series and other, element-wise (binary operator rmod).</td>
</tr>
<tr>
<td><code>rmul(other[, level, fill_value, axis])</code></td>
<td>Multiplication of series and other, element-wise (binary operator rmul).</td>
</tr>
<tr>
<td><code>rolling(window[, min_periods, freq, center, ...])</code></td>
<td>Provides 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>Exponential power of series and other, element-wise (binary operator rpow).</td>
</tr>
<tr>
<td><code>rsub(other[, level, fill_value, axis])</code></td>
<td>Subtraction of series and other, element-wise (binary operator rsub).</td>
</tr>
<tr>
<td><code>rtruediv(other[, level, fill_value, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator rtruediv).</td>
</tr>
<tr>
<td><code>sample([n, frac, replace, weights, ...])</code></td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>searchsorted</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>select</code></td>
<td>Return data corresponding to axis labels matching criteria.</td>
</tr>
<tr>
<td><code>sem</code></td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><code>set_axis</code></td>
<td>Assign desired index to given axis.</td>
</tr>
<tr>
<td><code>set_value</code></td>
<td>Quickly set single value at passed label.</td>
</tr>
<tr>
<td><code>shift</code></td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
<tr>
<td><code>skew</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>slice_shift</code></td>
<td>Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>sort_index</code></td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td><code>sort_values</code></td>
<td>Sort by the values along either axis.</td>
</tr>
<tr>
<td><code>squeeze</code></td>
<td>Squeeze length 1 dimensions.</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>StringMethods</code>.</td>
</tr>
<tr>
<td><code>subtract</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>sub</code>).</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 MultiIndex</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>Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.</td>
</tr>
<tr>
<td><code>to_csv</code></td>
<td>Write Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td><code>to_dense</code></td>
<td>Return dense representation of NDFrame (as opposed to sparse)</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 Series 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 HDFStore.</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 an object to a tabular environment table.</td>
</tr>
<tr>
<td><code>to_msgpack</code></td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_period</code></td>
<td>Convert Series from DatetimeIndex to PeriodIndex with desired</td>
</tr>
<tr>
<td><code>to_pickle</code></td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_sparse</code></td>
<td>Convert Series to SparseSeries</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>
</tbody>
</table>
Table 34.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_timestamp([freq, how, 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>tolist()</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>transform(func, *args, **kwags)</code></td>
<td>Call function producing a like-indexed NDFrame</td>
</tr>
<tr>
<td><code>transpose(*args, **kwargs)</code></td>
<td>Return the transpose, which is by definition self</td>
</tr>
<tr>
<td><code>truediv(other[, level, fill_value, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td><code>truncate([before, after, axis, copy])</code></td>
<td>Truncates a sorted DataFrame/Series before and/or after some particular index value.</td>
</tr>
<tr>
<td><code>tshift([periods, freq, axis])</code></td>
<td>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, ambiguous])</code></td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td><code>unstack([level, fill_value])</code></td>
<td>Unstack, a.k.a.</td>
</tr>
<tr>
<td><code>update(other)</code></td>
<td>Modify Series in place using non-NA values from passed Series.</td>
</tr>
<tr>
<td><code>valid([inplace])</code></td>
<td></td>
</tr>
<tr>
<td><code>value_counts([normalize, sort, ascending, ...])</code></td>
<td>Returns object containing counts of unique values.</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>view([dtype])</code></td>
<td></td>
</tr>
<tr>
<td><code>where(cond[, other, inplace, axis, level, ...])</code></td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.</td>
</tr>
<tr>
<td><code>xs(key[, axis, level, drop_level])</code></td>
<td>Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.Series.abs**

```python
def abs(self) -> Series
```

Return an object with absolute value taken–only applicable to objects that are all numeric.

**Returns** abs: type of caller

**pandas.Series.add**

```python
def add(self, other, axis=0, fill_value=None) -> Series
```

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 one of the inputs.

**Parameters**

- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
- **axis**: int or name

**Examples**

- `series.add(other=0)`
- `series.add(fill_value=0)`
- `series.add(level='A')`
pandas.Series.add_prefix

```
Series.add_prefix(prefix)
```

Concatenate prefix string with panel items names.

Parameters

- **prefix**: string

Returns

- **with_prefix**: type of caller

pandas.Series.add_suffix

```
Series.add_suffix(suffix)
```

Concatenate suffix string with panel items names.

Parameters

- **suffix**: string

Returns

- **with_suffix**: type of caller

pandas.Series.agg

```
Series.agg(func, axis=0, *args, **kwargs)
```

Aggregate using callable, string, dict, or list of string/callables

New in version 0.20.0.

Parameters

- **func**: callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

Returns

- **aggregated**: Series

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).
agg is an alias for aggregate. Use the alias.

Examples

```python
>>> s = Series(np.random.randn(10))
```

```python
>>> s.agg('min')
-1.301805
```

```python
>>> s.agg(['min', 'max'])
min -1.301805
max 1.127688
dtype: float64
```

pandas.Series.aggregate

Series.aggregate(func, axis=0, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables
New in version 0.20.0.

Parameters

- **func**: callable, string, dictionary, or list of string/callables
  
  Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.
  
  Accepted Combinations are:
  
  - string function name
  - function
  - list of functions
  - dict of column names -> functions (or list of functions)

Returns

- **aggregated**: Series

See also:

pandas.Series.apply, pandas.Series.transform

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.

Examples

```python
>>> s = Series(np.random.randn(10))
```
```python
>>> s.agg('min')
-1.3018049988556679

>>> s.agg(['min', 'max'])
min  -1.301805
max  1.127688
dtype: float64
```

### 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 for each axis.

**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**: boolean, 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**: str, default `None`  
- **limit**: int, default `None`  
- **fill_axis**: `{0, ‘index’}`, default `0`  
  - Filling axis, method and limit
- **broadcast_axis**: `{0, ‘index’}`, default `None`  
  - Broadcast values along this axis, if aligning two objects of different dimensions

**Returns**
- (left, right): (Series, type of other)  
  - Aligned objects

### pandas.Series.all

`Series.all(axis=None, bool_only=None, skipna=None, level=None, **kwargs)`  
Return whether all elements are True over requested axis.

**Parameters**
- **axis**: {index (0)}
- **skipna**: boolean, 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

**bool_only**: boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns all**: scalar or Series (if level specified)

**pandas.Series.any**

```
Series.any (axis=None, bool_only=None, skipna=None, level=None, **kwargs)
```

Returns whether any element is True over requested axis

**Parameters**

**axis**: {index (0)}

**skipna**: boolean, 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

**bool_only**: boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns any**: scalar or Series (if level specified)

**pandas.Series.append**

```
Series.append (to_append, ignore_index=False, verify_integrity=False)
```

Concatenate two or more Series.

**Parameters**

**to_append**: Series or list/tuple of Series

**ignore_index**: boolean, default False

If True, do not use the index labels.

**verify_integrity**: boolean, default False

If True, raise Exception on creating index with duplicates

**Returns appended**: Series

**See also**:

**pandas.concat** General function to concatenate DataFrame, Series or Panel 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
  4 5
  5 6
dtype: int64

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

Series.apply(func, convert_dtype=True, args=(), **kwds)

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

convert_dtype : boolean, default True
Try to find better dtype for elementwise function results. If False, leave as dtype=object

```python
args : tuple
```

Positional arguments to pass to function in addition to the value

**Additional keyword arguments will be passed as keywords to the function**

```python
Returns y : Series or DataFrame if func returns a Series
```

See also:

- `Series.map` For element-wise operations
- `Series.agg` only perform aggregating type operations
- `Series.transform` only perform transforming type operations

**Examples**

Create a series with typical summer temperatures for each city.

```python
>>> import pandas as pd
>>> import numpy as np

>>> series = pd.Series([20, 21, 12], index=['London', 'New York', 'Helsinki'])

>>> series
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

>>> series.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
>>> series.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

>>> series.apply(subtract_custom_value, args=(5,))
London    15
New York   16
```
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

>>> series.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
>>> series.apply(np.log)
London 2.995732
New York 3.044522
Helsinki 2.484907
dtype: float64
```

**pandas.Series.argmax**

```
Series.argmax(*args, **kwargs)
```

**pandas.Series.argmin**

```
Series.argmin(*args, **kwargs)
```

**pandas.Series.argsort**

```
Series.argsort(axis=0, kind='quicksort', order=None)
```

Overrides ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values.

**Parameters**
- `axis` : int (can only be zero)
- `kind` : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'
- `order` : ignored

**Returns**
- `argsorted` : Series, with -1 indicated where nan values are present

**See also:**
- `numpy.ndarray.argsort`
pandas.Series.as_blocks

Series.as_blocks(copy=True)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters copy : boolean, default True

Returns values : a dict of dtype -> Constructor Types

pandas.Series.as_matrix

Series.as_matrix(columns=None)

Convert the frame to its Numpy-array representation.

Parameters columns: list, optional, default:None

If None, return all columns, otherwise, returns specified columns.

Returns values : ndarray

If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

pandas.DataFrame.values

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

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 upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use ‘.values’.

pandas.Series.asfreq

Series.asfreq(freq, method=None, how=None, normalize=False, fill_value=None)

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. resample is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters freq : DateOffset object, or string

method : {‘backfill’/’bfill’, ‘pad’/’ffill’}, 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).

New in version 0.20.0.

**Returns converted**: type of caller

**See also**:

reindex

**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.
>>> df.asfreq(freq='30S', fill_value=9.0)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01 00:00:00</td>
<td>0.0</td>
</tr>
<tr>
<td>2000-01-01 00:00:30</td>
<td>9.0</td>
</tr>
<tr>
<td>2000-01-01 00:01:00</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-01 00:01:30</td>
<td>9.0</td>
</tr>
<tr>
<td>2000-01-01 00:02:00</td>
<td>2.0</td>
</tr>
<tr>
<td>2000-01-01 00:02:30</td>
<td>9.0</td>
</tr>
<tr>
<td>2000-01-01 00:03:00</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Upsample again, providing a method.

>>> df.asfreq(freq='30S', method='bfill')

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01 00:00:00</td>
<td>0.0</td>
</tr>
<tr>
<td>2000-01-01 00:00:30</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-01 00:01:00</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-01 00:01:30</td>
<td>2.0</td>
</tr>
<tr>
<td>2000-01-01 00:02:00</td>
<td>2.0</td>
</tr>
<tr>
<td>2000-01-01 00:02:30</td>
<td>3.0</td>
</tr>
<tr>
<td>2000-01-01 00:03:00</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**pandas.Series.asof**

Series.asof(where, subset=None)
The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

Parameters where : date or array of dates

    subset : string or list of strings, default None
      if not None use these columns for NaN propagation

Returns where is scalar

    • value or NaN if input is Series
    • Series if input is DataFrame

    where is Index: same shape object as input

See also:

merge_asof

**Notes**

Dates are assumed to be sorted Raises if this is not the case
pandas.Series.astype

Series.astype (dtype, copy=True, errors='raise', **kwargs)
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
New in version 0.20.0.
raise_on_error : raise on invalid input
Deprecated since version 0.20.0: Use errors instead
kwargs : keyword arguments to pass on to the constructor

Returns
casted : type of caller

See also:
pandas.to_datetime Convert argument to datetime.
pandas.to_timedelta Convert argument to timedelta.
pandas.to_numeric Convert argument to a numeric type.
numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

```python
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
>>> ser.astype('category')
0    1
1    2
```

Convert to categorical type:
dtype: category
Categories (2, int64): [1, 2]

Convert to ordered categorical type with custom ordering:

```python
>>> ser.astype('category', ordered=True, categories=[2, 1])
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('int', copy=False)
>>> s2[0] = 10
>>> s1
0 10
1 2
dtype: int64
```

**pandas.Series.at_time**

Series.at_time(time, asof=False)

Select values at particular time of day (e.g. 9:30AM).

- **Parameters**
  - `time`: datetime.time or string

- **Returns**
  - `values_at_time`: type of caller

**pandas.Series.autocorr**

Series.autocorr(lag=1)

Lag-N autocorrelation

- **Parameters**
  - `lag`: int, default 1
    - Number of lags to apply before performing autocorrelation.

- **Returns**
  - `autocorr`: float

**pandas.Series.between**

Series.between(left, right, inclusive=True)

Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

- **Parameters**
  - `left`: scalar
    - Left boundary
  - `right`: scalar
    - Right boundary

- **Returns**
  - `is_between`: Series
pandas.Series.between_time

Series.between_time (start_time, end_time, include_start=True, include_end=True)
Select values between particular times of the day (e.g., 9:00-9:30 AM).

Parameters
start_time : datetime.time or string
end_time : datetime.time or string
include_start : boolean, default True
include_end : boolean, default True

Returns
values_between_time : type of caller

pandas.Series.bfill

Series.bfill (axis=0, inplace=False, limit=None, downcast=None)
Synonym for DataFrame.fillna(method='bfill')

pandas.Series.bool

Series.bool ()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean.

pandas.Series.cat

Series.cat ()
Accessor object for categorical properties of the Series values.

Be aware that assigning to categories is an inplace operation, while all methods return new categorical data per default (but can be called with inplace=True).

Examples

```python
>>> s.cat.categories
>>> s.cat.categories = list('abc')
>>> s.cat.rename_categories(list('cab'))
>>> s.cat.reorder_categories(list('cab'))
>>> s.cat.add_categories(['d','e'])
>>> s.cat.remove_categories(['d'])
>>> s.cat.remove_unused_categories()
>>> s.cat.set_categories(list('abcde'))
>>> s.cat.as_ordered()
>>> s.cat.as_unordered()
```
pandas.Series.clip

Series.clip (lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)
Trim values at input threshold(s).

Parameters:

- lower : float or array_like, default None
- upper : float or array_like, default None
- axis : int or string axis name, optional
  Align object with lower and upper along the given axis.
- inplace : boolean, default False
  Whether to perform the operation in place on the data

New in version 0.21.0.

Returns:
clipped : Series

Examples

```python
>>> df
   0         1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967

>>> df.clip(-1.0, 0.5)
   0         1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000

>>> t
   0    1    2    3    4
0 -0.3 -0.2 -0.1  0.0  0.1
dtype: float64

>>> df.clip(t, t+1, axis=0)
   0         1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```
**pandas.Series.clip_lower**

Series.clip_lower(threshold, axis=None, inplace=False)
Return copy of the input with values below given value(s) truncated.

Parameters threshold : float or array_like
axis : int or string axis name, optional
Align object with threshold along the given axis.
inplace : boolean, default False
Whether to perform the operation in place on the data New in version 0.21.0.
Returns clipped : same type as input
See also:
clip

**pandas.Series.clip_upper**

Series.clip_upper(threshold, axis=None, inplace=False)
Return copy of input with values above given value(s) truncated.

Parameters threshold : float or array_like
axis : int or string axis name, optional
Align object with threshold along the given axis.
inplace : boolean, default False
Whether to perform the operation in place on the data New in version 0.21.0.
Returns clipped : same type as input
See also:
clip

**pandas.Series.combine**

Series.combine(other, func, fill_value=nan)
Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other

Parameters other : Series or scalar value
func : function
fill_value : scalar value
Returns result : Series

**pandas.Series.combine_first**

Series.combine_first(other)
Combine Series values, choosing the calling Series’s values first. Result index will be the union of the two indexes.
Parameters other: Series

Returns y: Series

pandas.Series.compound

Series.compound (axis=None, skipna=None, level=None)
Return the compound percentage of the values for the requested axis

Parameters axis: {index (0)}

skipna: boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only: boolean, 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 compounded: scalar or Series (if level specified)

pandas.Series.compress

Series.compress (condition, *args, **kwargs)
Return selected slices of an array along given axis as a Series

See also:
numpy.ndarray.compress

pandas.Series.consolidate

Series.consolidate (inplace=False)
DEPRECATED: consolidate will be an internal implementation only.

pandas.Series.convert_objects

Series.convert_objects (convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)
Deprecation. Attempt to infer better dtype for object columns

Parameters convert_dates: boolean, default True
If True, convert to date where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.

convert_numeric: boolean, default False
If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas: boolean, default True
If True, convert to timedelta where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.

**copy** : boolean, default True

If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns converted : same as input object

See also:

- **pandas.to_datetime** Convert argument to datetime.
- **pandas.to_timedelta** Convert argument to timedelta.
- **pandas.to_numeric** Return a fixed frequency timedelta index, with day as the default.

**pandas.Series.copy**

*Series.copy*(deep=True)

Make a copy of this objects data.

Parameters **deep** : boolean or string, default True

Make a deep copy, including a copy of the data and the indices. With deep=False neither the indices or the data are copied.

Note that when deep=True data is copied, 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.

Returns copy : type of caller

**pandas.Series.corr**

*Series.corr*(other, method='pearson', min_periods=None)

Compute correlation with other Series, excluding missing values

Parameters **other** : Series

method : {'pearson', ‘kendall’, ‘spearman’}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns correlation : float

**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**  
`nobs` : int or Series (if level specified)

**pandas.Series.cov**

`Series.cov(other, min_periods=None)`  
Compute covariance with Series, excluding missing values  

**Parameters**  
`other` : Series  
`min_periods` : int, optional  
Minimum number of observations needed to have a valid result  

**Returns**  
`covariance` : float  
Normalized by N-1 (unbiased estimator).

**pandas.Series.cummax**

`Series.cummax(axis=None, skipna=True, *args, **kwargs)`  
Return cumulative max over requested axis  

**Parameters**  
`axis` : {index (0)}  
`skipna` : boolean, default True  
Exclude NA/null values. If an entire row/column is NA, the result will be NA  

**Returns**  
`cummax` : scalar  
See also:  
`pandas.core.window.Expanding.max` Similar functionality but ignores NaN values.

**pandas.Series.cummin**

`Series.cummin(axis=None, skipna=True, *args, **kwargs)`  
Return cumulative minimum over requested axis  

**Parameters**  
`axis` : {index (0)}  
`skipna` : boolean, default True  
Exclude NA/null values. If an entire row/column is NA, the result will be NA  

**Returns**  
`cummin` : scalar  
See also:  
`pandas.core.window.Expanding.min` Similar functionality but ignores NaN values.
pandas.Series.cumprod

Series.cumprod(axis=None, skipna=True, *args, **kwargs)
Return cumulative product over requested axis.

Parameters axis : {index (0)}
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns cumprod : scalar

See also:

pandas.core.window.Expanding.prod Similar functionality but ignores NaN values.

pandas.Series.cumsum

Series.cumsum(axis=None, skipna=True, *args, **kwargs)
Return cumulative sum over requested axis.

Parameters axis : {index (0)}
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns cumsum : scalar

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores NaN values.

pandas.Series.describe

Series.describe(percentiles=None, include=None, exclude=None)
Generates descriptive statistics 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(include=['O'])`). To exclude pandas categorical columns, use 'category'

- None (default) : The result will exclude nothing.

**Returns** summary: Series/DataFrame of summary statistics

**See also:**

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

**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  mean  std  min  25%  50%  75%  max
1.000000 3.00 2.00 1.00 1.00 1.50 2.00 2.50 3.00
```

```
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()
count   3
unique  2
top   2010-01-01 00:00:00
freq   2
first  2000-01-01 00:00:00
last   2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({
...    'object': ['a', 'b', 'c'],
...    'numeric': [1, 2, 3],
...    'categorical': pd.Categorical(['d','e','f'])
...})
>>> df.describe()
```

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    c
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=[np.object])
  object
count     3
unique    3
top       c
freq      1
```

Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
  categorical
count      3
unique     3
top        f
top        c
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   c
top     c   f
freq    1   1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.object])
    categorical    numeric
count     3    3.0
```
pandas.Series.diff

Series.diff(periods=1)

1st discrete difference of object

Parameters periods : int, default 1

Periods to shift for forming difference

Returns diffed : Series

pandas.Series.div

Series.div(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters other : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.rtruediv

pandas.Series.divide

Series.divide(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters other : Series or scalar value

fill_value : None or float value, default None (NaN)
Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : Series

**See also:**

*Series.rtruediv*

### pandas.Series.dot

**Series.dot**(other)

Matrix multiplication with DataFrame or inner-product with Series objects

**Parameters**

other : Series or DataFrame

**Returns**

dot_product : scalar or Series

### pandas.Series.drop

**Series.drop**(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

Return new object with labels in requested axis removed.

**Parameters**

labels : single label or list-like

Index or column labels to drop.

axis : int or axis name

Whether to drop labels from the index (0 / ‘index’) or columns (1 / ‘columns’).

index, columns : single label or list-like

Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels).

New in version 0.21.0.

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {‘ignore’, ‘raise’}, default ‘raise’

If ‘ignore’, suppress error and existing labels are dropped.

**Returns**

dropped : type of caller

**Notes**

Specifying both labels and index or columns will raise a ValueError.
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
```

**pandas.Series.drop_duplicates**

Series.drop_duplicates(keep='first', inplace=False)

Return Series 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.

- **inplace** : boolean, default False

If True, performs operation inplace and returns None.

Returns deduplicated : Series

**pandas.Series.dropna**

Series.dropna(axis=0, inplace=False, **kwargs)

Return Series without null values

Returns valid : Series

inplace : boolean, default False
Do operation in place.

**pandas.Series.dt**

Series.dt()

Accessor object for datetimelike properties of the Series values.

**Examples**

```python
generate doctests not needed
```

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')

Return boolean Series denoting duplicate values

Parameters

- **keep**: {'first', 'last', False}, default 'first'
  
  - **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 **duplicated**: Series

**pandas.Series.eq**

Series.eq(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters

- **other**: Series or scalar value
  
  - **fill_value**: None or float value, default None (NaN)
    
    Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  
  - **level**: int or name
    
    Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result**: Series

See also:

Series.None
pandas.Series.equals

Series.equals(other)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

pandas.Series.ewm

Series.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, freq=None, adjust=True, ignore_na=False, axis=0)
Provides exponential weighted functions
New in version 0.18.0.

Parameters
- com : float, optional
  Specify decay in terms of center of mass, \( \alpha = 1/(1 + com) \), for \( com \geq 0 \)
- span : float, optional
  Specify decay in terms of span, \( \alpha = 2/(\text{span} + 1) \), for \( \text{span} \geq 1 \)
- halflife : float, optional
  Specify decay in terms of half-life, \( \alpha = 1 - \exp(\log(0.5)/\text{halflife}) \), for \( \text{halflife} > 0 \)
- alpha : float, optional
  Specify smoothing factor \( \alpha \) directly, \( 0 < \alpha \leq 1 \)
  New in version 0.18.0.
- min_periods : int, default 0
  Minimum number of observations in window required to have a value (otherwise result is NA).
- freq : None or string alias / date offset object, default=None
  Deprecated since version 0.18.0: Frequency to conform to before computing statistic
- adjust : boolean, default True
  Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)
- ignore_na : boolean, default False
  Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns
  a Window sub-classed for the particular operation

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

The freq keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of \texttt{resample()} (i.e. using the mean).
When adjust is True (default), weighted averages are calculated using weights \((1-\alpha)^{n-1}, (1-\alpha)^{n-2}, \ldots, 1-\alpha, 1\).

**When adjust is False, weighted averages are calculated recursively as:**
\[
\text{weighted\_average}[0] = \text{arg}[0];
\text{weighted\_average}[i] = (1-\alpha) \times \text{weighted\_average}[i-1] + \alpha \times \text{arg}[i].
\]

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of \(x\) and \(y\) used in calculating the final weighted average of \([x, None, y]\) are \((1-\alpha)^2\) and \(1\) (if adjust is True), and \((1-\alpha)^2\) and \(\alpha\) if adjust is False.

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of \(x\) and \(y\) used in calculating the final weighted average of \([x, None, y]\) are \(1-\alpha\) and \(1\) (if adjust is True), and \(1-\alpha\) and \(\alpha\) if adjust is False.

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

**Examples**

```python
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
    B
0   0
1   1
2   2
3  NaN
4   4

>>> df.ewm(com=0.5).mean()
        B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

**pandas.Series.expanding**

Series.expanding(min_periods=1, freq=None, center=False, axis=0)

Provides expanding transformations.

New in version 0.18.0.

Parameters

- **min_periods** : int, default None
  Minimum number of observations in window required to have a value (otherwise result is NA).

- **freq** : string or DateOffset object, optional (default None)
  Deprecated since version 0.18.0: Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

- **center** : boolean, default False
  Set the labels at the center of the window.

- **axis** : int or string, default 0

Returns

a Window sub-classed for the particular operation
Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting center=True.

The freq keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of resample() (i.e. using the mean).

Examples

```python
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

>>> df.expanding(2).sum()
   B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

pandas.Series.factorize

Series.factorize(sort=False, na_sentinel=-1)
Encode the object as an enumerated type or categorical variable

Parameters sort : boolean, default False
Sort by values

na_sentinel : int, default -1
Value to mark “not found”

Returns labels : the indexer to the original array
uniques : the unique Index

pandas.Series.ffill

Series.ffill(axis=None, inplace=False, limit=None, downcast=None)
Synonym for DataFrame.fillna(method='ffill')

pandas.Series.fillna

Series.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)
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 / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

- **axis**: {0, 'index'}

- **inplace**: boolean, 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**

- **filled**: Series

**See also:**

- reindex, asfreq

**Examples**

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                     [3, 4, np.nan, 1],
                     [np.nan, np.nan, np.nan, 5],
                     [np.nan, 3, np.nan, 4]],
                    columns=list('ABCD'))
>>> df
   A    B     C    D
0 NaN  2.0  NaN   0
1  3.0  4.0  NaN   1
2 NaN  NaN  NaN   5
3 NaN  3.0  NaN   4
```

Replace all NaN elements with 0s.

```python
>>> df.fillna(0)
   A    B     C    D
0 0.0  2.0  0.0   0
1 3.0  4.0  0.0   1
2 0.0  0.0  0.0   5
3 0.0  3.0  0.0   4
```
We can also propagate non-null values forward or backward.

```python
>>> df.fillna(method='ffill')
     A    B    C    D
0  NaN  2.0  NaN  NaN
1   3.0  4.0  NaN  1.0
2   3.0  4.0  NaN  5.0
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
2  0.0  1.0  2.0  5.0
3  0.0  3.0  2.0  4.0
```

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  5.0
3  NaN  3.0  NaN  4.0
```

**pandas.Series.filter**

`Series.filter(items=None, like=None, regex=None, axis=None)`

Subset rows or columns of dataframe according to labels in the specified index.

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
  - List of info axis to restrict to (must not all be present)
- **like**: string
  - Keep info axis where “arg in col == True”
- **regex**: string (regular expression)
  - Keep info axis with re.search(regex, col) == True
- **axis**: int or string axis name
  - The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame

**Returns**
- same type as input object

See also:
- `pandas.DataFrame.loc`
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
   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)

Convenience method for subsetting initial periods of time series data based on a date offset.

- Parameters offset : string, DateOffset, dateutil.relativedelta
- Returns subset : type of caller

Examples

```python
ts.first('10D') -> First 10 days
```

**pandas.Series.first_valid_index**

Series.first_valid_index()

Return index for first non-NA/null value.

- Returns scalar : type of index
Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series.

**pandas.Series.floordiv**

```python
Series.floordiv(other, level=None, fill_value=None, axis=0)
```

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 one of the inputs.

- **Parameters**
  - `other`: Series or scalar value
  - `fill_value`: None or float value, default None (NaN)
    Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  - `level`: int or name
    Broadcast across a level, matching Index values on the passed MultiIndex level

- **Returns**
  - `result`: Series

See also:

- `Series.rfloordiv`

**pandas.Series.from_array**

```python
classmethod Series.from_array(arr, index=None, name=None, dtype=None, copy=False, fast-path=False)
```

**pandas.Series.from_csv**

```python
classmethod Series.from_csv(path, sep=' ', parse_dates=True, header=None, index_col=0, encoding=None, infer_datetime_format=False)
```

Read CSV file (DEPRECATED, please use `pandas.read_csv()` instead).

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a time Series.

This method only differs from `pandas.read_csv()` in some defaults:

- `index_col` is 0 instead of None (take first column as index by default)
- `header` is None instead of 0 (the first row is not used as the column names)
- `parse_dates` is True instead of False (try parsing the index as datetime by default)

With `pandas.read_csv()`, the option `squeeze=True` can be used to return a Series like `from_csv`.

- **Parameters**
  - `path`: string file path or file handle / StringIO
  - `sep`: string, default `,`
    Field delimiter
parse_dates : boolean, default True

Parse dates. Different default from read_table

header : int, default None

Row to use as header (skip prior rows)

index_col : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

infer_datetime_format : boolean, default False

If True and parse_dates is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns y : Series

See also:

pandas.read_csv

pandas.Series.ge

Series.ge (other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters other : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.None

pandas.Series.get

Series.get (key, default=None)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters key : object
pandas.Series.get_dtype_counts

Series.get_dtype_counts()
Return the counts of dtypes in this object.

pandas.Series.get_ftype_counts

Series.get_ftype_counts()
Return the counts of ftypes in this object.

pandas.Series.get_value

Series.get_value(label, takeable=False)
Quickly retrieve single value at passed index label
Depreciated since version 0.21.0.
Please use .at[] or .iat[] accessors.
Parameters
index : label
    takeable : interpret the index as indexers, default False
Returns
value : scalar value

pandas.Series.get_values

Series.get_values()
same as values (but handles sparseness conversions); is a view

pandas.Series.groupby

Series.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, **kwargs)
Group series using mapper (dict or key function, apply given function to group, return result as series) or
by a series of columns.
Parameters
by : mapping, function, str, or iterable
    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 an ndarray is passed, the values are used as-is determine the groups. A str
    or list of strs may be passed to group by the columns in self
axis : int, default 0
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 : boolean, 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** : boolean, 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** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**Returns** GroupBy object

**Examples**

DataFrame results

```python
>>> data.groupby(func, axis=0).mean()
>>> data.groupby([['col1', 'col2']])['col3'].mean()
```

DataFrame with hierarchical index

```python
>>> data.groupby([['col1', 'col2']]).mean()
```

**pandas.Series.gt**

Series.gt(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

**Parameters** other : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns** result : Series

**See also:**

Series.None
pandas.Series.head

Series.head(n=5)
Return the first n rows.

Parameters n : int, default 5
Number of rows to select.

Returns obj_head : type of caller
The first n rows of the caller object.

pandas.Series.hist

Series.hist(by=None, ax=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, figsize=None, bins=10, **kwds)
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 : boolean, 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 : integer, default 10
Number of histogram bins to be used
kwds : keywords
To be passed to the actual plotting function

Notes

See matplotlib documentation online for more on this
pandas.Series.idxmax

Series.idxmax (axis=None, skipna=True, *args, **kwargs)
Index label of the first occurrence of maximum of values.

Parameters skipna : boolean, default True
Exclude NA/null values

Returns idxmax : Index of maximum of values

See also:
DataFrame.idxmax, numpy.ndarray.argmax

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().

pandas.Series.idxmin

Series.idxmin (axis=None, skipna=True, *args, **kwargs)
Index label of the first occurrence of minimum of values.

Parameters skipna : boolean, default True
Exclude NA/null values

Returns idxmin : Index of minimum of values

See also:
DataFrame.idxmin, numpy.ndarray.argmin

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().

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.

New in version 0.21.0.

Returns converted : same type as input object

See also:
pandas.to_datetime Convert argument to datetime.
pandas.to_timedelta Convert argument to timedelta.
pandas.to_numeric  Convert argument to numeric type

Examples

```python
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
  A
1  1
2  2
3  3

>>> df.dtypes
A    object
dtype: object

>>> df.infer_objects().dtypes
A    int64
dtype: object
```

pandas.Series.interpolate

Series.interpolate( method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', downcast=None, **kwargs)
Interpolate values according to different methods.

Please note that only method='linear' is supported for DataFrames/Series with a MultiIndex.

Parameters method : {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to scipy.interpolate.interp1d. Both 'polynomial' and 'spline' require that you also specify an order (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18
New in version 0.18.1: Added support for the ‘akima’ method. Added interpolate method ‘from_derivatives’ which replaces ‘piecewise_polynomial’ in scipy 0.18; backwards-compatible with scipy < 0.18.

axis: {0, 1}, default 0
- 0: fill column-by-column
- 1: fill row-by-row

limit: int, default None.
   Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction: {‘forward’, ‘backward’, ‘both’}, default ‘forward’
   If limit is specified, consecutive NaNs will be filled in this direction.
   New in version 0.17.0.

inplace: bool, default False
   Update the NDFrame in place if possible.

downcast: optional, ‘infer’ or None, defaults to None
   Downcast dtypes if possible.

kwargs: keyword arguments to pass on to the interpolating function.

Returns: Series or DataFrame of same shape interpolated at the NaNs

See also: reindex, replace, fillna

Examples

Filling in NaNs

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

Series.isin(values)

Return a boolean Series showing whether each element in the Series is exactly contained in the passed sequence of values.

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.

New in version 0.18.1.

Support for values as a set
Returs `isin` : Series (bool dtype)

Raises `TypeError`

- If `values` is a string

See also:

`pandas.DataFrame.isin`

**Examples**

```python
>>> s = pd.Series(list('abc'))
>>> s.isin(['a', 'c', 'e'])
0   True
1  False
2   True
dtype: bool
```

Passing a single string as `s.isin('a')` will raise an error. Use a list of one element instead:

```python
>>> s.isin(['a'])
0   True
1  False
2  False
dtype: bool
```

**pandas.Series.isna**

Series `isna()`

Return a boolean same-sized object indicating if the values are NA.

See also:

- `Series.notna` boolean inverse of isna
- `Series.isnull` alias of isna
- `isna` top-level isna

**pandas.Series.isnull**

Series `isnull()`

Return a boolean same-sized object indicating if the values are NA.

See also:

- `Series.notna` boolean inverse of isna
- `Series.isnull` alias of isna
- `isna` top-level isna
pandas.Series.item

Series.item()
return the first element of the underlying data as a python scalar

pandas.Series.items

Series.items()
Lazily iterate over (index, value) tuples

pandas.Series.iteritems

Series.iteritems()
Lazily iterate over (index, value) tuples

pandas.Series.keys

Series.keys()
Alias for index

pandas.Series.kurt

Series.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters
axis : {index (0)}

skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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
kurt : scalar or Series (if level specified)

pandas.Series.kurtosis

Series.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters
axis : {index (0)}

skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

**numeric_only**: boolean, 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**

- **kurt**: scalar or Series (if level specified)

### pandas.Series.last

**Series.last**(offset)

Convenience method for subsetting final periods of time series data based on a date offset.

**Parameters**

- **offset**: string, DateOffset, dateutil.relativedelta

**Returns**

- **subset**: type of caller

**Examples**

```python
ts.last('5M') -> Last 5 months
```

### pandas.Series.last_valid_index

**Series.last_valid_index**()

Return index for last non-NA/null value.

**Returns**

- **scalar**: type of index

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty Series.

### pandas.Series.le

**Series.le**(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

**Parameters**

- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level
pandas.Series.lt

Series.lt(other, level=None, fill_value=None, axis=0)

Less than of series and other, element-wise (binary operator \( \lt \)).

Equivalent to \( \text{series} < \text{other} \), but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other : Series or scalar value

fill_value : None or float value, default None (NaN)
    Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name
    Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result : Series

See also:

Series.None

pandas.Series.mad

Series.mad(axis=None, skipna=None, level=None)

Return the mean absolute deviation of the values for the requested axis

Parameters

axis : {index (0)}

skipna : boolean, default True
    Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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 mad : scalar or Series (if level specified)

pandas.Series.map

Series.map(arg, na_action=None)

Map values of Series using input correspondence (which can be a dict, Series, or function)
Parameters `arg` : function, dict, or Series

`na_action` : {None, ‘ignore’}

If ‘ignore’, propagate NA values, without passing them to the mapping function

Returns `y` : 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:

```python
>>> from collections import Counter
>>> counter = Counter()
>>> counter['bar'] += 1
>>> y.map(counter)
1 0
2 1
3 0
dtype: int64
```

Examples

Map inputs to outputs (both of type `Series`)

```python
>>> x = pd.Series([1,2,3], index=['one', 'two', 'three'])
>>> x
one    1
two    2
three  3
dtype: int64

>>> y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
>>> y
1  foo
2  bar
3  baz

>>> x.map(y)
one  foo
two  bar
three baz
```

If `arg` is a dictionary, return a new Series with values converted according to the dictionary’s mapping:
```python
>>> z = {1: 'A', 2: 'B', 3: 'C'}

>>> x.map(z)
one    A
two   B
three  C
```

Use `na_action` to control whether NA values are affected by the mapping function.

```python
>>> s = pd.Series([1, 2, 3, np.nan])

>>> s2 = s.map('this is a string {}'.format, na_action=None)
    0    this is a string 1.0
    1    this is a string 2.0
    2    this is a string 3.0
    3    this is a string nan
dtype: object

>>> s3 = s.map('this is a string {}'.format, na_action='ignore')
    0    this is a string 1.0
    1    this is a string 2.0
    2    this is a string 3.0
    3    NaN
dtype: object
```

### pandas.Series.mask

`Series.mask` (cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where `cond` is False and otherwise are from `other`.

**Parameters**

- **cond** : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

  New in version 0.18.1: A callable can be used as cond.

- **other** : scalar, NDFrame, or callable
  
  Entries where `cond` is True are replaced with corresponding value from `other`. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

  New in version 0.18.1: A callable can be used as other.

- **inplace** : boolean, default False
  
  Whether to perform the operation in place on the data

- **axis** : alignment axis if needed, default None

- **level** : alignment level if needed, default None

- **errors** : str, {‘raise’, ‘ignore’}, default ‘raise’
• **raise**: allow exceptions to be raised
• **ignore**: suppress exceptions. On error return original object

Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

**try_cast**: boolean, default False

try to cast the result back to the input type (if possible),

**raise_on_error**: boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

**Returns**  *wh*: same type as caller

See also:

*DataFrame.where()*

**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
generate_examples()
```
```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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.Series.max**

`Series.max(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

This method returns the maximum of the values in the object. If you want the index of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

**Parameters**

- **axis**: {index (0)}
- **skipna**: boolean, default True
  Exclude NA/null values. If an entire row/column is NA or empty, 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
- **numeric_only**: boolean, 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**

- **max**: scalar or Series (if level specified)

**pandas.Series.mean**

`Series.mean(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return the mean of the values for the requested axis

**Parameters**

- **axis**: {index (0)}
- **skipna**: boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

**numeric_only**: boolean, 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**
- **mean**: scalar or Series (if level specified)

**pandas.Series.median**

```
Series.median(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
```

Return the median of the values for the requested axis

**Parameters**
- **axis**: {index (0)}
- **skipna**: boolean, default True
  - Exclude NA/null values. If an entire row/column is NA or empty, 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
- **numeric_only**: boolean, 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**
- **median**: scalar or Series (if level specified)

**pandas.Series.memory_usage**

```
Series.memory_usage(index=True, deep=False)
```

Memory usage of the Series

**Parameters**
- **index**: bool
  - Specifies whether to include memory usage of Series index
- **deep**: bool
  - Introspect the data deeply, interrogate object dtypes for system-level memory consumption

**Returns**
- scalar bytes of memory consumed

**See also**

```
numpy.ndarray.nbytes
```

**Notes**

Memory usage does not include memory consumed by elements that are not components of the array if deep=False
pandas.Series.min

Series.min (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

This method returns the minimum of the values in the object. If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

Parameters

axis : {index (0)}
    skipna : boolean, default True
        Exclude NA/null values. If an entire row/column is NA or empty, 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
    numeric_only : boolean, 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

min : scalar or Series (if level specified)

pandas.Series.mod

Series.mod (other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters

other : Series or scalar value
    fill_value : None or float value, default None (NaN)
        Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
    level : int or name
        Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result : Series

See also:

Series.rmod

pandas.Series.mode

Series.mode ()

Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

Returns

modes : Series (sorted)
pandas.Series.mul

Series.mul(other, level=None, fill_value=None, axis=0)
   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 one of the inputs.

Parameters

other : Series or scalar value

fill_value : None or float value, default None (NaN)
   Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name
   Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result : Series

See also:

Series.rmul

pandas.Series.multiply

Series.multiply(other, level=None, fill_value=None, axis=0)
   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 one of the inputs.

Parameters

other : Series or scalar value

fill_value : None or float value, default None (NaN)
   Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name
   Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result : Series

See also:

Series.rmul

pandas.Series.ne

Series.ne(other, level=None, fill_value=None, axis=0)
   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 one of the inputs.

Parameters

other : Series or scalar value

fill_value : None or float value, default None (NaN)
Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.None

pandas.Series.nlargest

Series.nlargest(n=5, keep='first')

Return the largest n elements.

Parameters n : int

Return this many descending sorted values

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - first : take the first occurrence. - last : take the last occurrence.

Returns top_n : Series

The n largest values in the Series, in sorted order

See also:

Series.nsmallest

Notes

Faster than .sort_values(ascending=False).head(n) for small n relative to the size of the Series object.

Examples

```python
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921    4.644710
82124     4.608745
421689    4.564804
425277    4.447014
718691    4.414137
43154     4.403750
283187    4.313922
595519    4.273635
503969    4.250236
121637    4.240952
dtype: float64
```
pandas.Series.nonzero

Series.nonzero()  
Return the indices of the elements that are non-zero  
This method is equivalent to calling `numpy.nonzero` on the series data. For compatibility with NumPy, the return value is the same (a tuple with an array of indices for each dimension), but it will always be a one-item tuple because series only have one dimension.  
See also:  
numpy.nonzero

Examples

```python
gp = pd.Series([0, 3, 0, 4])
gp.nonzero()  
(array([1, 3]),)  
gp.iloc[gp.nonzero()[0]]  
1 3  
3 4  
dtype: int64
```

pandas.Series.notna

Series.notna()  
Return a boolean same-sized object indicating if the values are not NA.  
See also:  
Series.isna boolean inverse of notna  
Series.notnull alias of notna  
notna top-level notna

pandas.Series.notnull

Series.notnull()  
Return a boolean same-sized object indicating if the values are not NA.  
See also:  
Series.isna boolean inverse of notna  
Series.notnull alias of notna  
notna top-level notna

pandas.Series.nsmallest

Series.nsmallest (n=5, keep='first')  
Return the smallest n elements.
Parameters  

- **n**: int
  - Return this many ascending sorted values
- **keep**: {'first', 'last', False}, default 'first'
  - Where there are duplicate values: - `first`: take the first occurrence. - `last`: take the last occurrence.

Returns  **bottom_n**: Series

- The n smallest values in the Series, in sorted order

See also:  
**Series.nlargest**

Notes

- Faster than `.sort_values().head(n)` for small n relative to the size of the Series object.

Examples

```python
>>> import pandas as pd
>>> import numpy as np

>>> s = pd.Series(np.random.randn(10**6))

>>> s.nsmallest(10)  # only sorts up to the N requested
288532   -4.954580
732345   -4.835960
64803    -4.812550
446457   -4.609998
501225   -4.483945
669476   -4.472935
973615   -4.401699
621279   -4.355126
773916   -4.347355
359919   -4.331927

dtype: float64
```

**pandas.Series.nunique**

**Series.nunique**(dropna=True)

- Return number of unique elements in the object.
  - Excludes NA values by default.
- **Parameters** **dropna**: boolean, default True
  - Don’t include NaN in the count.

Returns  **nunique**: int

**pandas.Series.pct_change**

**Series.pct_change**(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)

- Percent change over given number of periods.
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 offset alias string, optional

Increment to use from time series API (e.g. ‘M’ or BDay())

Returns `chg` : NDFrame

Notes

By default, the percentage change is calculated along the stat axis: 0, or `Index`, for `DataFrame` and 1, or `minor` for `Panel`. You can change this with the `axis` keyword argument.

**pandas.Series.pipe**

`Series.pipe` *(func, *args, **kwargs)*

Apply `func(self, *args, **kwargs)`

Parameters `func` : function

function to apply to the NDFrame. `args`, and `kwarg` s are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

`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:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write
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(f, arg2=b, arg3=c)
... )
```

### pandas.Series.plot

The `Series.plot` method in the pandas library allows for various types of plots of a Series using matplotlib / pylab. New in version 0.17.0: Each plot kind has a corresponding method on the `Series.plot` accessor: `s.plot(kind='line')` is equivalent to `s.plot.line()`.

**Parameters**

- **data**: Series
- **kind**: str
  - '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
- **ax**: matplotlib axes object
  - If not passed, uses gca()
- **figsize**: a tuple (width, height) in inches
- **use_index**: boolean, default True
  - Use index as ticks for x axis
- **title**: string 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**: boolean, default None (matlab style default)
Axis grid lines

**legend**: False/True/'reverse'

Place legend on axis subplots

**style**: list or dict

matplotlib line style per column

**logx**: boolean, default False

Use log scaling on x axis

**logy**: boolean, default False

Use log scaling on y axis

**loglog**: boolean, default False

Use log scaling on both x and y axes

**xticks**: sequence

Values to use for the xticks

**yticks**: sequence

Values to use for the yticks

**xlim**: 2-tuple/list

**ylim**: 2-tuple/list

**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**: boolean, 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**: boolean, 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**: same types as yerr.

**label**: label argument to provide to plot

**secondary_y**: boolean or sequence of ints, default False
If True then y-axis will be on the right

**mark_right** : boolean, default True

When using a secondary_y axis, automatically mark the column labels with “(right)” in the legend

**kwds** : keywords

Options to pass to matplotlib plotting method

Returns **axes** : matplotlib.AxesSubplot or np.array of them

**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 drop from frame. Raise KeyError if not found.

**Parameters**

- **item** : str
  
  Column label to be popped

**Returns**

- **popped** : 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'])
>>> df
   name     class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal         NaN

>>> df.pop('class')
0   bird
1   bird
2  mammal
3  mammal
Name: class, dtype: object

>>> df
   name  max_speed
0  falcon      389.0
1  parrot      24.0
```
2   lion       80.5
3  monkey     NaN

pandas.Series.pow

Series.pow (other, level=None, fill_value=None, axis=0)
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 one of the inputs.

Parameters other : Series or scalar value

fill_value : None or float value, default None (NaN)
Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name
Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:
Series.rpow

pandas.Series.prod

Series.prod (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the product of the values for the requested axis

Parameters axis : {index (0)}

skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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 prod : scalar or Series (if level specified)

pandas.Series.product

Series.product (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the product of the values for the requested axis

Parameters axis : {index (0)}

skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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

**numeric_only**: boolean, 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**

**prod**: scalar or Series (if level specified)

### pandas.Series.ptp

**Series.ptp**(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

**Returns the difference between the maximum value and the** minimum value in the object. This is the equivalent of the numpy.ndarray method ptp.

**Parameters**

**axis**: {index (0)}

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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

**numeric_only**: boolean, 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**

**ptp**: scalar or Series (if level specified)

### pandas.Series.put

**Series.put**(*args, **kwargs)

Applies the put method to its values attribute if it has one.

**See also**:

numpy.ndarray.put

### pandas.Series.quantile

**Series.quantile**(q=0.5, interpolation='linear')

Return value at the given quantile, a la numpy.percentile.

**Parameters**

**q**: float or array-like, default 0.5 (50% quantile)

0 <= q <= 1, the quantile(s) to compute

**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** *quantile*: float or Series

If \( q \) is an array, a Series will be returned where the index is \( q \) and the values are the quantiles.

**Examples**

```python
>>> s = 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
```

...Series:add

**pandas.Series.radd**

Series.radd(*other*, **, **level=None, **fill_value=None, **axis=0)

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 one of the inputs.

**Parameters**

- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
  Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
- **level**: int or name
  Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns** *result*: Series

**See also:**

Series.add
pandas.Series.rank

```python
Series.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)
```

Compute numerical data ranks (1 through n) along axis. 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'}
  - 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
- `numeric_only` : boolean, default None
  - Include only float, int, boolean data. Valid only for DataFrame or Panel objects
- `na_option` : {'keep', 'top', 'bottom'}
  - keep: leave NA values where they are
  - top: smallest rank if ascending
  - bottom: smallest rank if descending
- `ascending` : boolean, default True
  - False for ranks by high (1) to low (N)
- `pct` : boolean, default False
  - Computes percentage rank of data

**Returns**
- `ranks` : same type as caller

pandas.Series.ravel

```python
Series.ravel(order='C')
```

Return the flattened underlying data as an ndarray.

**See also:**
- `numpy.ndarray.ravel`

pandas.Series.rdiv

```python
Series.rdiv(other=None, fill_value=None, axis=0)
```

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 one of the inputs.
**Parameters**

- `other`: Series or scalar value
  - `fill_value`: None or float value, default None (NaN)
    - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  - `level`: int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- `result`: Series

See also:

- `Series.truediv`

### pandas.Series.reindex

**Series.reindex**(index=None, **kwargs)

Conform Series to new index with optional filling logic, placing 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 (should be specified using keywords)
  - New labels / index to conform to. Preferably an Index object to avoid duplicating data
  - method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
    - 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`: boolean, 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 \( \text{abs}(\text{index}[\text{indexer}] - \text{target}) \leq \text{tolerance} \).
    - 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.

New in version 0.17.0.

New in version 0.21.0: (list-like tolerance)

**Returns** reindexed : Series

**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.

```
>>> 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    301.00     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.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...     ...     'Chrome']
... >>> df.reindex(new_index)
   http_status response_time
Safari        404.00       0.07
Iceweasel     NaN          NaN
Comodo Dragon NaN          NaN
IE10          404.00       0.08
Chrome        200.00       0.02
```

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.

```
>>> df.reindex(new_index, fill_value=0)
   http_status response_time
Safari        404.0       0.07
Iceweasel     0            0.00
Comodo Dragon 0            0.00
IE10          404.0       0.08
Chrome        200.0       0.02
```
We can also reindex the columns.

We can also reindex the columns.

Or we can use "axis-style" keyword arguments

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

Suppose we decide to expand the dataframe to cover a wider date range.
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 backpropagate 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
2009-12-30  100
2009-12-31  100
2010-01-01  100
2010-01-02  101
2010-01-03  NaN
2010-01-04  100
2010-01-05  89
2010-01-06  88
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 thefillna() method.

See the user guide for more.

### pandas.Series.reindex_axis

`Series.reindex_axis(labels, axis=0, **kwargs)`

for compatibility with higher dims

### pandas.Series.reindex_like

`Series.reindex_like(other, method=None, copy=True, limit=None, tolerance=None)`

Return an object with matching indices to myself.

**Parameters**

- **other**: Object
  - **method**: string or None
  - **copy**: boolean, default True
  - **limit**: int, default None
    - Maximum number of consecutive labels to fill for inexact matches.
  - **tolerance**: optional
    - Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.
    - New in version 0.17.0.
    - New in version 0.21.0: (list-like tolerance)

**Returns**

- **reindexed**: same as input
Notes

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

**pandas.Series.rename**

`Series.rename(index=None, **kwargs)`  
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**  
- **index**: scalar, hashable sequence, dict-like or function, optional  
  - dict-like or functions are transformations to apply to the index. Scalar or hashable sequence-like will alter the `Series.name` attribute.
  - **copy**: boolean, default True  
    - Also copy underlying data
  - **inplace**: boolean, default False  
    - Whether to return a new `{% class %}`. 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.

**Returns**  
- `renamed`: Series (new object)

See also:  
`pandas.Series.rename_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 2
4 3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0 1
3 2
```

34.3. Series
pandas.Series.rename_axis

Series.rename_axis(mapper, axis=0, copy=True, inplace=False)
Alter the name of the index or columns.

Parameters
- mapper : scalar, list-like, optional
  Value to set the axis name attribute.
- axis : int or string, default 0
- copy : boolean, default True
  Also copy underlying data
- inplace : boolean, default False

Returns renamed : type of caller or None if inplace=True

See also:
pandas.Series.rename, pandas.DataFrame.rename, pandas.Index.rename

Notes
Prior to version 0.21.0, rename_axis could also be used to change the axis labels by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use rename instead.

Examples

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
   A  B
foo 0 1 4
    1 2 5
    2 3 6
```

```python
>>> df.rename_axis("bar", axis="columns")
   A  B
bar 0 1 4
    1 2 5
    2 3 6
```

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.
pandas: powerful Python data analysis toolkit, Release 0.21.0

(axis : where to reorder levels)

Returns type of caller (new object)

pandas.Series.repeat

Series.repeat(repeats, *args, **kwargs)

Repeat elements of an Series. Refer to numpy.ndarray.repeat for more information about the repeats argument.

See also:

numpy.ndarray.repeat

pandas.Series.replace

Series.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)

Replace values given in ‘to_replace’ with ‘value’.

Parameters to_replace : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - 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 regexs 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 and regex rules apply as above.

- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.

  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.

- 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 use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**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. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method** : string, optional, {‘pad’, ‘ffill’, ‘bfill’}

The method to use when for replacement, when `to_replace` is a list.

Returns **filled** : DataFrame

Raises **AssertionError**

- If `regex` is not a bool and `to_replace` is not None.

**TypeError**

- 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.

**ValueError**

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

See also:

NDFrame.reindex, NDFrame.asfreq, NDFrame.fillna

**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.
pandas.Series.resample

Series.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)

Convenience method for frequency conversion and resampling of time series. Object must have a
datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to
the on or level keyword.

Parameters

rule : string
    the offset string or object representing target conversion

axis : int, optional, default 0

closed : {'right', 'left'}
    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'}
    Which bin edge label to label bucket with. The default is ‘left’ for all frequency
    default of ‘right’.

convention : {'start', 'end', 's', 'e'}
    For PeriodIndex only, controls whether to use the start or end of rule

loffset : timedelta
    Adjust the resampled time labels

base : int, default 0
    For frequencies that evenly subdivide 1 day, the “origin” of the aggregated in-
    tervals. For example, for ’5min’ frequency, base could range from 0 through 4.
    Defaults to 0

on : string, optional
    For a DataFrame, column to use instead of index for resampling. Column must
    be datetime-like.

    New in version 0.19.0.

level : string or int, optional
    For a MultiIndex, level (name or number) to use for resampling. Level must be
datetime-like.

    New in version 0.19.0.

Notes

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
2000-01-01 00:01:30 NaN
2000-01-01 00:02:00 2.0
Freq: 30S, dtype: float64
```
Upsample the series into 30 second bins and fill the NaN values using the \textit{pad} method.

\begin{verbatim}
>>> 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
\end{verbatim}

Upsample the series into 30 second bins and fill the NaN values using the \textit{bfill} method.

\begin{verbatim}
>>> 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
\end{verbatim}

Pass a custom function via \textit{apply}

\begin{verbatim}
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+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
\end{verbatim}

For a Series with a PeriodIndex, the keyword \textit{convention} can be used to control whether to use the start or end of \textit{rule}.

\begin{verbatim}
>>> 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
\end{verbatim}

Resample by month using ‘start’ \textit{convention}. Values are assigned to the first month of the period.

\begin{verbatim}
>>> s.resample('M', convention='start').asfreq().head()
2012-01   1.0
2012-02   NaN
2012-03   NaN
2012-04   NaN
2012-05   NaN
Freq: M, dtype: float64
\end{verbatim}

Resample by month using ‘end’ \textit{convention}. Values are assigned to the last month of the period.

\begin{verbatim}
>>> s.resample('M', convention='end').asfreq()
2012-12   1.0
2013-01   NaN
2013-02   NaN
\end{verbatim}
For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```python
>>> df = pd.DataFrame(data=9*range(4), columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
    a   b   c   d
time
2000-01-01 00:00:00   0   3   6   9
2000-01-01 00:03:00   0   3   6   9
2000-01-01 00:06:00   0   3   6   9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```python
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*range(4),
                    columns=['a', 'b', 'c', 'd'],
                    index=pd.MultiIndex.from_product([time, [1, 2]])
                    )
>>> df2.resample('3T', level=0).sum()
    a   b   c   d
2000-01-01 00:00:00   0   6  12  18
2000-01-01 00:03:00   0   4   8  12
```

### pandas.Series.reset_index

`Series.reset_index(level=None, drop=False, name=None, inplace=False)`

Analogous to the `pandas.DataFrame.reset_index()` function, see docstring there.

**Parameters**

- **level**: int, str, tuple, or list, default None
  - Only remove the given levels from the index. Removes all levels by default
- **drop**: boolean, default False
  - Do not try to insert index into dataframe columns
- **name**: object, default None
  - The name of the column corresponding to the Series values
- **inplace**: boolean, default False
  - Modify the Series in place (do not create a new object)

**Returns**

- **resetted**: DataFrame, or Series if drop == True
Examples

```python
>>> s = pd.Series([1, 2, 3, 4], index=pd.Index(['a', 'b', 'c', 'd'],
name = 'idx'))
>>> s.reset_index()
     index 0 1 2 3 4
      0  0  1  2  3  4

>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo',
'foo', 'qux', 'qux'])),
           np.array(['one', 'two', 'one', 'two', 'one', 'two',
'one', 'two'])]
>>> s2 = pd.Series(np.random.randn(8),
index=pd.MultiIndex.from_arrays(arrays,
names=['a', 'b']))
>>> s2.reset_index(level='a')
          a       b
    one  bar -0.286320
    two  bar -0.587934
    one  baz  0.710491
    two  baz -1.429006
    one  foo  0.790700
    two  foo  0.824863
    one  qux -0.718963
    two  qux -0.055028
```

**pandas.Series.reshape**

Series.reshape(*args, **kwargs)

Deprecated since version 0.19.0: Calling this method will raise an error. Please call .values.reshape(...) instead.

return an ndarray with the values shape if the specified shape matches exactly the current shape, then return self (for compat)

See also:

numpy.ndarray.reshape

**pandas.Series.rfloordiv**

Series.rfloordiv(other, level=None, fill_value=None, axis=0)

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 one of the inputs.

Parameters other : Series or scalar value

fill_value : None or float value, default None (NaN)
Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : Series

**See also:**

`Series.floordiv`

---

**pandas.Series.rmod**

`Series.rmod(other, level=None, fill_value=None, axis=0)`

Modulo of series and other, element-wise (binary operator `rmod`).

Equivalent to `other % series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters**

- **other** : Series or scalar value
  - **fill_value** : None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  - **level** : int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : Series

**See also:**

`Series.mod`

---

**pandas.Series.rmul**

`Series.rmul(other, level=None, fill_value=None, axis=0)`

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 one of the inputs.

**Parameters**

- **other** : Series or scalar value
  - **fill_value** : None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  - **level** : int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : Series

**See also:**

`Series.mul`
pandas.Series.rolling

`Series.rolling(window, min_periods=None, freq=None, center=False, win_type=None, on=None, axis=0, closed=None)`

Provides rolling window calculations.

New in version 0.18.0.

**Parameters**

- **window**: int, or offset
  
  Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.
  
  If its an offset then this will be 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. This is new in 0.19.0

- **min_periods**: int, default None
  
  Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

- **freq**: string or DateOffset object, optional (default None)
  
  Deprecated since version 0.18.0: Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

- **center**: boolean, default False
  
  Set the labels at the center of the window.

- **win_type**: string, default None
  
  Provide a window type. See the notes below.

- **on**: string, optional
  
  For a DataFrame, column on which to calculate the rolling window, rather than the index

- **closed**: string, default None
  
  Make the interval closed on the ‘right’, ‘left’, ‘both’ or ‘neither’ endpoints. For offset-based windows, it defaults to ‘right’. For fixed windows, defaults to ‘both’. Remaining cases not implemented for fixed windows.

  New in version 0.20.0.

- **axis**: int or string, default 0

**Returns**

a Window or Rolling sub-classed for the particular operation

**Notes**

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The `freq` keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the `mean`).

To learn more about the offsets & frequency strings, please see [this link](#).

The recognized win_types are:
• boxcar
• triang
• blackman
• hamming
• bartlett
• parzen
• bohman
• blackmanharris
• nuttall
• barthann
• kaiser (needs beta)
• gaussian (needs std)
• general_gaussian (needs power, width)
• slepian (needs width).

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

Rolling sum with a window length of 2, using the ‘triang’ window type.

```python
def.rolling(2, win_type='triang').sum()
```

B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN

Rolling sum with a window length of 2, min_periods defaults to the window length.

```python
def.rolling(2).sum()
```

B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN

Same as above, but explicitly set the min_periods
```
A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```python
>>> df = 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')])
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```python
>>> df.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 2.0
2013-01-01 09:00:05 NaN
2013-01-01 09:00:06 4.0
```

**pandas.Series.round**

Series.round(decimals=0, *args, **kwargs)

Round each value in a Series to the given number of decimals.

**Parameters**

decimals : int
    Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

**Returns**

Series object

See also:

numpy.around, DataFrame.round

**pandas.Series.rpow**

Series.rpow(other, level=None, fill_value=None, axis=0)

Exponential power of series and other, element-wise (binary operator rpow).
pandas: powerful Python data analysis toolkit, Release 0.21.0

Equivalent to other ** series, but with support to substitute a fill_value for missing data in one of the inputs.

**Parameters**
- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**
- **result**: Series

**See also**:
- `Series.pow`

### pandas.Series.rsub

**Series.rsub** (other, level=None, fill_value=None, axis=0)

Subtraction of series and other, element-wise (binary operator rsub).

Equivalent to other - series, but with support to substitute a fill_value for missing data in one of the inputs.

**Parameters**
- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**
- **result**: Series

**See also**:
- `Series.sub`

### pandas.Series.rtruediv

**Series.rtruediv** (other, level=None, fill_value=None, axis=0)

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 one of the inputs.

**Parameters**
- **other**: Series or scalar value
- **fill_value**: None or float value, default None (NaN)
  - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level
Returns result: Series

See also:
Series.truediv

pandas.Series.sample

Series.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)

Returns a random sample of items from an axis of object.

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: boolean, optional
   Sample with or without replacement. Default = False.
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. inf and -inf values not allowed.
random_state: int or numpy.random.RandomState, optional
   Seed for the random number generator (if int), or numpy RandomState object.
axis: int or string, optional
   Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns A new object of same type as caller.

Examples

Generate an example Series and DataFrame:

```python
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2    -0.972766
3    -1.598270
4    -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
```
Next extract a random sample from both of these objects...

3 random elements from the Series:

```python
>>> s.sample(n=3)
27  -0.994689
55  -1.049016
67  -0.224565
dtype: float64
```

And a random 10% of the DataFrame with replacement:

```python
>>> df.sample(frac=0.1, replace=True)
   A     B     C     D
35  1.981780 0.142106 1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640 0.217116
40  0.823173 -0.078816 1.009536 1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
  6  -0.148339 0.832938 1.787600 -1.383767
```

**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.

**Parameters**

- `value` : array_like
  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**

- `indices` : array of ints
  Array of insertion points with the same shape as `value`.

**See also:**

- `numpy.searchsorted`
Notes

Binary search is used to find the required insertion points.

Examples

```python
cpy = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```  
```python
>>> x.searchsorted(4)
array([3])
```  
```python
>>> x.searchsorted([0, 4])
array([0, 3])
```  
```python
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```  
```python
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```  
```python
x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'])
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```  
```python
>>> x.searchsorted('bread')
array([1])  # Note: an array, not a scalar
```  
```python
>>> x.searchsorted(['bread'])
array([1])
```  
```python
>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])
```  
```python
>>> x.searchsorted(['bread', 'eggs'], side='right')
array([3, 4])  # eggs before milk
```  
**pandas.Series.select**

Series.select (crit, axis=0)

Return data corresponding to axis labels matching criteria

DEPRECATED: use df.loc[df.index.map(crit)] to select via labels

Parameters

- **crit**: function

  To be called on each index (label). Should return True or False
pandas: powerful Python data analysis toolkit, Release 0.21.0

axis : int

Returns selection : type of caller

pandas.Series.sem

Series.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased standard error of the mean over requested axis.

Parameters

axis : {index (0)}

skipna : boolean, 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

degrees of freedom

numeric_only : boolean, 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 sem : scalar or Series (if level specified)

pandas.Series.set_axis

Series.set_axis(labels, axis=0, inplace=None)

Assign desired index to given axis

Parameters

labels: list-like or Index

The values for the new index

axis : int or string, default 0

inplace : boolean, default None

Whether to return a new NDFrame instance.

WARNING: inplace=None currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

.. versionadded:: 0.21.0

The signature is make consistent to the rest of the API. Previously, the “axis” and “labels” arguments were respectively the first and second positional arguments.

Returns renamed : NDFrame or None

An object of same type as caller if inplace=False, None otherwise.

See also:

pandas.NDFrame.rename
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, inplace=False)
a 1
b 2
c 3
dtype: int64
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
>>> df.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
   A  B
0 1 4
1 2 5
2 3 6
>>> df.set_axis(['I', 'II'], axis=1, inplace=False)
   I  II
0 1 4
1 2 5
2 3 6
>>> df.set_axis(['i', 'ii'], axis=1, inplace=True)
>>> df
   i  ii
0 1 4
1 2 5
2 3 6
```

`pandas.Series.set_value`

Series.set_value(label, value, takeable=False)

Quickly set single value at passed label. If label is not contained, a new object is created with the label placed at the end of the result index.

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

**Parameters**

- **label**: object
  
  Partial indexing with MultiIndex not allowed

- **value**: object
  
  Scalar value

- **takeable**: interpret the index as indexers, default False

**Returns**

- **series**: Series
  
  If label is contained, will be reference to calling Series, otherwise a new object
**pandas.Series.shift**

Series.shift \((periods=1, freq=None, axis=0)\)
Shift index by desired number of periods with an optional time freq

**Parameters**
- **periods**: int
  Number of periods to move, can be positive or negative
- **freq**: DateOffset, timedelta, or time rule string, optional
  Increment to use from the tseries module or time rule (e.g. ‘EOM’). See Notes.
- **axis**: {0, ‘index’}

**Returns**
- **shifted**: Series

**Notes**
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.

**pandas.Series.skew**

Series.skew \((axis=None, skipna=None, level=None, numeric_only=None, **kwargs)\)
Return unbiased skew over requested axis Normalized by N-1

**Parameters**
- **axis**: {index (0)}
- **skipna**: boolean, default True
  Exclude NA/null values. If an entire row/column is NA or empty, 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
- **numeric_only**: boolean, 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**
- **skew**: 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.

**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**

```python
Series.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True)
```

Sort object by labels (along an axis)

**Parameters**

- **axis**: index to direct sorting
- **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**: boolean, default True
  - Sort ascending vs. descending
- **inplace**: bool, default False
  - if True, perform operation in-place
- **kind**: {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
  - Choice of sorting algorithm. See also `ndarray.np.sort` for more information. `mergesort` is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position**: {'first', 'last'}, default 'last'
  - `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

**Returns**

- **sorted_obj**: Series

**pandas.Series.sort_values**

```python
Series.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

Sort by the values along either axis

New in version 0.17.0.

**Parameters**

- **axis**: {0, 'index'}, default 0
  - Axis to direct sorting
- **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'}, default 'quicksort'

Choice of sorting algorithm. See also ndarray.np.sort for more information. mergesort is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position : {'first', 'last'}, default 'last'

first puts NaNs at the beginning, last puts NaNs at the end

Returns sorted_obj : 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],
... })
>>> df
   col1 col2  col3
0     A    2     0
1     A    1     1
2     B    9     9
3   NaN    8     4
4     D    7     2
5     C    4     3
```

Sort by col1

```python
>>> df.sort_values(by=['col1'])
   col1  col2  col3
0     A    2     0
1     A    1     1
2     B    9     9
5     C    4     3
4     D    7     2
3   NaN    8     4
```

Sort by multiple columns

```python
>>> df.sort_values(by=['col1', 'col2'])
   col1  col2  col3
1     A    1     1
0     A    2     0
2     B    9     9
5     C    4     3
4     D    7     2
3   NaN    8     4
```

Sort Descending

```python
>>> df.sort_values(by='col1', ascending=False)
   col1  col2  col3
0     A    2     0
5     C    4     3
2     B    9     9
4     D    7     2
3   NaN    8     4
```

Putting NAs first

```python
>>> df.sort_values(by='col1', ascending=False, na_position='first')
coll col2 col3
3 NaN 8 4
4 D 7 2
5 C 4 3
2 B 9 9
0 A 2 0
1 A 1 1
```

**pandas.Series.sortlevel**

`Series.sortlevel(level=0, ascending=True, sort_remaining=True)`

DEPRECATED: use `Series.sort_index()`

Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

- **Parameters**
  - `level`: int or level name, default None
  - `ascending`: bool, default True

- **Returns**
  - `sorted`: Series

See also:

`Series.sort_index`

**pandas.Series.squeeze**

`Series.squeeze(axis=None)`

Squeeze length 1 dimensions.

- **Parameters**
  - `axis`: None, integer or string axis name, optional
    - The axis to squeeze if 1-sized.
      - New in version 0.20.0.
  - `skipna`: boolean, default True

- **Returns**
  - `scalar` if 1-sized, else original object

**pandas.Series.std**

`Series.std(axis=None, skipna=None, 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`: boolean, 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

degrees of freedom

numeric_only : boolean, 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 std : scalar or Series (if level specified)

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.str.split('_')
>>> s.str.replace('_', '')
```

pandas.Series.sub

Series.sub(other, level=None, fill_value=None, axis=0)

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 one of
the inputs.

Parameters other : Series or scalar value

fill_value : None or float scalar, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result
will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.rsub

pandas.Series.subtract

Series.subtract(other, level=None, fill_value=None, axis=0)

Subtraction of series and other, element-wise (binary operator sub).
pandas: powerful Python data analysis toolkit, Release 0.21.0

Equivalent to `series - other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters**

- `other`: Series or scalar value
- `fill_value`: None or float value, default None (NaN)
  
  Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

- `level`: int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- `result`: Series

**See also:**

- `Series.rsub`
- `Series.sum`
- `Series.swapaxes`
- `Series.swaplevel`

---

### pandas.Series.sum

**Series.sum** *(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)*

Return the sum of the values for the requested axis

**Parameters**

- `axis`: {index (0)}
- `skipna`: boolean, default True
  
  Exclude NA/null values. If an entire row/column is NA or empty, 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

- `numeric_only`: boolean, 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**

- `sum`: scalar or Series (if level specified)

---

### 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

**Parameters**

- `i, j`: int, string (can be mixed)
  
  Level of index to be swapped. Can pass level name as string.
**pandas.Series.tail**

`Series.tail(n=5)`

Return the last n rows.

**Parameters**

- `n`: int, default 5
  - Number of rows to select.

**Returns**

- `obj_tail`: type of caller
  - The last n rows of the caller object.

**pandas.Series.take**

`Series.take(indices, axis=0, convert=None, is_copy=True, **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`: int, default 0
  - The axis on which to select elements. “0” means that we are selecting rows, “1” means that we are selecting columns, etc.
- `convert`: bool, default True
  - Deprecated since version 0.21.0: In the future, negative indices will always be converted.
  - Whether to convert negative indices into positive ones. For example, -1 would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.
- `is_copy`: bool, default True
  - Whether to return a copy of the original object or not.

**Returns**

- `taken`: type of caller
  - An array-like containing the elements taken from the object.

See also:

- `numpy.ndarray.take`, `numpy.take`

Examples
>>> 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.

>>> 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).

>>> 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.

>>> 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=None, sep=None, **kwargs)

Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for
example.

Parameters excel : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy past-
ing into excel. if False, write a string representation of the object to the clipboard

sep : optional, defaults to tab

other keywords are passed to to_csv
Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

pandas.Series.to_csv

`Series.to_csv(path=None, index=True, sep='\n', na_rep='', float_format=None, header=False, index_label=None, mode='w', encoding=None, date_format=None, decimal='\,')`

Write Series to a comma-separated values (csv) file

Parameters

- **path**: string or file handle, default None
  - File path or object, if None is provided the result is returned as a string.
- **na_rep**: string, default “”
  - Missing data representation
- **float_format**: string, default None
  - Format string for floating point numbers
- **header**: boolean, default False
  - Write out series name
- **index**: boolean, default True
  - Write row names (index)
- **index_label**: string or sequence, 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 DataFrame uses MultiIndex.
- **mode**: Python write mode, default ‘w’
- **sep**: character, default “,”
  - Field delimiter for the output file.
- **encoding**: string, optional
  - A string representing the encoding to use if the contents are non-ascii, for python versions prior to 3
- **date_format**: string, default None
  - Format string for datetime objects.
- **decimal**: string, default ‘,’
  - Character recognized as decimal separator. E.g. use ‘,’ for European data
pandas.Series.to_dense

Series.to_dense()
Return dense representation of NDFrame (as opposed to sparse)

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.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 collec-
tions.defaultdict, you must pass it initialized.

New in version 0.21.0.

Returns value_dict : collections.Mapping

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(<type '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)
Write Series to an excel sheet

New in version 0.20.0.

Parameters excel_writer : string or ExcelWriter object
File path or existing ExcelWriter

sheet_name : string, default ‘Sheet1’
Name of sheet which will contain DataFrame

na_rep : string, default ”
Missing data representation

float_format : string, default None
Format string for floating point numbers
**columns**: sequence, optional

Columns to write

**header**: boolean or list of string, 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**: boolean, default True

Write row names (index)

**index_label**: string or sequence, 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 DataFrame uses MultiIndex.

**startrow**: upper left cell row to dump data frame

**startcol**: upper left cell column to dump data frame

**engine**: string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge_cells**: boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.

**encoding**: string, default None

encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf_rep**: string, default ‘inf’

Representation for infinity (there is no native representation for infinity in Excel)

**freeze_panes**: tuple of integer (length 2), default None

Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

**Notes**

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```python
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.
### pandas.Series.to_frame

#### Series.to_frame (name=None)

Convert Series to DataFrame

**Parameters**

- **name**: object, default None
  
The passed name should substitute for the series name (if it has one).

**Returns**

- **data_frame**: DataFrame

### pandas.Series.to_hdf

#### Series.to_hdf (path_or_buf, key, **kwargs)

Write the contained data to an HDF5 file using HDFStore.

**Parameters**

- **path_or_buf**: the path (string) or HDFStore object
- **key**: string
  
  identifier for the group in the store
- **mode**: optional, {'a', 'w', 'r+'}, default 'a'
  
  - `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+` It is similar to `a`, but the file must already exist.
- **format**: 'fixed(f)|table(t)', default is 'fixed'
  
  - `fixed(f)` [Fixed format] Fast writing/reading. Not-appendable, nor searchable
  - `table(t)` [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**: boolean, default False
  
  For Table formats, 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.
  
  Applicable only to format='table'.
- **complevel**: int, 0-9, default None
  
  Specifies a compression level for data. A value of 0 disables compression.
- **complib**: {'zlib', 'lzoo', '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:z4`, `blosc:z4hc`, `blosc:snappy`,
  `blosc:zlib`, `blosc:zstd`}. Specifying a compression library which is not available issues a ValueError.
- **fletcher32**: bool, default False
If applying compression use the fletcher32 checksum

**dropna**: boolean, default False.
If true, ALL nan rows will not be written to store.

**pandas.Series.to_json**

```python
Series.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=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**: the path or buffer to write the result string
  - if this is None, return the converted string
- **orient**: string
  - **Series**
    - default is ‘index’
    - allowed values are: {‘split’,’records’,’index’}
  - **DataFrame**
    - default is ‘columns’
    - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
  - 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, and the data component is like orient=’records’.
      - Changed in version 0.20.0.
- **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**: 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**: string, default ‘ms’ (milliseconds)
pandas: powerful Python data analysis toolkit, Release 0.21.0

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 : boolean, default False
If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError
if incorrect ‘orient’ since others are not list like.
New in version 0.19.0.
compression : {None, ‘gzip’, ‘bz2’, ‘xz’}
A string representing the compression to use in the output file, only used when
the first argument is a filename
New in version 0.21.0.
Returns same type as input object with filtered info axis
See also:
pd.read_json
Examples
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...
index=['row 1', 'row 2'],
...
columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
"index":["row 1","row 2"],
"data":[["a","b"],["c","d"]]}'

Encoding/decoding a Dataframe using 'index' formatted JSON:
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'

Encoding with Table Schema
>>> 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": "0.20.0"},

34.3. Series

1401


"data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
{"index": "row 2", "col 1": "c", "col 2": "d"}]

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)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series
to\texttt{latex}-specific options:

\textbf{bold_rows} [boolean, default False] Make the row labels bold in the output

\textbf{column_format} [str, default None] The columns format as specified in \LaTeX{} table format e.g: ‘rcl’ for 3 columns

\textbf{longtable} [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage\{longtable\} to your \LaTeX{} preamble.

\textbf{escape} [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

\textbf{encoding} [str, default None] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

\textbf{decimal} [string, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.

New in version 0.18.0.

\textbf{multicolumn} [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

\textbf{multicolumn_format} [str, default ‘l’] The alignment for multicolumns, similar to column_format The default will be read from the config module.

New in version 0.20.0.

\textbf{multirow} [boolean, 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.

New in version 0.20.0.

pandas.Series.to_msgpack

Series.to_msgpack (path_or_buf=None, encoding='utf-8', **kwargs)

msgpack (serialize) object to input file path

\textbf{THIS IS AN EXPERIMENTAL LIBRARY} and the storage format may not be stable until a future release.

\textbf{Parameters} \textbf{path} : string File path, buffer-like, or None
if None, return generated string

append : boolean whether to append to an existing msgpack
(default is False)

compress : type of compressor (zlib or blosc), default to None (no
compression)

**pandas.Series.to_period**

```
Series.to_period(freq=None, copy=True)
```

Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not
passed)

**Parameters**

freq : string, default

**Returns**

ts : Series with PeriodIndex

**pandas.Series.to_pickle**

```
Series.to_pickle(path, compression='infer', protocol=4)
```

Pickle (serialize) object to input file path.

**Parameters**

path : string

File path

a string representing the compression to use in the output file

New in version 0.20.0.

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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python >=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value.A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

**pandas.Series.to_sparse**

```
Series.to_sparse(kind='block', fill_value=None)
```

Convert Series to SparseSeries

**Parameters**

kind : {‘block’, ‘integer’}

fill_value : float, defaults to NaN (missing)

**Returns**

sp : SparseSeries
pandas.Series.to_sql

Series.to_sql(name, con, flavor=None, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None)
Write records stored in a DataFrame to a SQL database.

Parameters

- name : string
  Name of SQL table

- con : SQLAlchemy engine or DBAPI2 connection (legacy mode)
  Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

- flavor : 'sqlite', default None
  Deprecated since version 0.19.0: ‘sqlite’ is the only supported option if SQLAlchemy is not used.

- schema : string, default None
  Specify the schema (if database flavor supports this). If None, use default schema.

- if_exists : {'fail', 'replace', 'append'}, default 'fail'
  • fail: If table exists, do nothing.
  • replace: If table exists, drop it, recreate it, and insert data.
  • append: If table exists, insert data. Create if does not exist.

- index : boolean, default True
  Write DataFrame index as a column.

- index_label : string 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, default None
  If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

- dtype : dict of column name to SQL type, default None
  Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

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)
Render a string representation of the Series

Parameters

- buf : StringIO-like, optional
  buffer to write to

- na_rep : string, 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: boolean, default True
    Add the Series header (index name)

index : bool, optional
    Add index (row) labels, default True

length : boolean, default False
    Add the Series length

dtype : boolean, default False
    Add the Series dtype

name : boolean, 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.

Returns formatted : string (if not buffer passed)

pandas.Series.to_timestamp

Series.to_timestamp (freq=None, how='start', copy=True)
    Cast to datetimeindex of timestamps, at beginning of period

Parameters freq : string, default frequency of PeriodIndex
    Desired frequency

how : {'s', 'e', 'start', 'end'}
    Convention for converting period to timestamp; start of period vs. end

Returns ts : Series with DatetimeIndex

pandas.Series.to_xarray

Series.to_xarray ()
    Return an xarray object from the pandas object.

Returns a DataArray for a Series
    a Dataset for a DataFrame
    a DataArray for higher dims

Notes

See the xarray docs
Examples

```python
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                    'B' : ['foo', 'bar', 'foo'],
                    'C' : np.arange(4.,7))
>>> df
   A  B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0

>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (index: 3)
Coordinates:
  * index (index) int64 0 1 2
Data variables:
  A (index) int64 1 1 2
  B (index) object 'foo' 'bar' 'foo'
  C (index) float64 4.0 5.0 6.0

>>> df = pd.DataFrame({'A' : [1, 1, 2],
                    'B' : ['foo', 'bar', 'foo'],
                    'C' : np.arange(4.,7)}).set_index(['B','A'])
>>> df
    C
   B A
foo 1 4.0
   bar 1 5.0
   foo 2 6.0

>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (A: 2, B: 2)
Coordinates:
  * B (B) object 'bar' 'foo'
  * A (A) int64 1 2
Data variables:
  C (B, A) float64 5.0 nan 4.0 6.0

>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
               items=list('ABCD'),
               major_axis=pd.date_range('20130101', periods=3),
               minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second

>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
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]],
       [[24, 25],
       [26, 27],
       [28, 29]],
       [[30, 31],
       [32, 33],
       [34, 35]],
       [[36, 37],
       [38, 39],
       [40, 41]],
       [[42, 43],
       [44, 45],
       [46, 47]]],
shape=(3, 2),
dtype=int64,
depth=4)
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)

See also:
numpy.ndarray.tolist

pandas.Series.transform

Series.transform(func, *args, **kwargs)
Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
New in version 0.20.0.

Parameters func : callable, string, dictionary, or list of string/callables
To apply to column

Accepted Combinations are:
• string function name
• function
• list of functions
• dict of column names -> functions (or list of functions)

Returns transformed : NDFrame

See also:
pandas.NDFrame.aggregate, pandas.NDFrame.apply
Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan

>>> df.transform(lambda x: (x - x.mean()) / x.std())
```

```
   A         B         C
2000-01-01 0.579457 1.236184 0.123424
2000-01-02 0.370357 -0.605875 -1.231325
2000-01-03 1.455756 -0.277446 0.288967
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 -0.498658 1.274522 1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710 0.005378
```

pandas.Series.transpose

```python
Series.transpose(*args, **kwargs)
return the transpose, which is by definition self
```

pandas.Series.truediv

```python
Series.truediv(other, level=None, fill_value=None, axis=0)
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 one of the inputs.
```

Parameters

- **other**: Series or scalar value
  - **fill_value**: None or float value, default None (NaN)
    - Fill missing (NaN) values with this value. If both Series are missing, the result will be missing
  - **level**: int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

- **result**: Series

See also:

- Series.rtruediv

pandas.Series.truncate

```python
Series.truncate(before=None, after=None, axis=None, copy=True)
```

Truncates a sorted DataFrame/Series before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters

- **before**: date, string, int
Truncate all rows before this index value

\textbf{after} : date, string, int

Truncate all rows after this index value

\textbf{axis} : \{0 or ‘index’, 1 or ‘columns’\}

- 0 or ‘index’: apply truncation to rows
- 1 or ‘columns’: apply truncation to columns

Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels)

\textbf{copy} : boolean, default is True,

return a copy of the truncated section

\textbf{Returns} \textbf{truncated} : type of caller

\textbf{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.truncate(before=2, after=4)
A   B   C
2  b   g   l
3  c   h   m
4  d   i   n
```

```python
>>> df = pd.DataFrame({'A': [1, 2, 3, 4, 5],
                    ...
                    'B': [6, 7, 8, 9, 10],
                    ...
                    'C': [11, 12, 13, 14, 15]},
                    ...
                    index=['a', 'b', 'c', 'd', 'e'])
>>> df.truncate(before='b', after='d')
A   B   C
b  2   7   12
c  3   8   13
d  4   9   14
```

The index values in \texttt{truncate} can be datetimes or string dates. Note that \texttt{truncate} assumes a 0 value for any unspecified date component in a \texttt{DatetimeIndex} in contrast to slicing which returns any partially matching dates.

```python
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.truncate('2016-01-05', '2016-01-10').tail()
A
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
```

```python
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
A
2016-01-10 23:59:55 1
2016-01-10 23:59:56 1
2016-01-10 23:59:57 1
```
pandas.Series.tshift

Series.tshift \( \text{(periods=1, freq=None, axis=0)} \)
Shift the time index, using the index’s frequency if available.

**Parameters**
- **periods** : int
  Number of periods to move, can be positive or negative
- **freq** : DateOffset, timedelta, or time rule string, default None
  Increment to use from the tseries module or time rule (e.g. ‘EOM’)
- **axis** : int or basestring
  Corresponds to the axis that contains the Index

**Returns**
- **shifted** : NDFrame

**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

Series.tz_convert \( \text{(tz, axis=0, level=None, copy=True)} \)
Convert tz-aware axis to target time zone.

**Parameters**
- **tz** : string or pytz.timezone object
- **axis** : the axis to convert
- **level** : int, str, default None
  If axis ia a MultiIndex, convert a specific level. Otherwise must be None
- **copy** : boolean, default True
  Also make a copy of the underlying data

**Raises**
- **TypeError**
  If the axis is tz-naive.

pandas.Series.tz_localize

Series.tz_localize \( \text{(tz, axis=0, level=None, copy=True, ambiguous='raise')} \)
Localize tz-naive TimeSeries to target time zone.

**Parameters**
- **tz** : string or pytz.timezone object
- **axis** : the axis to localize
- **level** : int, str, default None
If axis ia a MultiIndex, localize a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**ambiguous** : ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’

- ‘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

**infer_dst** : boolean, default False

Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order

Raises **TypeError**

If the TimeSeries is tz-aware and tz is not None.

### pandas.Series.unique

**Series.unique()**

Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

**Parameters** **values** : 1d array-like

**Returns** unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

See also:

**unique, Index.unique, Series.unique**

### pandas.Series.unstack

**Series.unstack**(level=-1, fill_value=None)

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

**Parameters** **level** : int, string, or list of these, default last level

Level(s) to unstack, can pass level name

**fill_value** : replace NaN with this value if the unstack produces missing values

**Returns** **unstacked** : DataFrame
Examples

```python
>>> s = pd.Series([1, 2, 3, 4],
... index=pd.MultiIndex.from_product([['one', 'two'], ['a', 'b']]))
>>> s
one    a   1
      b   2
two   a   3
      b   4
dtype: int64

>>> s.unstack(level=-1)
   a  b
one 1 2
two 3 4

>>> s.unstack(level=0)
   one  two
   a    1  3
   b    2  4
```

**pandas.Series.update**

`Series.update(other)`

Modify Series in place using non-NA values from passed Series. Aligns on index.

**Parameters**

- **other** : Series

**Examples**

```python
>>> 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.

```python
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, np.nan, 6]))
>>> s
0   4
1   2
2   6
dtype: int64
```

**pandas.Series.valid**

Series.valid(*inplace=False, **kwargs*)

**pandas.Series.value_counts**

Series.value_counts(*normalize=False, sort=True, ascending=False, bins=None, dropna=True*)

Returns object 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**: boolean, default False
  
  If True then the object returned will contain the relative frequencies of the unique values.

- **sort**: boolean, default True
  
  Sort by values

- **ascending**: boolean, default False
  
  Sort in ascending order

- **bins**: integer, optional
  
  Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data

- **dropna**: boolean, default True
  
  Don’t include counts of NaN.

**Returns**

- **counts**: Series

**pandas.Series.var**

Series.var(*axis=None, skipna=None, 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**: boolean, 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

degrees of freedom

**numeric_only**: boolean, 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**

var : scalar or Series (if level specified)

**pandas.Series.view**

Series.view(*dtype=None*)

**pandas.Series.where**

Series.where(*cond*, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from other.

**Parameters**

cond : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as cond.

other : scalar, NDFrame, or callable

Entries where *cond* is False are replaced with corresponding value from other. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

events : str, {'raise', 'ignore'}, default ‘raise’

- raise : allow exceptions to be raised
- ignore : suppress exceptions. On error return original object

Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

try_cast : boolean, default False
try to cast the result back to the input type (if possible),

**raise_on_error**: boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)
Deprecation: since version 0.21.0.

**Returns** wh : same type as caller

**See also:**

*DataFrame.mask()*

**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

>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN

>>> s.where(s > 1, 10)
0    10.0
1    10.0
2      2.0
3      3.0
4      4.0

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A   B
0   0   -1
1   -2    3
2   -4   -5
3    6   -7
```
pandas.Series.xs

Series.xs(key, axis=0, level=None, drop_level=True)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on
the rows (axis=0).

Parameters key: object

Some label contained in the index, or partially in a MultiIndex

axis: int, 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: boolean, default True

If False, returns object with same levels as self.

Returns xs: Series or DataFrame

Notes

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs function-
ality, see MultiIndex Slicers

Examples

>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A 4
B 5
C 2
Name: a

```
>>> df.xs('C', axis=1)
a 2
b 9
c 3
Name: C
```

```
>>> df
   A  B  C  D
first second third
bar  one 1  4  1 8 9
two  one 1  7  5 5 0
baz  one 1  6  6 8 0
three two 2  5  3 5 3
>>> df.xs(('baz', 'three'))
   A  B  C  D
third
2  5 3 5 3
```
34.3.3 Conversion

- `Series.astype(dtype[, copy, errors])` Cast a pandas object to a specified dtype `dtype`.
- `Series.infer_objects()` Attempt to infer better dtypes for object columns.
- `Series.copy([deep])` Make a copy of this objects data.
- `Series.isna()` Return a boolean same-sized object indicating if the values are NA.
- `Series.notna()` Return a boolean same-sized object indicating if the values are not NA.

34.3.4 Indexing, iteration

- `Series.get(key[, default])` Get item from object for given key (DataFrame column, Panel slice, etc.).
- `Series.at` Fast label-based scalar accessor
- `Series.iat` Fast integer location scalar accessor.
- `Series.loc` Purely label-location based indexer for selection by label.
- `Series.iloc` Purely integer-location based indexing for selection by position.
- `Series.__iter__()` Return an iterator of the values.
- `Series.iteritems()` Lazily iterate over (index, value) tuples

34.3.4.1 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)

For more information on .at, .iat, .loc, and .iloc, see the indexing documentation.

34.3.5 Binary operator functions

- `Series.add(other[, level, fill_value, axis])` Addition of series and other, element-wise (binary operator `add`).
- `Series.sub(other[, level, fill_value, axis])` Subtraction of series and other, element-wise (binary operator `sub`).
- `Series.mul(other[, level, fill_value, axis])` Multiplication of series and other, element-wise (binary operator `mul`).
- `Series.div(other[, level, fill_value, axis])` Floating division of series and other, element-wise (binary operator `truediv`).
- `Series.truediv(other[, level, fill_value, axis])` Floating division of series and other, element-wise (binary operator `truediv`).
- `Series.floor_div(other[, level, fill_value, axis])` Integer division of series and other, element-wise (binary operator `floordiv`).
- `Series.mod(other[, level, fill_value, axis])` Modulo of series and other, element-wise (binary operator `mod`).

Continued on next page
Table 34.29 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.pow</code></td>
<td>Exponential power of series and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>Series.radd</code></td>
<td>Addition of series and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>Series.rsub</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>Series.rmul</code></td>
<td>Multiplication of series and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>Series.rdiv</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>Series.rtruediv</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>Series.rfloordiv</code></td>
<td>Integer division of series and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>Series.rmod</code></td>
<td>Modulo of series and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>Series.rpow</code></td>
<td>Exponential power of series and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>Series.combine</code></td>
<td>Perform elementwise binary operation on two Series using given function</td>
</tr>
<tr>
<td><code>Series.combine_first</code></td>
<td>Combine Series values, choosing the calling Series’s values first.</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>Less than of series and other, element-wise (binary operator <code>lt</code>).</td>
</tr>
<tr>
<td><code>Series.gt</code></td>
<td>Greater than of series and other, element-wise (binary operator <code>gt</code>).</td>
</tr>
<tr>
<td><code>Series.le</code></td>
<td>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>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>Not equal to of series and other, element-wise (binary operator <code>ne</code>).</td>
</tr>
<tr>
<td><code>Series.eq</code></td>
<td>Equal to of series and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
</tbody>
</table>

34.3.6 Function application, GroupBy & Window

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.apply</code></td>
<td>Invoke function on values of Series.</td>
</tr>
<tr>
<td><code>Series.aggregate</code></td>
<td>Aggregate using callable, string, dict, or list of string/callables</td>
</tr>
<tr>
<td><code>Series.transform</code></td>
<td>Call function producing a like-indexed NDFrame</td>
</tr>
<tr>
<td><code>Series.map</code></td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td><code>Series.groupby</code></td>
<td>Group series using mapper (dict or key function, apply</td>
</tr>
<tr>
<td><code>Series.rolling</code></td>
<td>Provides rolling window calculations.</td>
</tr>
<tr>
<td><code>Series.expanding</code></td>
<td>Provides expanding transformations.</td>
</tr>
</tbody>
</table>

Continued on next page
Series.ewm([com, span, halflife, alpha, ...]) Provides exponential weighted functions

### 34.3.7 Computations / Descriptive Stats

**Series.abs()**
Return an object with absolute value taken–only applicable to objects that are all numeric.

**Series.all([axis, bool_only, skipna, level])**
Return whether all elements are True over requested axis.

**Series.any([axis, bool_only, skipna, level])**
Return whether any element is True over requested axis.

**Series.autocorr([lag])**
Lag-N autocorrelation

**Series.between(left, right[, inclusive])**
Return boolean Series equivalent to left <= series <= right.

**Series.clip([lower, upper, axis, inplace])**
Trim values at input threshold(s).

**Series.clip_lower(threshold[, axis, inplace])**
Return copy of the input with values below given value(s) truncated.

**Series.clip_upper(threshold[, axis, inplace])**
Return copy of input with values above given value(s) truncated.

**Series.corr(other[, method, min_periods])**
Compute correlation with other Series, excluding missing values.

**Series.count([level])**
Return number of non-NA/null observations in the Series.

**Series.cov(other[, min_periods])**
Compute covariance with Series, excluding missing values.

**Series.cummax([axis, skipna])**
Return cumulative max over requested axis.

**Series.cummin([axis, skipna])**
Return cumulative minimum over requested axis.

**Series.cumprod([axis, skipna])**
Return cumulative product over requested axis.

**Series.cumsum([axis, skipna])**
Return cumulative sum over requested axis.

**Series.describe([percentiles, include, exclude])**
Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.

**Series.diff([periods])**
1st discrete difference of object

**Series.factorize([sort, na_sentinel])**
Encode the object as an enumerated type or categorical variable.

**Series.kurt([axis, skipna, level, numeric_only])**
Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).

**Series.mad([axis, skipna, level])**
Return the mean absolute deviation of the values for the requested axis.

**Series.max([axis, skipna, level, numeric_only])**
This method returns the maximum of the values in the object.

**Series.mean([axis, skipna, level, numeric_only])**
Return the mean of the values for the requested axis.

**Series.median([axis, skipna, level, ...])**
Return the median of the values for the requested axis.

**Series.min([axis, skipna, level, numeric_only])**
This method returns the minimum of the values in the object.

**Series.mode()**
Return the mode(s) of the dataset.

**Series.nlargest([n, keep])**
Return the largest n elements.

**Series.nsmallest([n, keep])**
Return the smallest n elements.

**Series pct_change([periods, fill_method, ...])**
Percent change over given number of periods.

**Series.prod([axis, skipna, level, numeric_only])**
Return the product of the values for the requested axis.

**Series.quantile([iq, interpolation])**
Return value at the given quantile, a la numpy.percentile.

**Series.rank([axis, method, numeric_only, ...])**
Compute numerical data ranks (1 through n) along axis.

**Series sem([axis, skipna, level, ddof, ...])**
Return unbiased standard error of the mean over requested axis.

**Series.skew([axis, skipna, level, numeric_only])**
Return unbiased skew over requested axis
Table 34.31 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.std</td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td>Series.sum</td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td>Series.var</td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td>Series.unique()</td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td>Series.nunique</td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td>Series.is_monotonic</td>
<td>Return boolean if values in the object are unique.</td>
</tr>
<tr>
<td>Series.is_monotonic_increasing</td>
<td>Return boolean if values in the object are increasing.</td>
</tr>
<tr>
<td>Series.is_monotonic_decreasing</td>
<td>Return boolean if values in the object are decreasing.</td>
</tr>
<tr>
<td>Series.value_counts</td>
<td>Returns object containing counts of unique values.</td>
</tr>
</tbody>
</table>

34.3.8 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.align</td>
<td>Align two objects on their axes with the</td>
</tr>
<tr>
<td>Series.drop</td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td>Series.drop_duplicates</td>
<td>Return Series with duplicate values removed</td>
</tr>
<tr>
<td>Series.duplicated</td>
<td>Return boolean Series denoting duplicate values</td>
</tr>
<tr>
<td>Series.equals</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td>Series.first</td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>Series.head</td>
<td>Return the first n rows.</td>
</tr>
<tr>
<td>Series.idxmax</td>
<td>Index label of the first occurrence of maximum of values.</td>
</tr>
<tr>
<td>Series.idxmin</td>
<td>Index label of the first occurrence of minimum of values.</td>
</tr>
<tr>
<td>Series.isin</td>
<td>Return a boolean Series showing whether each element in the Series is exactly contained in the passed sequence of values.</td>
</tr>
<tr>
<td>Series.last</td>
<td>Convenience method for subsetting final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>Series.reindex</td>
<td>Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td>Series.reindex_like</td>
<td>Return an object with matching indices to myself.</td>
</tr>
<tr>
<td>Series.rename</td>
<td>Alter Series index labels or name</td>
</tr>
<tr>
<td>Series.rename_axis</td>
<td>Alter the name of the index or columns.</td>
</tr>
<tr>
<td>Series.reset_index</td>
<td>Analogous to the pandas.DataFrame.reset_index() function, see docstring there.</td>
</tr>
<tr>
<td>Series.sample</td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td>Series.select</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td>Series.set_axis</td>
<td>Assign desired index to given axis</td>
</tr>
<tr>
<td>Series.take</td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td>Series.tail</td>
<td>Return the last n rows.</td>
</tr>
<tr>
<td>Series.truncate</td>
<td>Truncates a sorted DataFrame/Series before and/or after some particular index value.</td>
</tr>
<tr>
<td>Series.where</td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.32 – continued from previous page

```
Series.mask(cond[, other, inplace, axis, ...]) Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.
```

### 34.3.9 Missing data handling

```
Series.dropna([axis, inplace]) Return Series without null values
Series.fillna([value, method, axis, ...]) Fill NA/NaN values using the specified method
Series.interpolate([method, axis, limit, ...]) Interpolate values according to different methods.
```

### 34.3.10 Reshaping, sorting

```
Series.argsort([axis, kind, order]) Overrides ndarray.argsort.
Series.reorder_levels(order) Rearrange index levels using input order.
Series.sort_values([axis, ascending, ...]) Sort by the values along either axis
Series.sort_index([axis, level, ascending, ...]) Sort object by labels (along an axis)
Series.swaplevel([i, j, copy]) Swap levels i and j in a MultiIndex
Series.unstack([level, fill_value]) Unstack, a.k.a.
Series.searchsorted(value[, side, sorter]) Find indices where elements should be inserted to maintain order.
```

### 34.3.11 Combining / joining / merging

```
Series.append(to_append[, ignore_index, ...]) Concatenate two or more Series.
Series.replace([to_replace, value, inplace, ...]) Replace values given in ‘to_replace’ with ‘value’.
Series.update(other) Modify Series in place using non-NA values from passed Series.
```

### 34.3.12 Time series-related

```
Series.asfreq(freq[, method, how, ...]) Convert TimeSeries to specified frequency.
Series.asof(where[, subset]) The last row without any NaN is taken (or the last row without
Series.shift([periods, freq, axis]) Shift index by desired number of periods with an optional time freq
Series.first_valid_index() Return index for first non-NA/null value.
Series.last_valid_index() Return index for last non-NA/null value.
Series.resample(rule[, how, axis, ...]) Convenience method for frequency conversion and resampling of time series.
Series.tz_convert(tz[, axis, level, copy]) Convert tz-aware axis to target time zone.
Series.tz_localize(tz[, axis, level, copy, ...]) Localize tz-naive TimeSeries to target time zone.
```

### 34.3.13 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>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.dt.date</td>
<td>Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).</td>
</tr>
<tr>
<td>Series.dt.time</td>
<td>Returns numpy array of datetime.time.</td>
</tr>
<tr>
<td>Series.dt.year</td>
<td>The year of the datetime</td>
</tr>
<tr>
<td>Series.dt.month</td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td>Series.dt.day</td>
<td>The days of the datetime</td>
</tr>
<tr>
<td>Series.dt.hour</td>
<td>The hours of the datetime</td>
</tr>
<tr>
<td>Series.dt.minute</td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td>Series.dt.second</td>
<td>The seconds of the datetime</td>
</tr>
<tr>
<td>Series.dt.microsecond</td>
<td>The microseconds of the datetime</td>
</tr>
<tr>
<td>Series.dt.nanosecond</td>
<td>The nanoseconds of the datetime</td>
</tr>
<tr>
<td>Series.dt.week</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>Series.dt.weekofyear</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>Series.dt.dayofweek</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>Series.dt.weekday</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>Series.dt.dayofyear</td>
<td>The name of day in a week (ex: Friday)</td>
</tr>
<tr>
<td>Series.dt.dayofyear</td>
<td>The name of day in a week (ex: Friday)</td>
</tr>
<tr>
<td>Series.dt.is_month_start</td>
<td>Logical indicating if first day of month (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_month_end</td>
<td>Logical indicating if last day of month (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_quarter_start</td>
<td>Logical indicating if first day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_quarter_end</td>
<td>Logical indicating if last day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_year_start</td>
<td>Logical indicating if first day of year (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_year_end</td>
<td>Logical indicating if last day of year (defined by frequency)</td>
</tr>
<tr>
<td>Series.dt.is_leap_year</td>
<td>Logical indicating if the date belongs to a leap year</td>
</tr>
<tr>
<td>Series.dt.daysinmonth</td>
<td>The number of days in the month</td>
</tr>
<tr>
<td>Series.dt.days_in_month</td>
<td>The number of days in the month</td>
</tr>
<tr>
<td>Series.dt.tz</td>
<td></td>
</tr>
<tr>
<td>Series.dt.freq</td>
<td></td>
</tr>
</tbody>
</table>

#### 34.3.13.1 pandas.Series.dt.date

`Series.dt.date`  
Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).

#### 34.3.13.2 pandas.Series.dt.time

`Series.dt.time`  
Returns numpy array of datetime.time. The time part of the Timestamps.
34.3.13.3 pandas.Series.dt.year

Series.dt.year
The year of the datetime

34.3.13.4 pandas.Series.dt.month

Series.dt.month
The month as January=1, December=12

34.3.13.5 pandas.Series.dt.day

Series.dt.day
The days of the datetime

34.3.13.6 pandas.Series.dt.hour

Series.dt.hour
The hours of the datetime

34.3.13.7 pandas.Series.dt.minute

Series.dt.minute
The minutes of the datetime

34.3.13.8 pandas.Series.dt.second

Series.dt.second
The seconds of the datetime

34.3.13.9 pandas.Series.dt.microsecond

Series.dt.microsecond
The microseconds of the datetime

34.3.13.10 pandas.Series.dt.nanosecond

Series.dt.nanosecond
The nanoseconds of the datetime

34.3.13.11 pandas.Series.dt.week

Series.dt.week
The week ordinal of the year
pandas.Series.dt.weekofyear

Series.dt.weekofyear
The week ordinal of the year

pandas.Series.dt.dayofweek

Series.dt.dayofweek
The day of the week with Monday=0, Sunday=6

pandas.Series.dt.weekday

Series.dt.weekday
The day of the week with Monday=0, Sunday=6

pandas.Series.dt.weekday_name

Series.dt.weekday_name
The name of day in a week (ex: Friday)
New in version 0.18.1.

pandas.Series.dt.dayofyear

Series.dt.dayofyear
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
Logical indicating if first day of month (defined by frequency)

pandas.Series.dt.is_month_end

Series.dt.is_month_end
Logical indicating if last day of month (defined by frequency)

pandas.Series.dt.is_quarter_start

Series.dt.is_quarter_start
Logical indicating if first day of quarter (defined by frequency)
34.3.13.21 pandas.Series.dt.is_quarter_end

Series.dt.is_quarter_end
Logical indicating if last day of quarter (defined by frequency)

34.3.13.22 pandas.Series.dt.is_year_start

Series.dt.is_year_start
Logical indicating if first day of year (defined by frequency)

34.3.13.23 pandas.Series.dt.is_year_end

Series.dt.is_year_end
Logical indicating if last day of year (defined by frequency)

34.3.13.24 pandas.Series.dt.is_leap_year

Series.dt.is_leap_year
Logical indicating if the date belongs to a leap year

34.3.13.25 pandas.Series.dt.daysinmonth

Series.dt.daysinmonth
The number of days in the month

34.3.13.26 pandas.Series.dt.days_in_month

Series.dt.days_in_month
The number of days in the month

34.3.13.27 pandas.Series.dt.tz

Series.dt.tz

34.3.13.28 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 PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>Series.dt.to_pydatetime()</td>
<td></td>
</tr>
<tr>
<td>Series.dt.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive DatetimeIndex to given time zone (using</td>
</tr>
<tr>
<td>Series.dt.tz_convert(*args, **kwargs)</td>
<td>Convert tz-aware DatetimeIndex from one time zone to another (using</td>
</tr>
<tr>
<td>Series.dt.normalize(*args, **kwargs)</td>
<td>Return DatetimeIndex with times to midnight.</td>
</tr>
</tbody>
</table>

Continued on next page
### 34.3.13.29 pandas.Series.dt.to_period

Series.dt.to_period(*args, **kwargs)

Cast to PeriodIndex at a particular frequency

### 34.3.13.30 pandas.Series.dt.to_pydatetime

Series.dt.to_pydatetime()

### 34.3.13.31 pandas.Series.dt.tz_localize

Series.dt.tz_localize(*args, **kwargs)

Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

**Parameters**

- **tz**: string, pytz.timezone, dateutil.tz.tzfile or None
  - Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding local time.

- **ambiguous**: ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’
  - ‘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

- **errors**: ‘raise’, ‘coerce’, default ‘raise’
  - ‘raise’ will raise a NonExistentTimeError if a timestamp is not valid in the specified timezone (e.g. due to a transition from or to DST time)
  - ‘coerce’ will return NaT if the timestamp can not be converted into the specified timezone

New in version 0.19.0.

- **infer_dst**: boolean, default False
  - Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order

**Returns**

- **localized**: DatetimeIndex

**Raises**

- **TypeError**
  - If the DatetimeIndex is tz-aware and tz is not None.
34.3.13.32 pandas.Series.dt.tz_convert

Series.dt.tz_convert(*args, **kwargs)
    Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)

    Parameters tz : string, pytz.timezone, dateutil.tz.tzfile or None
        Time zone for time. Corresponding timestamps would be converted to time zone of
        the TimeSeries. None will remove timezone holding UTC time.

    Returns normalized : DatetimeIndex
        If DatetimeIndex is tz-naive.

34.3.13.33 pandas.Series.dt.normalize

Series.dt.normalize(*args, **kwargs)
    Return DatetimeIndex with times to midnight. Length is unaltered

    Returns normalized : DatetimeIndex

34.3.13.34 pandas.Series.dt.strftime

Series.dt.strftime(*args, **kwargs)
    Return an array 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
    New in version 0.17.0.

    Parameters date_format : str
        date format string (e.g. “%Y-%m-%d”)

    Returns ndarray of formatted strings

34.3.13.35 pandas.Series.dt.round

Series.dt.round(*args, **kwargs)
    round the index to the specified freq

    Parameters freq : freq string/object

    Returns index of same type

    Raises ValueError if the freq cannot be converted

34.3.13.36 pandas.Series.dt.floor

Series.dt.floor(*args, **kwargs)
    floor the index to the specified freq

    Parameters freq : freq string/object

    Returns index of same type

    Raises ValueError if the freq cannot be converted
34.3.13.37 pandas.Series.dt.ceil

Series.dt.ceil(*args, **kwargs)
ceil the index to the specified freq

Parameters freq : freq string/object

Returns index of same type

Raises ValueError if the freq cannot be converted

Timedelta Properties

<table>
<thead>
<tr>
<th>Series.dt.days</th>
<th>Number of days for each element.</th>
</tr>
</thead>
<tbody>
<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 (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.</td>
</tr>
</tbody>
</table>

34.3.13.38 pandas.Series.dt.days

Series.dt.days
Number of days for each element.

34.3.13.39 pandas.Series.dt.seconds

Series.dt.seconds
Number of seconds (>= 0 and less than 1 day) for each element.

34.3.13.40 pandas.Series.dt.microseconds

Series.dt.microseconds
Number of microseconds (>= 0 and less than 1 second) for each element.

34.3.13.41 pandas.Series.dt.nanoseconds

Series.dt.nanoseconds
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

34.3.13.42 pandas.Series.dt.components

Series.dt.components
Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns a DataFrame

Timedelta Methods
34.3.13.43 pandas.Series.dt.to_pytimedelta

Series.dt.to_pytimedelta()

34.3.13.44 pandas.Series.dt.total_seconds

Series.dt.total_seconds(*args, **kwargs)
Total duration of each element expressed in seconds.

New in version 0.17.0.

34.3.14 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.cat([others, sep, na_rep])</td>
<td>Concatenate strings in the Series/Index with given separator.</td>
</tr>
<tr>
<td>Series.str.center(width[, fillchar])</td>
<td>Filling left and right side of strings in the Series/Index with an additional character.</td>
</tr>
<tr>
<td>Series.str.contains(pat[, case, flags, na])</td>
<td>Return boolean Series/array whether given pattern/regex is contained in each string in the Series/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(encoding[, errors])</td>
<td>Decode character string in the Series/Index using indicated encoding.</td>
</tr>
<tr>
<td>Series.str.encode(encoding[, errors])</td>
<td>Encode character string in the Series/Index using indicated encoding.</td>
</tr>
<tr>
<td>Series.str.endswith(pat[, na])</td>
<td>Return boolean Series indicating whether each string in the Series/Index ends with passed pattern.</td>
</tr>
<tr>
<td>Series.str.extract(pat[, flags, expand])</td>
<td>For each subject string in the Series, extract groups from the first match of regular expression pat.</td>
</tr>
<tr>
<td>Series.str.extractall(pat[, flags])</td>
<td>For each subject string in the Series, extract groups from all matches of regular expression pat.</td>
</tr>
<tr>
<td>Series.str.find(sub[, start, end])</td>
<td>Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].</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.get(i)</td>
<td>Extract element from lists, tuples, or strings in each element in the Series/Index.</td>
</tr>
<tr>
<td>Series.str.index(sub[, start, end])</td>
<td>Return lowest indexes in each strings where the substring is fully contained between [start:end].</td>
</tr>
<tr>
<td>Series.str.join(sep)</td>
<td>Join lists contained as elements in the Series/Index with passed delimiter.</td>
</tr>
<tr>
<td>Series.str.len()</td>
<td>Compute length of each string in the Series/Index.</td>
</tr>
</tbody>
</table>
### pandas: powerful Python data analysis toolkit, Release 0.21.0

**Table 34.41 – continued from previous page**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.str.ljust(width[, fillchar])</code></td>
<td>Filling right side of strings in the Series/Index with an additional character.</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>Strip whitespace (including newlines) from each string in the Series/Index from left side.</td>
</tr>
<tr>
<td><code>Series.str.match(pat[, case, flags, na, ...])</code></td>
<td>Determine if each string matches 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 with an additional character to specified side.</td>
</tr>
<tr>
<td><code>Series.str.partition([pat, expand])</code></td>
<td>Split the string at the first occurrence of sep, and return 3 elements containing the part before the separator, the separator itself, and the part after the separator.</td>
</tr>
<tr>
<td><code>Series.str.repeat(repeats)</code></td>
<td>Duplicate each string in the Series/Index by indicated number of times.</td>
</tr>
<tr>
<td><code>Series.str.replace(pat, repl[, n, case, flags])</code></td>
<td>Replace occurrences of pattern/regex in the Series/Index with some other string.</td>
</tr>
<tr>
<td><code>Series.str.rfind(sub[, start, end])</code></td>
<td>Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].</td>
</tr>
<tr>
<td><code>Series.str.rindex(sub[, start, end])</code></td>
<td>Return highest indexes in each strings where the substring is fully contained between [start:end].</td>
</tr>
<tr>
<td><code>Series.str.rjust(width[, fillchar])</code></td>
<td>Filling left side of strings in the Series/Index with an additional character.</td>
</tr>
<tr>
<td><code>Series.str.rpartition([pat, expand])</code></td>
<td>Split the string at the last occurrence of sep, and return 3 elements containing the part before the separator, the separator itself, and the part after the separator.</td>
</tr>
<tr>
<td><code>Series.str.rstrip([to_strip])</code></td>
<td>Strip whitespace (including newlines) from each string in the Series/Index from right side.</td>
</tr>
<tr>
<td><code>Series.str.slice([start, stop, step])</code></td>
<td>Slice substrings from each element in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.slice_replace([start, stop, repl])</code></td>
<td>Replace a slice of each string in the Series/Index with another string.</td>
</tr>
<tr>
<td><code>Series.str.split([pat, n, expand])</code></td>
<td>Split each string (a la re.split) in the Series/Index by given pattern, propagating NA values.</td>
</tr>
<tr>
<td><code>Series.str.rsplit([pat, n, expand])</code></td>
<td>Split each string in the Series/Index by the given delimiter string, starting at the end of the string and working to the front.</td>
</tr>
<tr>
<td><code>Series.str.startswith(pat[, na])</code></td>
<td>Return boolean Series/array indicating whether each string in the Series/Index starts with passed pattern.</td>
</tr>
<tr>
<td><code>Series.str.strip([to_strip])</code></td>
<td>Strip whitespace (including newlines) from each string in the Series/Index from left and right sides.</td>
</tr>
<tr>
<td><code>Series.str.swapcase()</code></td>
<td>Convert strings in the Series/Index to be swappcased.</td>
</tr>
<tr>
<td><code>Series.str.title()</code></td>
<td>Convert strings in the Series/Index to titlecased.</td>
</tr>
<tr>
<td><code>Series.str.translate(table[, deletechars])</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 long strings in the Series/Index to be formatted in paragraphs with length less than a given width.</td>
</tr>
<tr>
<td><code>Series.str.zfill(width)</code></td>
<td>Filling left side of strings in the Series/Index with 0.</td>
</tr>
<tr>
<td><code>Series.str.isalnum()</code></td>
<td>Check whether all characters in each string in the Series/Index are alphanumeric.</td>
</tr>
</tbody>
</table>

**Continued on next page**
Series[str.isalpha()] Check whether all characters in each string in the Series/Index are alphabetic.

Series[str.isdigit()] Check whether all characters in each string in the Series/Index are digits.

Series[str.isspace()] Check whether all characters in each string in the Series/Index are whitespace.

Series[str.islower()] Check whether all characters in each string in the Series/Index are lowercase.

Series[str.isupper()] Check whether all characters in each string in the Series/Index are uppercase.

Series[str.istitle()] Check whether all characters in each string in the Series/Index are titlecase.

Series[str.isnumeric()] Check whether all characters in each string in the Series/Index are numeric.

Series[str.isdecimal()] Check whether all characters in each string in the Series/Index are decimal.

Series[str.get_dummies([sep])] Split each string in the Series by sep and return a frame of dummy/indicator variables.

34.3.14.1 pandas.Series.str.capitalize

Series[str.capitalize()] Convert strings in the Series/Index to be capitalized. Equivalent to str.capitalize().

Returns converted : Series/Index of objects

34.3.14.2 pandas.Series.str.cat

Series[str.cat(others=None, sep=None, na_rep=None)] Concatenate strings in the Series/Index with given separator.

Parameters others : list-like, or list of list-likes

If None, returns str concatenating strings of the Series

sep : string or None, default None

na_rep : string or None, default None

If None, NA in the series are ignored.

Returns concat : Series/Index of objects or str

Examples

When na_rep is None (default behavior), NaN value(s) in the Series are ignored.

```python
>>> Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ')
'a b c'

>>> Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ', na_rep='?')
'a b ? c'
```
If `others` is specified, corresponding values are concatenated with the separator. Result will be a Series of strings.

```python
>>> Series(['a', 'b', 'c']).str.cat(['A', 'B', 'C'], sep=',')
0 a,A
1 b,B
2 c,C
dtype: object
```

Otherwise, strings in the Series are concatenated. Result will be a string.

```python
>>> Series(['a', 'b', 'c']).str.cat(sep=',')
a,b,c
```

Also, you can pass a list of list-likes.

```python
>>> Series(['a', 'b']).str.cat([['x', 'y'], ['1', '2']], sep=',')
0 a,x,1
1 b,y,2
dtype: object
```

### 34.3.14.3 pandas.Series.str.center

Series.str.center(width, fillchar=' ')

Filling left and right side of strings in the Series/Index with an additional character. 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

### 34.3.14.4 pandas.Series.str.contains

Series.str.contains(pat, case=True, flags=0, na=nan, regex=True)

Return boolean Series/array whether given pattern/regex is contained in each string in the Series/Index.

**Parameters**
- `pat`: string
  - Character sequence or regular expression
- `case`: boolean, default True
  - If True, case sensitive
- `flags`: int, default 0 (no flags)
  - re module flags, e.g. re.IGNORECASE
- `na`: default NaN, fill value for missing values.
- `regex`: bool, default True
  - If True use re.search, otherwise use Python in operator
Returns contained : Series/array of boolean values

See also:

match analogous, but stricter, relying on re.match instead of re.search

### 34.3.14.5 pandas.Series.str.count

Series.str.count (pat, flags=0, **kwargs)

Count occurrences of pattern in each string of the Series/Index.

**Parameters**
- pat : string, valid regular expression
- flags : int, default 0 (no flags)
  - re module flags, e.g. re.IGNORECASE

**Returns**
- counts : Series/Index of integer values

### 34.3.14.6 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**
- decoded : Series/Index of objects

### 34.3.14.7 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

### 34.3.14.8 pandas.Series.str.endswith

Series.str.endswith (pat, na=nan)

Return boolean Series indicating whether each string in the Series/Index ends with passed pattern. Equivalent to str.endswith().

**Parameters**
- pat : string
  - Character sequence
- na : bool, default NaN

**Returns**
- endswith : Series/array of boolean values
34.3.14.9 pandas.Series.str.extract

Series.str.extract(pat, flags=0, expand=None)
For each subject string in the Series, extract groups from the first match of regular expression pat.

Parameters pat : string
    Regular expression pattern with capturing groups
flags : int, default 0 (no flags)
    re module flags, e.g. re.IGNORECASE
expand : bool, default False
    • If True, return DataFrame.
    • If False, return Series/Index/DataFrame.
    New in version 0.18.0.

Returns 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 = Series(['a1', 'b2', 'c3'])
>>> s.str.extract('([ab])(\d)')
          0  1
0         a  1
1         b  2
2        NaN NaN
```

A pattern may contain optional groups.

```python
>>> s.str.extract('([ab])?\d')
          0  1
0         a  1
1         b  2
2        NaN NaN
```

Named groups will become column names in the result.
A pattern with one group will return a DataFrame with one column if expand=True.

```
>>> s.str.extract('(?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 Series if expand=False.

```
>>> s.str.extract('[ab](\d)', expand=True)
   0
 0  1
 1  2
 2  NaN
```

dtype: object

34.3.14.10 pandas.Series.str.extractall

Series.str.extractall(pat, flags=0)

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).

New in version 0.18.0.

Parameters:

- **pat** : string
  - Regular expression pattern with capturing groups
- **flags** : int, default 0 (no flags)
  - re module flags, e.g. re.IGNORECASE

Returns:

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 indicates the order in the subject. 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.
>>> s = Series(["ala2", "bl", "cl"], index=["A", "B", "C"])
>>> s.str.extractall("[ab](\d)")
          0
match  
A     0  1
   1  2
B     0  1

Capture group names are used for column names of the result.

>>> s.str.extractall("[ab]\(?P<digit>\d)\")
      digit
match  
A      0  1
   1  2
B      0  1

A pattern with two groups will return a DataFrame with two columns.

>>> s.str.extractall("\(?P<letter>[ab]\)?\(?P<digit>\d)\")
          letter  digit
match  
A      0   a  1
   1   a  2
B      0   b  1
C      0  NaN  1

Optional groups that do not match are NaN in the result.

34.3.14.11 pandas.Series.str.find

Series.str.find(sub, start=0, end=None)

Return lowest indexes in each strings in the Series/Index 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

- **found**: Series/Index of integer values

See also:

- rfind Return highest indexes in each strings
34.3.14.12 pandas.Series.str.findall

Series.str.findall(pat, flags=0, **kwargs)
Find all occurrences of pattern or regular expression in the Series/Index. Equivalent to re.findall().

Parameters pat : string
    Pattern or regular expression
flags : int, default 0 (no flags)
    re module flags, e.g. re.IGNORECASE

Returns matches : Series/Index of lists

See also:
extractall returns DataFrame with one column per capture group

34.3.14.13 pandas.Series.str.get

Series.str.get(i)
Extract element from lists, tuples, or strings in each element in the Series/Index.

Parameters i : int
    Integer index (location)

Returns items : Series/Index of objects

34.3.14.14 pandas.Series.str.index

Series.str.index(sub, start=0, end=None)
Return lowest indexes in each strings 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 found : Series/Index of objects

See also:

rindex Return highest indexes in each strings

34.3.14.15 pandas.Series.str.join

Series.str.join(sep)
Join lists contained as elements in the Series/Index with passed delimiter. Equivalent to str.join().

Parameters sep : string
Delimiter

Returns joined: Series/Index of objects

### 34.3.14.16 pandas.Series.str.len

Series.str.len()
Compute length of each string in the Series/Index.

Returns lengths: Series/Index of integer values

### 34.3.14.17 pandas.Series.str.ljust

Series.str.ljust(width, fillchar=' ')
Filling right side of strings in the Series/Index with an additional character. 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

### 34.3.14.18 pandas.Series.str.lower

Series.str.lower()
Convert strings in the Series/Index to lowercase. Equivalent to str.lower().

Returns converted: Series/Index of objects

### 34.3.14.19 pandas.Series.str.lstrip

Series.str.lstrip(to_strip=None)
Strip whitespace (including newlines) from each string in the Series/Index from left side. Equivalent to str.lstrip().

Returns stripped: Series/Index of objects

### 34.3.14.20 pandas.Series.str.match

Series.str.match(pat, case=True, flags=0, na=nan, as_indexer=None)
Determine if each string matches a regular expression.

Parameters pat: string
Character sequence or regular expression

case: boolean, default True
If True, case sensitive

flags: int, default 0 (no flags)
re module flags, e.g. re.IGNORECASE
na : default NaN, fill value for missing values.

as_indexer : DEPRECATED - Keyword is ignored.

Returns Series/array of boolean values

See also:

contains analogous, but less strict, relying on re.search instead of re.match
extract extract matched groups

34.3.14.21 pandas.Series.str.normalize

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

34.3.14.22 pandas.Series.str.pad

Series.str.pad(width, side='left', fillchar=' ')
Pad strings in the Series/Index with an additional character to specified side.

Parameters width : int
Minimum width of resulting string; additional characters will be filled with spaces
side : {'left', 'right', 'both'}, default 'left'
fillchar : str
Additional character for filling, default is whitespace

Returns padded : Series/Index of objects

34.3.14.23 pandas.Series.str.partition

Series.str.partition(pat=' ', expand=True)
Split the string at the first occurrence of sep, and return 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 pat : string, default whitespace
String to split on.
expand : bool, default True
• If True, return DataFrame/MultiIndex expanding dimensionality.
• If False, return Series/Index.

Returns split : DataFrame/MultiIndex or Series/Index of objects

See also:
**rpartition** Split the string at the last occurrence of `sep`

**Examples**

```python
>>> s = Series(['A_B_C', 'D_E_F', 'X'])
0   A_B_C
1   D_E_F
2     X
dtype: object

>>> s.str.partition('_')
0 1 2
0 A _ B_C
1 D _ E_F
2 X

>>> s.str.rpartition('_')
0 1 2
0 A_B _ C
1 D_E _ F
2 X
```

34.3.14.24 pandas.Series.str.repeat

The `Series.str.repeat(repeats)` method duplicates each string in the Series/Index by indicated number of times.

**Parameters**
- `repeats` : int or array
  - Same value for all (int) or different value per (array)

**Returns**
- `repeated` : Series/Index of objects

34.3.14.25 pandas.Series.str.replace

The `Series.str.replace(pat, repl, n=-1, case=None, flags=0)` method replaces occurrences of pattern/regex in the Series/Index with some other string. Equivalent to `str.replace()` or `re.sub()`.

**Parameters**
- `pat` : string or compiled regex
  - String can be a character sequence or regular expression.
    - New in version 0.20.0: `pat` also accepts a compiled regex.

- `repl` : string 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()`
    - New in version 0.20.0: `repl` also accepts a callable.

- `n` : int, default -1 (all)
  - Number of replacements to make from start

- `case` : boolean, default None
• 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)
• re module flags, e.g. re.IGNORECASE
• Cannot be set if pat is a compiled regex

Returns replaced : Series/Index of objects

Notes

When pat is a compiled regex, all flags should be included in the compiled regex. Use of case or flags with a compiled regex will raise an error.

Examples

When repl is a string, every pat is replaced as with str.replace(). NaN value(s) in the Series are left as is.

```python
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f', 'b')
0  boo
1  buz
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)
0  <_sre.SRE_Match object; span=(0, 1), match='f'>oo
1  <_sre.SRE_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]
>>> pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(r'[a-z]+', repl)
0  oof 123
1  rab zab
2  NaN
dtype: object
```

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()
>>> pd.Series(['One Two Three', 'Foo Bar Baz']).str.replace(pat, repl)
0  tWO
1  bAR
dtype: object
```
Using a compiled regex with flags

```python
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
```

```python
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar')
0   foo
1   bar
2  NaN
dtype: object
```

### 34.3.14.26 pandas.Series.str.rfind

Series.str.\texttt{rfind}(\texttt{sub}, \texttt{start}=0, \texttt{end}=\texttt{None})

Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard \texttt{str.rfind}().

**Parameters**

\texttt{sub} : str

Substring being searched

\texttt{start} : int

Left edge index

\texttt{end} : int

Right edge index

**Returns**

\texttt{found} : Series/Index of integer values

**See also:**

\texttt{find} Return lowest indexes in each strings

### 34.3.14.27 pandas.Series.str.rindex

Series.str.\texttt{rindex}(\texttt{sub}, \texttt{start}=0, \texttt{end}=\texttt{None})

Return highest indexes in each strings where the substring is fully contained between [start:end]. This is the same as \texttt{str.rfind} except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard \texttt{str.rindex}.

**Parameters**

\texttt{sub} : str

Substring being searched

\texttt{start} : int

Left edge index

\texttt{end} : int

Right edge index

**Returns**

\texttt{found} : Series/Index of objects

**See also:**

\texttt{index} Return lowest indexes in each strings
34.3.14.28 pandas.Series.str.rjust

Series.str.rjust(width, fillchar=' ')
Filling left side of strings in the Series/Index with an additional character. 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

34.3.14.29 pandas.Series.str.rpartition

Series.str.rpartition(pat=' ', expand=True)
Split the string at the last occurrence of sep, and return 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

pat : string, default whitespace
String to split on.

expand : bool, default True
• If True, return DataFrame/MultiIndex expanding dimensionality.
• If False, return Series/Index.

Returns

split : DataFrame/MultiIndex or Series/Index of objects

See also:

partition Split the string at the first occurrence of sep

Examples

```python
>>> s = Series(['A_B_C', 'D_E_F', 'X'])
0  A_B_C
1  D_E_F
2   X
dtype: object

>>> s.str.partition('_')
   0  1  2
0  A  _  B_C
1  D  _  E_F
2   X

>>> s.str.rpartition('_')
   0  1  2
0  A_B_  C
1  D_E_  F
2   X
```
34.3.14.30 pandas.Series.str.rstrip

Series.str.rstrip(to_strip=None)
Strip whitespace (including newlines) from each string in the Series/Index from right side. Equivalent to str.rstrip().

Returns stripped : Series/Index of objects

34.3.14.31 pandas.Series.str.slice

Series.str.slice(start=None, stop=None, step=None)
Slice substrings from each element in the Series/Index

Parameters start : int or None
stop : int or None
step : int or None

Returns sliced : Series/Index of objects

34.3.14.32 pandas.Series.str.slice_replace

Series.str.slice_replace(start=None, stop=None, repl=None)
Replace a slice of each string in the Series/Index with another string.

Parameters start : int or None
stop : int or None
repl : str or None
String for replacement

Returns replaced : Series/Index of objects

34.3.14.33 pandas.Series.str.split

Series.str.split(pat=None, n=-1, expand=False)
Split each string (a la re.split) in the Series/Index by given pattern, propagating NA values. Equivalent to str.split().

Parameters pat : string, default None
String or regular expression to split on. If None, splits on whitespace
n : int, default -1 (all)
None, 0 and -1 will be interpreted as return all splits
expand : bool, default False
• If True, return DataFrame/MultiIndex expanding dimensionality.
• If False, return Series/Index.
return_type : deprecated, use expand

Returns split : Series/Index or DataFrame/MultiIndex of objects
34.3.14.34 pandas.Series.str.rsplit

Series.str.rsplit (pat=None, n=-1, expand=False)
Split each string in the Series/Index by the given delimiter string, starting at the end of the string and working to the front. Equivalent to str.rsplit().

Parameters
- **pat**: string, default None
  - Separator to split on. If None, splits on whitespace
- **n**: int, default -1 (all)
  - None, 0 and -1 will be interpreted as return all splits
- **expand**: bool, default False
  - If True, return DataFrame/MultiIndex expanding dimensionality.
  - If False, return Series/Index.

Returns
- **split**: Series/Index or DataFrame/MultiIndex of objects

34.3.14.35 pandas.Series.str.startswith

Series.str.startswith (pat, na=nan)
Return boolean Series/array indicating whether each string in the Series/Index starts with passed pattern. Equivalent to str.startswith().

Parameters
- **pat**: string
  - Character sequence
- **na**: bool, default NaN

Returns
- **startswith**: Series/array of boolean values

34.3.14.36 pandas.Series.str.strip

Series.str.strip (to_strip=None)
Strip whitespace (including newlines) from each string in the Series/Index from left and right sides. Equivalent to str.strip().

Returns
- **stripped**: Series/Index of objects

34.3.14.37 pandas.Series.str.swapcase

Series.str.swapcase()
Convert strings in the Series/Index to be swapcased. Equivalent to str.swapcase().

Returns
- **converted**: Series/Index of objects

34.3.14.38 pandas.Series.str.title

Series.str.title()
Convert strings in the Series/Index to titlecase. Equivalent to str.title().

Returns
- **converted**: Series/Index of objects
### 34.3.14.39 pandas.Series.str.translate

```python
Series.str.translate(table, deletechars=None)
```

Map all characters in the string through the given mapping table. Equivalent to standard `str.translate()`. Note that the optional argument `deletechars` is only valid if you are using python 2. For python 3, character deletion should be specified via the table argument.

**Parameters**
- `table`: dict (python 3), str or None (python 2)
  - In python 3, 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.
  - In python 2, table is either a string of length 256 or None. If the table argument is None, no translation is applied and the operation simply removes the characters in `deletechars`. `string.maketrans()` is a helper function for making translation tables.
- `deletechars`: str, optional (python 2)
  - A string of characters to delete. This argument is only valid in python 2.

**Returns**
- `translated`: Series/Index of objects

### 34.3.14.40 pandas.Series.str.upper

```python
Series.str.upper()
```

Convert strings in the Series/Index to uppercase. Equivalent to `str.upper()`.

**Returns**
- `converted`: Series/Index of objects

### 34.3.14.41 pandas.Series.str.wrap

```python
Series.str.wrap(width, **kwargs)
```

Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given 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**: wrapped : Series/Index of objects

**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      nto be\n      wrapped
```

### 34.3.14.42 pandas.Series.str.zfill

**Series.str.zfill(width)**

Filling left side of strings in the Series/Index with 0. Equivalent to `str.zfill()`.

**Parameters**: `width` : int

Minimum width of resulting string; additional characters will be filled with 0

**Returns**: filled : Series/Index of objects

### 34.3.14.43 pandas.Series.str.isalnum

**Series.str.isalnum()**

Check whether all characters in each string in the Series/Index are alphanumeric. Equivalent to `str.isalnum()`.

**Returns**: `is` : Series/array of boolean values

### 34.3.14.44 pandas.Series.str.isalpha

**Series.str.isalpha()**

Check whether all characters in each string in the Series/Index are alphabetic. Equivalent to `str.isalpha()`.

**Returns**: `is` : Series/array of boolean values
34.3.14.45 pandas.Series.str.isdigit

Series.str.isdigit()
Check whether all characters in each string in the Series/Index are digits. Equivalent to str.isdigit().

Returns is : Series/array of boolean values

34.3.14.46 pandas.Series.str.isspace

Series.str.isspace()
Check whether all characters in each string in the Series/Index are whitespace. Equivalent to str.isspace().

Returns is : Series/array of boolean values

34.3.14.47 pandas.Series.str.islower

Series.str.islower()
Check whether all characters in each string in the Series/Index are lowercase. Equivalent to str.islower().

Returns is : Series/array of boolean values

34.3.14.48 pandas.Series.str.isupper

Series.str.isupper()
Check whether all characters in each string in the Series/Index are uppercase. Equivalent to str.isupper().

Returns is : Series/array of boolean values

34.3.14.49 pandas.Series.str.istitle

Series.str.istitle()
Check whether all characters in each string in the Series/Index are titlecase. Equivalent to str.istitle().

Returns is : Series/array of boolean values

34.3.14.50 pandas.Series.str.isnumeric

Series.str.isnumeric()
Check whether all characters in each string in the Series/Index are numeric. Equivalent to str.isnumeric().

Returns is : Series/array of boolean values

34.3.14.51 pandas.Series.str.isdecimal

Series.str.isdecimal()
Check whether all characters in each string in the Series/Index are decimal. Equivalent to str.isdecimal().

Returns is : Series/array of boolean values
34.3.14.52 pandas.Series.str.get_dummies

Series.str.get_dummies(sep='|')
Split each string in the Series by sep and return a frame of dummy/indicator variables.

**Parameters**
- `sep`: string, default “|”
  String to split on.

**Returns**
- `dummies`: DataFrame

See also:
- `pandas.get_dummies`

**Examples**

```python
>>> 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
>>> 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
```

34.3.15 Categorical

The dtype of a Categorical can be described by a `pandas.api.types.CategoricalDtype`.

```
api.types.CategoricalDtype(categories=None, ordered=False)
Type for categorical data with the categories and orderedness
```

34.3.15.1 pandas.api.types.CategoricalDtype

```
class pandas.api.types.CategoricalDtype(categories=None, ordered=False)
    Type for categorical data with the categories and orderedness
    
    Changed in version 0.21.0.
    
    **Parameters**
    - `categories`: sequence, optional
      Must be unique, and must not contain any nulls.
    - `ordered`: bool, default False

    See also:
    - `pandas.Categorical`
```
Notes

This class is useful for specifying the type of a `Categorical` independent of the values. See `CategoricalDtype` for more.

Examples

```python
>>> t = 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]
```

If the Series is of dtype `CategoricalDtype`, `Series.cat` can be used to change the categorical data. This accessor is similar to the `Series.dt` or `Series.str` and has the following usable methods and properties:

<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></td>
</tr>
</tbody>
</table>

### 34.3.15.2 pandas.Series.cat.categories

`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, reorder_categories, add_categories, remove_categories, remove_unused_categories, set_categories`

### 34.3.15.3 pandas.Series.cat.ordered

`Series.cat.ordered`

Whether the categories have an ordered relationship

### 34.3.15.4 pandas.Series.cat.codes

`Series.cat.codes`
Series.cat.rename_categories(*args, **kwargs)

Renames categories.

Parameters:
- new_categories : list-like or dict-like
  - 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. New in version 0.21.0.

Warning: Currently, Series are considered list like. In a future version of pandas they’ll be considered dict-like.

inplace : boolean (default: False)

Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns:
- cat : Categorical or None
  With inplace=False, the new categorical is returned. With inplace=True, there is no return value.

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, add_categories, remove_categories, remove_unused_categories, set_categories

Examples
```python
>>> c = 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]
```

### 34.3.15.6 pandas.Series.cat.reorder_categories

Series.cat.reorder_categories(*args, **kwargs)
Reorders 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**: boolean, optional
  Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.
- **inplace**: boolean (default: False)
  Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

- **cat**: Categorical with reordered categories or None if inplace.

**Raises**

- **ValueError**: If the new categories do not contain all old category items or any new ones

**See also**

rename_categories, add_categories, remove_categories, remove_unused_categories, set_categories

### 34.3.15.7 pandas.Series.cat.add_categories

Series.cat.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.
- **inplace**: boolean (default: False)
  Whether or not to add the categories inplace or return a copy of this categorical with added categories.

**Returns**

- **cat**: Categorical with new categories added or None if inplace.
Raises ValueError

If the new categories include old categories or do not validate as categories

See also:

rename_categories, reorder_categories, remove_categories,
remove_unused_categories, set_categories

34.3.15.8 pandas.Series.cat.remove_categories

Series.cat.remove_categories(*args, **kwargs)

Removes 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 : boolean (default: False)

Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns cat : Categorical with removed categories or None if inplace.

Raises ValueError

If the removals are not contained in the categories

See also:

rename_categories, reorder_categories, add_categories,
remove_unused_categories, set_categories

34.3.15.9 pandas.Series.cat.remove_unused_categories

Series.cat.remove_unused_categories(*args, **kwargs)

Removes categories which are not used.

Parameters inplace : boolean (default: False)

Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Returns cat : Categorical with unused categories dropped or None if inplace.

See also:

rename_categories, reorder_categories, add_categories, remove_categories, set_categories

34.3.15.10 pandas.Series.cat.set_categories

Series.cat.set_categories(*args, **kwargs)

Sets 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 simply 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 on python3, 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**: boolean (default: False)
  - Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.
- **rename**: boolean (default: False)
  - Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.
- **inplace**: boolean (default: False)
  - Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

- **cat**: Categorical with reordered categories or None if inplace.

**Raises**

- ValueError
  - If new_categories does not validate as categories

**See also:**

- rename_categories, reorder_categories, add_categories, remove_categories, remove_unused_categories

### 34.3.15.11 pandas.Series.cat.as_ordered

Series.cat.as_ordered(*args, **kwargs)

Sets the Categorical to be ordered

**Parameters**

- **inplace**: boolean (default: False)
  - Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True

### 34.3.15.12 pandas.Series.cat.as_unordered

Series.cat.as_unordered(*args, **kwargs)

Sets the Categorical to be unordered

**Parameters**

- **inplace**: boolean (default: False)
  - Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to False
To create a Series of dtype `category`, use `cat = s.astype("category")`.

The following two `Categorical` constructors are considered API but should only be used when adding ordering information or special categories is need at creation time of the categorical data:

```python
Categorical(values[, categories, ordered, ...]) Represents a categorical variable in classic R / S-plus fashion
```

### 34.3.15.13 pandas.Categorical

**class** `pandas.Categorical` *(values, categories=None, ordered=None, dtype=None, fastpath=False)*

Represents 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.

- **ordered** : boolean, (default False)
  
  Whether or not this categorical is treated as a ordered categorical. If not given, the resulting categorical will not be ordered.

- **dtype** : CategoricalDtype
  
  An instance of `CategoricalDtype` to use for this categorical

  New in version 0.21.0.

**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:**

- `pandas.api.types.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]
```

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]
>>> c.min()
'c'
```

`Categorical.from_codes` makes a Categorical type from codes and categories arrays.

### 34.3.15.14 pandas.Categorical.from_codes

classmethod `Categorical.from_codes(codes, categories[, ...])` Make a Categorical type from codes and categories arrays.

This constructor is useful if you already have codes and categories 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, integers
  - An integer array, where each integer points to a category in categories or -1 for NaN

- **categories**: index-like
  - The categories for the categorical. Items need to be unique.

- **ordered**: boolean, (default False)
  - Whether or not this categorical is treated as a ordered categorical. If not given, the resulting categorical will be unordered.

`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.
```

### 34.3.15.15 pandas.Categorical.__array__

`Categorical.__array__((dtype)=None)` The numpy array interface.

**Returns**

- **values**: numpy array
A numpy array of either the specified dtype or, if dtype=None (default), the same
dtype as categorical.categories.dtype

34.3.16 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>Series.plot((kind, ax, figsize, ....))</th>
<th>Series plotting accessor and method</th>
</tr>
</thead>
</table>

Series.plot.area(**kwds) Area plot
Series.plot.bar(**kwds) Vertical bar plot
Series.plot.barh(**kwds) Horizontal bar plot
Series.plot.box(**kwds) Boxplot
Series.plot.density(**kwds) Kernel Density Estimate plot
Series.plot.hist([bins]) Histogram
Series.plot.kde(**kwds) Kernel Density Estimate plot
Series.plot.line(**kwds) Line plot
Series.plot.pie(**kwds) Pie chart

34.3.16.1 pandas.Series.plot.area

Series.plot.area(**kwds)
Area plot
New in version 0.17.0.

Parameters **kwds : optional
Keyword arguments to pass on to pandas.Series.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.3.16.2 pandas.Series.plot.bar

Series.plot.bar(**kwds)
Vertical bar plot
New in version 0.17.0.

Parameters **kwds : optional
Keyword arguments to pass on to pandas.Series.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.3.16.3 pandas.Series.plot.barh

Series.plot.barh(**kwds)
Horizontal bar plot
New in version 0.17.0.

Parameters **kwds : optional
Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** `axes`: matplotlib.AxesSubplot or np.array of them

### 34.3.16.4 pandas.Series.plot.box

`Series.plot.box(**kwds)`

Boxplot

New in version 0.17.0.

**Parameters** `**kwds` : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** `axes`: matplotlib.AxesSubplot or np.array of them

---

### 34.3.16.5 pandas.Series.plot.density

`Series.plot.density(**kwds)`

Kernel Density Estimate plot

New in version 0.17.0.

**Parameters** `**kwds` : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** `axes`: matplotlib.AxesSubplot or np.array of them

---

### 34.3.16.6 pandas.Series.plot.hist

`Series.plot.hist(bins=10, **kwds)`

Histogram

New in version 0.17.0.

**Parameters** `bins`: integer, default 10

Number of histogram bins to be used

`**kwds` : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** `axes`: matplotlib.AxesSubplot or np.array of them

---

### 34.3.16.7 pandas.Series.plot.kde

`Series.plot.kde(**kwds)`

Kernel Density Estimate plot

New in version 0.17.0.

**Parameters** `**kwds` : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** `axes`: matplotlib.AxesSubplot or np.array of them
### 34.3.16.8 pandas.Series.plot.line

Series.plot.line(**kwds)

Line plot

New in version 0.17.0.

**Parameters** **kwds** : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** axes : matplotlib.AxesSubplot or np.array of them

### 34.3.16.9 pandas.Series.plot.pie

Series.plot.pie(**kwds)

Pie chart

New in version 0.17.0.

**Parameters** **kwds** : optional

Keyword arguments to pass on to `pandas.Series.plot()`.

**Returns** axes : matplotlib.AxesSubplot or np.array of them

Series.hist(by, ax, grid, xlabelsize, ...) 

Draw histogram of the input series using matplotlib

### 34.3.17 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.from_csv(path[, sep, parse_dates, ...])</td>
<td>Read CSV file (DEPRECATED, please use pandas.read_csv() instead).</td>
</tr>
<tr>
<td>Series.to_csv(path[, index, sep, na_rep, ...])</td>
<td>Write Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>Series.to_dict([into])</td>
<td>Convert Series to {label -&gt; value} dict or dict-like object.</td>
</tr>
<tr>
<td>Series.to_excel(excel_writer[, sheet_name, ...])</td>
<td>Write Series to an excel sheet</td>
</tr>
<tr>
<td>Series.to_frame([name])</td>
<td>Convert Series to DataFrame</td>
</tr>
<tr>
<td>Series.to_hdf(path_or_buf, key, **kwargs)</td>
<td>Write the contained data to an HDF5 file using HDFStore.</td>
</tr>
<tr>
<td>Series.to_json([path_or_buf, encoding])</td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td>Series.to_msgpack([path_or_buf, orient, ...])</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>Series.to_pickle(path[, compression, protocol])</td>
<td>Pickle (serialize) object to input file path.</td>
</tr>
<tr>
<td>Series.to_sparse([kind, fill_value])</td>
<td>Convert Series to SparseSeries</td>
</tr>
<tr>
<td>Series.to_string([buf, na_rep, ...])</td>
<td>Render a string representation of the Series</td>
</tr>
<tr>
<td>Series.to_clipboard([excel, sep])</td>
<td>Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.</td>
</tr>
<tr>
<td>Series.to_latex([buf, columns, col_space, ...])</td>
<td>Render an object to a tabular environment table.</td>
</tr>
</tbody>
</table>

### 34.3.18 Sparse
SparseSeries.to_coo([row_levels, ...])
Create a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.

SparseSeries.from_coo(A[, dense_index])
Create a SparseSeries from a scipy.sparse.coo_matrix.

34.3.18.1 pandas.SparseSeries.to_coo

SparseSeries.to_coo(row_levels=(0,), column_levels=(1,), sort_labels=False)
Create a scipy.sparse.coo_matrix from a SparseSeries 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.

Returns
- y : scipy.sparse.coo_matrix
- rows : list (row labels)
- columns : list (column labels)

Examples

```python
>>> from numpy import nan
>>> s = Series([3.0, nan, 1.0, 3.0, nan, nan])
>>> s.index = 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'])
>>> ss = s.to_sparse()
>>> A, rows, columns = ss.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>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  3.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> rows
[(1, 1), (1, 2), (2, 1)]
>>> columns
[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```
34.3.18.2 pandas.SparseSeries.from_coo

**classmethod** SparseSeries.from_coo(A, dense_index=False)

Create a SparseSeries 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** : SparseSeries

**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
t<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 = SparseSeries.from_coo(A)
>>> ss
0 2 1
 3 2
1 0 3
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

34.4 DataFrame

34.4.1 Constructor

**DataFrame([data, index, columns, dtype, copy])**

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

**pandas.DataFrame**

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with 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** : numpy ndarray (structured or homogeneous), dict, or DataFrame
```

1462 Chapter 34. API Reference
Dict can contain Series, arrays, constants, or list-like objects

**index**: Index or array-like

Index to use for resulting frame. Will default to np.arange(n) if no indexing information part of input data and no index provided

**columns**: Index or array-like

Column labels to use for resulting frame. Will default to np.arange(n) if no column labels are provided

**dtype**: dtype, default None

Data type to force. Only a single dtype is allowed. If None, infer

**copy**: boolean, default False

Copy data from inputs. Only affects DataFrame / 2d ndarray input

See also:

- `DataFrame.from_records` constructor from tuples, also record arrays
- `DataFrame.from_dict` from dicts of Series, arrays, or dicts
- `DataFrame.from_items` from sequence of (key, value) pairs

See also:

- `pandas.read_csv`, `pandas.read_table`, `pandas.read_clipboard`

**Examples**

Constructing DataFrame from a dictionary.

```python
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0    1    3
1    2    4
```

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 numpy ndarray:
```python
>> df2 = pd.DataFrame(np.random.randint(low=0, high=10, size=(5, 5)),
       columns=['a', 'b', 'c', 'd', 'e'])
>>> df2
   a  b  c  d  e
0  2  8  8  3  4
1  4  2  9  0  9
2  1  0  7  8  0
3  5  1  7  1  3
4  6  2  4  2
```

### Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td><code>at</code></td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>Return a list with the row axis labels and column axis labels as the only members.</td>
</tr>
<tr>
<td><code>blocks</code></td>
<td>Internal property, property synonym for as_blocks()</td>
</tr>
<tr>
<td><code>dtypes</code></td>
<td>Return the dtypes in this object.</td>
</tr>
<tr>
<td><code>empty</code></td>
<td>True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.</td>
</tr>
<tr>
<td><code>ftypes</code></td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<tr>
<td><code>iat</code></td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td><code>iloc</code></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><code>is_copy</code></td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td><code>ix</code></td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>Return a tuple representing the dimensionality of the DataFrame.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>number of elements in the NDFrame</td>
</tr>
<tr>
<td><code>style</code></td>
<td>Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.</td>
</tr>
<tr>
<td><code>values</code></td>
<td>Numpy representation of NDFrame</td>
</tr>
</tbody>
</table>

### pandas.DataFrame.T

DataFrame.T

Transpose index and columns

### pandas.DataFrame.at

DataFrame.at

Fast label-based scalar accessor

Similarly to `loc`, `at` provides label based scalar lookups. You can also set using these indexers.
pandas.DataFrame.axes

DataFrame.axes
Return a list with the row axis labels and column axis labels as the only members. They are returned in that order.

pandas.DataFrame.blocks

DataFrame.blocks
Internal property, property synonym for as_blocks()
Deprecated since version 0.21.0.

pandas.DataFrame.dtypes

DataFrame.dtypes
Return the dtypes in this object.

pandas.DataFrame.empty

DataFrame.empty
True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.
See also:
pandas.Series.dropna, pandas.DataFrame.dropna

Notes

If NDFrame 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

**pandas.DataFrame.ftypes**

Dataframe.ftypes

Return the ftypes (indication of sparse/dense and dtype) in this object.

**pandas.DataFrame.iat**

Dataframe.iat

Fast integer location scalar accessor.

Similarly to iloc, iat provides integer based lookups. You can also set using these indexers.

**pandas.DataFrame.iloc**

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, DataFrame or Panel) and that returns valid output for indexing (one of the above)

.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

**pandas.DataFrame.is_copy**

Dataframe.is_copy = None

**pandas.DataFrame.ix**

Dataframe.ix

A primarily label-location based indexer, with integer position fallback.

.ix[] supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.
.ix is the most general indexer and will support any of the inputs in .loc and .iloc. .ix also supports floating point label schemes. .ix is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it’s usually better to be explicit and use .iloc or .loc.

See more at Advanced Indexing.

**pandas.DataFrame.loc**

DataFrame.loc

Purely label-location based indexer for selection by label.

.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'].
- A slice object with labels, e.g. 'a':'f' (note that contrary to usual python slices, both the start and the stop are included!).
- A boolean array.
- A **callable** function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

.loc will raise a KeyError when the items are not found.

See more at Selection by Label

**pandas.DataFrame.ndim**

DataFrame.ndim

Number of axes / array dimensions

**pandas.DataFrame.shape**

DataFrame.shape

Return a tuple representing the dimensionality of the DataFrame.

**pandas.DataFrame.size**

DataFrame.size

number of elements in the NDFrame

**pandas.DataFrame.style**

DataFrame.style

Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.
See also:

pandas.io.formats.style.Styler

pandas.DataFrame.values

DataFrame.values

Numpy representation of NDFrame

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.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abs</strong>()</td>
<td>Return an object with absolute value taken—only applicable to objects that are all numeric.</td>
</tr>
<tr>
<td><strong>add</strong>(other[, axis, level, fill_value])</td>
<td>Addition of dataframe and other, element-wise (binary operator add).</td>
</tr>
<tr>
<td><strong>add_prefix</strong>(prefix)</td>
<td>Concatenate prefix string with panel items names.</td>
</tr>
<tr>
<td><strong>add_suffix</strong>(suffix)</td>
<td>Concatenate suffix string with panel items names.</td>
</tr>
<tr>
<td><strong>agg</strong>(func[, axis])</td>
<td>Aggregate using callable, string, dict, or list of string/callables</td>
</tr>
<tr>
<td><strong>aggregate</strong>(func[, axis])</td>
<td>Aggregate using callable, string, dict, or list of string/callables</td>
</tr>
<tr>
<td><strong>align</strong>(other[, join, axis, level, copy, ...])</td>
<td>Align two objects on their axes with the</td>
</tr>
<tr>
<td><strong>all</strong>(axis, bool_only, skipna, level)</td>
<td>Return whether all elements are True over requested axis</td>
</tr>
<tr>
<td><strong>any</strong>(axis, bool_only, skipna, level)</td>
<td>Return whether any element is True over requested axis</td>
</tr>
<tr>
<td><strong>append</strong>(other[, ignore_index, verify_integrity])</td>
<td>Append rows of other to the end of this frame, returning a new object.</td>
</tr>
<tr>
<td><strong>apply</strong>(func[, axis, broadcast, raw, reduce, args])</td>
<td>Applies function along input axis of DataFrame.</td>
</tr>
<tr>
<td><strong>applymap</strong>(func)</td>
<td>Apply a function to a DataFrame that is intended to operate elementwise, i.e.</td>
</tr>
<tr>
<td><strong>as_blocks</strong>([copy])</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has a homogeneous dtype.</td>
</tr>
<tr>
<td><strong>as_matrix</strong>([columns])</td>
<td>Convert the frame to its Numpy-array representation.</td>
</tr>
<tr>
<td><strong>asfreq</strong>(freq[, method, how, normalize, ...])</td>
<td>Convert TimeSeries to specified frequency.</td>
</tr>
<tr>
<td><strong>asof</strong>(where[, subset])</td>
<td>The last row without any NaN is taken (or the last row without</td>
</tr>
<tr>
<td><strong>assign</strong>(**kwargs)</td>
<td>Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.</td>
</tr>
<tr>
<td><strong>astype</strong>(dtype[, copy, errors])</td>
<td>Cast a pandas object to a specified dtype dtype.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>at_time(time[, asof])</code></td>
<td>Select values at particular time of day (e.g., 9:00-9:30 AM).</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>fillna(method='bfill')</code>.</td>
</tr>
<tr>
<td><code>bool()</code></td>
<td>Return the bool of a single element PandasObject.</td>
</tr>
<tr>
<td><code>boxplot([column, by, ax, fontsize, rot, ...])</code></td>
<td>Make a box plot from DataFrame column optionally grouped by some columns or</td>
</tr>
<tr>
<td><code>clip([lower, upper, axis, inplace])</code></td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><code>clip_lower(threshold[, axis, inplace])</code></td>
<td>Return copy of the input with values below given value(s) truncated.</td>
</tr>
<tr>
<td><code>clip_upper(threshold[, axis, inplace])</code></td>
<td>Return copy of input with values above given value(s) truncated.</td>
</tr>
<tr>
<td><code>combine(other, func[, fill_value, overwrite])</code></td>
<td>Add two DataFrame objects and do not propagate NaN values, so if for a</td>
</tr>
<tr>
<td><code>combine_first(other)</code></td>
<td>Combine two DataFrame objects and default to non-null values in frame calling the method.</td>
</tr>
<tr>
<td><code>compound([axis, skipna, level])</code></td>
<td>Return the compound percentage of the values for the requested axis</td>
</tr>
<tr>
<td><code>consolidate([inplace])</code></td>
<td>DEPRECATED: consolidate will be an internal implementation only.</td>
</tr>
<tr>
<td><code>convert_objects([convert_dates, ...])</code></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Make a copy of this objects 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])</code></td>
<td>Compute pairwise correlation between rows or columns of two DataFrame objects.</td>
</tr>
<tr>
<td><code>count([axis, level, numeric_only])</code></td>
<td>Return Series with number of non-NA/null observations over requested axis.</td>
</tr>
<tr>
<td><code>cov([min_periods])</code></td>
<td>Compute pairwise covariance of columns, excluding NA/null values</td>
</tr>
<tr>
<td><code>cummax([axis, skipna])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>cummin([axis, skipna])</code></td>
<td>Return cumulative minimum over requested axis.</td>
</tr>
<tr>
<td><code>cumprod([axis, skipna])</code></td>
<td>Return cumulative product over requested axis.</td>
</tr>
<tr>
<td><code>cumsum([axis, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td><code>describe([percentiles, include, exclude])</code></td>
<td>Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.</td>
</tr>
<tr>
<td><code>diff([periods, axis])</code></td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td><code>div(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>divide(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>dot(other)</code></td>
<td>Matrix multiplication with DataFrame or Series objects</td>
</tr>
<tr>
<td><code>drop([labels, axis, index, columns, level, ...])</code></td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td><code>drop_duplicates([subset, keep, inplace])</code></td>
<td>Return DataFrame with duplicate rows removed, optionally only.</td>
</tr>
<tr>
<td><code>dropna([axis, how, thresh, subset, inplace])</code></td>
<td>Return object with labels on given axis omitted where alternately any</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicated([subset, keep])</td>
<td>Return boolean Series denoting duplicate rows, optionally only</td>
</tr>
<tr>
<td>eq(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods eq</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td>eval(expr[, inplace])</td>
<td>Evaluate an expression in the context of the calling DataFrame instance.</td>
</tr>
<tr>
<td>ewm([com, span, halflife, alpha, ...])</td>
<td>Provides exponential weighted functions</td>
</tr>
<tr>
<td>expanding([min_periods, freq, center, axis])</td>
<td>Provides expanding transformations.</td>
</tr>
<tr>
<td>ffill([axis, inplace, limit, downcast()])</td>
<td>Synonym for DataFrame. fillna(method='ffill')</td>
</tr>
<tr>
<td>fillna([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td>filter([items, like, regex, axis])</td>
<td>Subset rows or columns of dataframe according to labels in the specified index.</td>
</tr>
<tr>
<td>first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>first_valid_index()</td>
<td>Return index for first non-NA/null value.</td>
</tr>
<tr>
<td>floordiv(other[, axis, level, fill_value])</td>
<td>Integer division of dataframe and other, element-wise (binary operator floordiv).</td>
</tr>
<tr>
<td>from_csv(path[, header, sep, index_col, ...])</td>
<td>Read CSV file (DEPRECATED, please use pandas.read_csv() instead).</td>
</tr>
<tr>
<td>from_dict(data[, orient, dtype])</td>
<td>Construct DataFrame from dict of array-like or dicts</td>
</tr>
<tr>
<td>from_items(items[, columns, orient])</td>
<td>Convert (key, value) pairs to DataFrame.</td>
</tr>
<tr>
<td>from_records(data[, index, exclude, ...])</td>
<td>Convert structured or record ndarray to DataFrame</td>
</tr>
<tr>
<td>ge(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods ge</td>
</tr>
<tr>
<td>get(key[, default])</td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.).</td>
</tr>
<tr>
<td>get_dtype_counts()</td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>get_ftype_counts()</td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td>get_value(index, col[, takeable])</td>
<td>Quickly retrieve single value at passed column and index</td>
</tr>
<tr>
<td>get_values()</td>
<td>same as values (but handles sparseness conversions)</td>
</tr>
<tr>
<td>groupby([by, axis, level, as_index, sort, ...])</td>
<td>Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.</td>
</tr>
<tr>
<td>gt(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods gt</td>
</tr>
<tr>
<td>head(n)</td>
<td>Return the first n rows.</td>
</tr>
<tr>
<td>hist(data[, column, by, grid, xlabels, ...])</td>
<td>Draw histogram of the DataFrame’s series using matplotlib / pylab.</td>
</tr>
<tr>
<td>idxmax([axis, skipna])</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td>idxmin([axis, skipna])</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([verbose, buf, max_cols, memory_usage, ...])</td>
<td>Concise summary of a DataFrame.</td>
</tr>
<tr>
<td>insert(loc, column, value[, allow_duplicates])</td>
<td>Insert column into DataFrame at specified location.</td>
</tr>
<tr>
<td>interpolate([method, axis, limit, inplace, ...])</td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td>isin(values)</td>
<td>Return boolean DataFrame showing whether each element in the DataFrame is contained in values.</td>
</tr>
</tbody>
</table>
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isna()</td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td>isnull()</td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td>items()</td>
<td>Iterator over (column name, Series) pairs.</td>
</tr>
<tr>
<td>iteritems()</td>
<td>Iterator over (column name, Series) pairs.</td>
</tr>
<tr>
<td>iterrows()</td>
<td>Iterate over DataFrame rows as (index, Series) pairs.</td>
</tr>
<tr>
<td>itertuples()</td>
<td>Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.</td>
</tr>
<tr>
<td>join()</td>
<td>Join columns with other DataFrame either on index or on a key column.</td>
</tr>
<tr>
<td>keys()</td>
<td>Get the ‘info axis’ (see Indexing for more)</td>
</tr>
<tr>
<td>kurt([axis, skipna, level, numeric_only])</td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td>kurtosis([axis, skipna, level, numeric_only])</td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td>last(offset)</td>
<td>Convenience method for subsetting final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>last_valid_index()</td>
<td>Return index for first non-NA/null value.</td>
</tr>
<tr>
<td>le(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods le</td>
</tr>
<tr>
<td>lookup(row_labels, col_labels)</td>
<td>Label-based “fancy indexing” function for DataFrame.</td>
</tr>
<tr>
<td>lt(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods lt</td>
</tr>
<tr>
<td>mad([axis, skipna, level])</td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td>mask(cond[, other, inplace, axis, level, ...])</td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.</td>
</tr>
<tr>
<td>max([axis, skipna, level, numeric_only])</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>mean([axis, skipna, level, numeric_only])</td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td>median([axis, skipna, level, numeric_only])</td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td>melt([id_vars, value_vars, var_name, ...])</td>
<td>“Unpivots” a DataFrame from wide format to long format, optionally</td>
</tr>
<tr>
<td>memory_usage([index, deep])</td>
<td>Memory usage of DataFrame columns.</td>
</tr>
<tr>
<td>merge(right[, how, on, left_on, right_on, ...])</td>
<td>Merge DataFrame objects by performing a database-style join operation by columns or indexes.</td>
</tr>
<tr>
<td>min([axis, skipna, level, numeric_only])</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td>mod(other[, axis, level, fill_value])</td>
<td>Modulo of dataframe and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td>mode([axis, numeric_only])</td>
<td>Gets the mode(s) of each element along the axis selected.</td>
</tr>
<tr>
<td>mul(other[, axis, level, fill_value])</td>
<td>Multiplication of dataframe and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td>multiply(other[, axis, level, fill_value])</td>
<td>Multiplication of dataframe and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td>ne(other[, axis, level])</td>
<td>Wrapper for flexible comparison methods ne</td>
</tr>
<tr>
<td>nlargest(n, columns[, keep])</td>
<td>Get the rows of a DataFrame sorted by the n largest values of columns.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>notna()</code></td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
<tr>
<td><code>nsmallest(n, columns[, keep])</code></td>
<td>Get the rows of a DataFrame sorted by the n smallest values of columns.</td>
</tr>
<tr>
<td><code>nunique([axis, dropna])</code></td>
<td>Return Series with number of distinct observations over requested axis.</td>
</tr>
<tr>
<td><code>pct_change([periods, fill_method, limit, freq])</code></td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td><code>pipe(func, *args, **kwargs)</code></td>
<td>Apply func(self, *args, **kwargs)</td>
</tr>
<tr>
<td><code>pivot([index, columns, values])</code></td>
<td>Reshape data (produce a “pivot” table) based on 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 FramePlotMethods</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>Exponential power of dataframe and other, element-wise (binary operator pow).</td>
</tr>
<tr>
<td><code>prod([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>product([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for 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, a la numpy.percentile.</td>
</tr>
<tr>
<td><code>query(expr[, inplace])</code></td>
<td>Query the columns of a frame with a boolean expression.</td>
</tr>
<tr>
<td><code>radd(other[, axis, level, fill_value])</code></td>
<td>Addition of dataframe and other, element-wise (binary operator radd).</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>Floating division of dataframe and other, element-wise (binary operator rtruediv).</td>
</tr>
<tr>
<td><code>reindex([labels, index, columns, axis, ...])</code></td>
<td>Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><code>reindex_axis(labels[, axis, method, level, ...])</code></td>
<td>Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit, ...])</code></td>
<td>Return an object with matching indices to myself.</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[, axis, copy, inplace])</code></td>
<td>Alter the name of the index or columns.</td>
</tr>
<tr>
<td><code>reorder_levels(order[, axis])</code></td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td><code>replace([to_replace, value, inplace, limit, ...])</code></td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td><code>resample(rule[, how, axis, fill_method, ...])</code></td>
<td>Convenience method for frequency conversion and resampling of time series.</td>
</tr>
<tr>
<td><code>reset_index([level, drop, inplace, ...])</code></td>
<td>For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to ‘level_0’, ‘level_1’, etc.</td>
</tr>
<tr>
<td><code>rfloordiv(other[, axis, level, fill_value])</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator rfloordiv).</td>
</tr>
<tr>
<td><code>rmod(other[, axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator rmod).</td>
</tr>
<tr>
<td><code>rmul(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator rmul).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rolling(window[, min_periods, freq, center, ...])</code></td>
<td>Provides rolling window calculations.</td>
</tr>
<tr>
<td><code>round([decimals])</code></td>
<td>Round a DataFrame to a variable number of decimal places.</td>
</tr>
<tr>
<td><code>rpow(other[, axis, level, fill_value])</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>rsub(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>rtruediv(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>sample([n, frac, replace, weights, ...])</code></td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><code>select(crit[, axis])</code></td>
<td>Return data corresponding to axis labels matching criteria.</td>
</tr>
<tr>
<td><code>select_dtypes([include, exclude])</code></td>
<td>Return a subset of a DataFrame including/excluding columns based on their <code>dtype</code>.</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_index(keys[, drop, append, inplace, ...])</code></td>
<td>Set the DataFrame index (row labels) using one or more existing columns.</td>
</tr>
<tr>
<td><code>set_value(index, col, value[, takeable])</code></td>
<td>Put single value at passed column and index</td>
</tr>
<tr>
<td><code>shift([periods, freq, axis])</code></td>
<td>Shift index by desired number of periods with an optional time freq</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>Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>sort_index([axis, level, ascending, ...])</code></td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td><code>sort_values(by[, axis, ascending, inplace, ...])</code></td>
<td>Sort by the values along either axis</td>
</tr>
<tr>
<td><code>sortlevel([level, axis, ascending, inplace, ...])</code></td>
<td>DEPRECATED: use <code>DataFrame.sort_index()</code></td>
</tr>
<tr>
<td><code>squeeze(axis)[]</code></td>
<td>Squeeze length 1 dimensions.</td>
</tr>
<tr>
<td><code>stack([level, dropna])</code></td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.</td>
</tr>
<tr>
<td><code>std([axis, skipna, level, ddof, numeric_only])</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>subtract(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2[, copy])</code></td>
<td>Interchange axes and swap values axes appropriately</td>
</tr>
<tr>
<td><code>swaplevel([i, j, axis])</code></td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
<tr>
<td><code>tail([n])</code></td>
<td>Return the last n rows.</td>
</tr>
<tr>
<td><code>take(indices[, axis, convert, is_copy])</code></td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><code>to_clipboard([excel, sep])</code></td>
<td>Attempt to write text representation of object to the system clipboard. This can be pasted into Excel, for example.</td>
</tr>
<tr>
<td><code>to_csv([path_or_buf, sep, na_rep, ...])</code></td>
<td>Write DataFrame to a comma-separated values (csv) file.</td>
</tr>
<tr>
<td><code>to_dense()</code></td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.55 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_dict([orient, into])</code></td>
<td>Convert DataFrame to dictionary.</td>
</tr>
<tr>
<td><code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code></td>
<td>Write DataFrame to an excel sheet</td>
</tr>
<tr>
<td><code>to_feather(fname)</code></td>
<td>Write out the binary feather-format for DataFrames</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, **kwargs)</code></td>
<td>Write the contained data to an HDF5 file using HDFS-tore.</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 an object to a tabular environment table.</td>
</tr>
<tr>
<td><code>to_msgpack([path_or_buf, encoding])</code></td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_panel()</code></td>
<td>Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.</td>
</tr>
<tr>
<td><code>to_parquet(fname[, engine, compression])</code></td>
<td>Write a DataFrame to the binary parquet format.</td>
</tr>
<tr>
<td><code>to_period([Freq, axis, copy])</code></td>
<td>Convert a DataFrame from DatetimeIndex to PeriodIndex with desired</td>
</tr>
<tr>
<td><code>to_pickle(path[, compression, protocol])</code></td>
<td>Pickle (serialize) object to input file path.</td>
</tr>
<tr>
<td><code>to_records([index, convert_datetime64])</code></td>
<td>Convert DataFrame to record array.</td>
</tr>
<tr>
<td><code>to_sparse([fill_value, kind])</code></td>
<td>Convert to SparseDataFrame</td>
</tr>
<tr>
<td><code>to_sql(name, con[, flavor, schema, ...])</code></td>
<td>Write records stored in a DataFrame to a SQL database.</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>transform(func, *args, **kwargs)</code></td>
<td>Call function producing a like-indexed NDFrame</td>
</tr>
<tr>
<td><code>transpose(*args, **kwargs)</code></td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td><code>truediv(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td><code>truncate([before, after, axis, copy])</code></td>
<td>Truncates a sorted DataFrame/Series before and/or after some particular index value.</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, ambiguous])</code></td>
<td>Localize tz-naive TimeSeries 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, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.</td>
</tr>
<tr>
<td><code>update(other[, join, overwrite, ...])</code></td>
<td>Modify DataFrame in place using non-NA values from passed 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>Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.</td>
</tr>
<tr>
<td><code>xs(key[, axis, level, drop_level])</code></td>
<td>Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.</td>
</tr>
</tbody>
</table>
**pandas.DataFrame.abs**

DataFrame.abs()  
Return an object with absolute value taken—only applicable to objects that are all numeric.  

Returns abs: type of caller

**pandas.DataFrame.add**

DataFrame.add(other, axis='columns', level=None, fill_value=None)  
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.  

Parameters other : Series, DataFrame, or constant  
axis : {0, 1, ‘index’, ‘columns’}  
For Series input, axis to match Series index on  
fill_value : None or float value, default None  
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing  
level : int or name  
Broadcast across a level, matching Index values on the passed MultiIndex level  

Returns result : DataFrame

See also:  
DataFrame.radd

Notes  
Mismatched indices will be unioned together

**pandas.DataFrame.add_prefix**

DataFrame.add_prefix(prefix)  
Concatenate prefix string with panel items names.  

Parameters prefix : string  

Returns with_prefix : type of caller

**pandas.DataFrame.add_suffix**

DataFrame.add_suffix(suffix)  
Concatenate suffix string with panel items names.  

Parameters suffix : string  

Returns with_suffix : type of caller
pandas.DataFrame.agg

DataFrame.agg (func, axis=0, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables
New in version 0.20.0.

Parameters func : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:
• string function name
• function
• list of functions
• dict of column names -> functions (or list of functions)

Returns aggregated : DataFrame

See also:
groupby.aggregate, pandas.DataFrame.resample.aggregate, pandas.
DataFrame.rolling.aggregate

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.

Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                   index=pd.date_range('1/1/2000', periods=10))
>>> df.iloc[3:7] = np.nan
```

Aggregate these functions across all columns

```python
>>> df.agg(['sum', 'min'])
     A          B          C
sum -0.182253 -0.614014 -2.909534
min -1.916563 -1.460076 -1.568297
```

Different aggregations per column

```python
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
     A         B
max   NaN  1.514318
min -1.916563 -1.460076
sum -0.182253   NaN
```
pandas.DataFrame.aggregate

DataFrame.aggregate (func, axis=0, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables
New in version 0.20.0.

Parameters func : callable, string, dictionary, or list of string/callables
Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:
- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

Returns aggregated : DataFrame

See also:

Notes
Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.

Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                    index=pd.date_range('1/1/2000', periods=10))
>>> df.iloc[3:7] = np.nan
Aggregate these functions across all columns

```
```python
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
   A     B
max NaN  1.514318
min -1.916563 -1.460076
sum -0.182253 NaN
```

**pandas.DataFrame.align**

*DataFrame.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 for each axis

**Parameters**

- **other**: DataFrame or Series
  - *axis*: allowed axis of the other object, default None
  - *level*: int or level name, default None
  - *copy*: boolean, 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*: str, default None
  - *limit*: int, default 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**

*left, right*: (DataFrame, type of other)

Aligned objects

**pandas.DataFrame.all**

*DataFrame.all* (*axis=None, bool_only=None, skipna=None, level=None, **kwargs)*

Return whether all elements are True over requested axis

**Parameters**

- **axis**: {index (0), columns (1)}
- **skipna**: boolean, 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

bool_only : boolean, default None
Include only boolean columns. If None, will attempt to use everything, then use
only boolean data. Not implemented for Series.

Returns all : Series or DataFrame (if level specified)

pandas.DataFrame.any

DataFrame.any (axis=None, bool_only=None, skipna=None, level=None, **kwargs)
Return whether any element is True over requested axis

Parameters axis : {index (0), columns (1)}

skipna : boolean, 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

bool_only : boolean, default None
Include only boolean columns. If None, will attempt to use everything, then use
only boolean data. Not implemented for Series.

Returns any : Series or DataFrame (if level specified)

pandas.DataFrame.append

DataFrame.append (other, ignore_index=False, verify_integrity=False)
Append rows of other to the end of this frame, returning a new object. Columns not in this frame are
added as new columns.

Parameters other : DataFrame or Series/dict-like object, or list of these
The data to append.

ignore_index : boolean, default False
If True, do not use the index labels.

verify_integrity : boolean, default False
If True, raise ValueError on creating index with duplicates.

Returns appended : DataFrame

See also:

pandas.concat General function to concatenate DataFrame, Series or Panel 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'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

With `ignore_index` set to True:

```python
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

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
```
**pandas.DataFrame.apply**

**DataFrame.apply** *(func, axis=0, broadcast=False, raw=False, reduce=None, args=(), **kwds)*

Applies function along input axis of DataFrame.

Objects passed to functions are Series objects having index either the DataFrame’s index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates, or the reduce argument if the DataFrame is empty.

**Parameters**

- **func**: function
  - Function to apply to each column/row
- **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
  - 0 or ‘index’: apply function to each column
  - 1 or ‘columns’: apply function to each row
- **broadcast**: boolean, default False
  - For aggregation functions, return object of same size with values propagated
- **raw**: boolean, default False
  - If False, convert each row or column into a Series. If raw=True the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance
- **reduce**: boolean or None, default None
  - Try to apply reduction procedures. If the DataFrame is empty, apply will use reduce to determine whether the result should be a Series or a DataFrame. If reduce is None (the default), apply’s return value will be guessed by calling func on an empty Series (note: while guessing, exceptions raised by func will be ignored). If reduce is True a Series will always be returned, and if False a DataFrame will always be returned.
- **args**: tuple
  - Positional arguments to pass to function in addition to the array/series

**Returns**

- **applied**: Series or DataFrame

**See also:**

- *DataFrame.applymap* For elementwise operations
- *DataFrame.aggregate* only perform aggregating type operations
- *DataFrame.transform* only perform transforming type operations
Notes

In the current implementation apply calls func twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if func has side-effects, as they will take effect twice for the first column/row.

Examples

```python
>>> df.apply(numpy.sqrt)  # returns DataFrame
>>> df.apply(numpy.sum, axis=0)  # equiv to df.sum(0)
>>> df.apply(numpy.sum, axis=1)  # equiv to df.sum(1)
```

pandas.DataFrame.applymap

DataFrame.applymap(func)

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing map(func, series) for each series in the DataFrame

Parameters

func : function

Python function, returns a single value from a single value

Returns

applied : DataFrame

See also:

DataFrame.apply For operations on rows/columns

Examples

```python
>>> df = pd.DataFrame(np.random.randn(3, 3))
>>> df
   0   1   2
0 -0.029638 1.081563 1.280300
1 0.647747 0.831136 -1.549481
2 0.513416 -0.884417 0.195343
>>> df = df.applymap(lambda x: '%.2f' % x)
>>> df
   0   1   2
0 -0.03 1.08 1.28
1 0.65 0.83 -1.55
2 0.51 -0.88 0.20
```

pandas.DataFrame.as_blocks

DataFrame.as_blocks(copy=True)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)
Parameters  

**copy**: boolean, default True

**Returns**  

**values**: a dict of dtype -> Constructor Types

---

### pandas.DataFrame.as_matrix

**DataFrame.as_matrix**(columns=None)  
Convert the frame to its Numpy-array representation.

**Parameters**  

**columns**: list, optional, default:None  
If None, return all columns, otherwise, returns specified columns.

**Returns**  

**values**: ndarray  
If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

**pandas.DataFrame.values**

---

### Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

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 upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use `.values`.

---

### pandas.DataFrame.asfreq

**DataFrame.asfreq**(freq, method=None, how=None, normalize=False, fill_value=None)  
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. resample is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

**Parameters**  

**freq**: DateOffset object, or string  

**method**: {'backfill'/'bfill', 'pad'/'ffill'}, 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).

New in version 0.20.0.

Returns converted : type of caller

See also:

reindex

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
In [13]: df.asfreq(freq='30S', method='bfill')
Out[13]:
          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)`

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

**Parameters**

- `where` : date or array of dates
- `subset` : string or list of strings, default None
  
  if not None use these columns for NaN propagation

**Returns**

- `where` is scalar
  - value or NaN if input is Series
  - Series if input is DataFrame

where is Index: same shape object as input

**See also:**

`merge_asof`

**Notes**

Dates are assumed to be sorted Raises if this is not the case

**pandas.DataFrame.assign**

`DataFrame.assign(**kwargs)`

Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.

**Parameters**

- `kwargs` : keyword, value pairs

  keywords are the column names. 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**

- `df` : DataFrame
A new DataFrame with the new columns in addition to all the existing columns.

Notes

For python 3.6 and above, the columns are inserted in the order of **kwargs. For python 3.5 and earlier, since **kwargs is unordered, the columns are inserted in alphabetical order at the end of your DataFrame. Assigning multiple columns within the same assign is possible, but you cannot reference other columns created within the same assign call.

Examples

```python
>>> df = DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})

Where the value is a callable, evaluated on df:
```

```python
>>> df.assign(ln_A = lambda x: np.log(x.A))

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>ln_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.426905</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-0.780949</td>
<td>0.693147</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-0.418711</td>
<td>1.098612</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-0.269708</td>
<td>1.386294</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-0.274002</td>
<td>1.609438</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>-0.500792</td>
<td>1.791759</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1.649697</td>
<td>1.945910</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>-1.495604</td>
<td>2.079442</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>0.549296</td>
<td>2.197225</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>-0.758542</td>
<td>2.302585</td>
</tr>
</tbody>
</table>
```

Where the value already exists and is inserted:

```python
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>ln_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.426905</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-0.780949</td>
<td>0.693147</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-0.418711</td>
<td>1.098612</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-0.269708</td>
<td>1.386294</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-0.274002</td>
<td>1.609438</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>-0.500792</td>
<td>1.791759</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1.649697</td>
<td>1.945910</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>-1.495604</td>
<td>2.079442</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>0.549296</td>
<td>2.197225</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>-0.758542</td>
<td>2.302585</td>
</tr>
</tbody>
</table>
```

```
pandas.DataFrame.astype
```

DataFrame.astype(dtype, copy=True, errors='raise', **kwargs)

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

New in version 0.20.0.

**raise_on_error**: raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs**: keyword arguments to pass on to the constructor

**Returns**

- **casted**: type of caller

See also:

- `pandas.to_datetime` Convert argument to datetime.
- `pandas.to_timedelta` Convert argument to timedelta.
- `pandas.to_numeric` Convert argument to a numeric type.
- `numpy.ndarray.astype` Cast a numpy array to a specified type.

### Examples

```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
>>> ser.astype('category', ordered=True, categories=[2, 1])
0  1
1  2
```
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('int', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0  10
1  2
dtype: int64
```

### pandas.DataFrame.at_time

**DataFrame.at_time** (time, asof=False)

Select values at particular time of day (e.g. 9:30AM).

- **Parameters**
  - `time`: datetime.time or string
- **Returns**
  - `values_at_time`: type of caller

### pandas.DataFrame.between_time

**DataFrame.between_time** (start_time, end_time, include_start=True, include_end=True)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

- **Parameters**
  - `start_time`: datetime.time or string
  - `end_time`: datetime.time or string
  - `include_start`: boolean, default True
  - `include_end`: boolean, default True
- **Returns**
  - `values_between_time`: type of caller

### pandas.DataFrame.bfill

**DataFrame.bfill** (axis=None, inplace=False, limit=None, downcast=None)

Synonym for **DataFrame.fillna(method='bfill')**

### pandas.DataFrame.bool

**DataFrame.bool()**

Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean.
pandas.DataFrame.boxplot

DataFrame.boxplot (column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, figsize=None, layout=None, return_type=None, **kwds)

Make a box plot from DataFrame column optionally grouped by some columns or other inputs

Parameters data: the pandas object holding the data
  column: column name or list of names, or vector
    Can be any valid input to groupby
  by: string or sequence
    Column in the DataFrame to group by
  ax: Matplotlib axes object, optional
  fontsize: int or string
  rot: label rotation angle
  figsize: A tuple (width, height) in inches
  grid: Setting this to True will show the grid
  layout: tuple (optional)
    (rows, columns) for the layout of the plot
  return_type: {None, ‘axes’, ‘dict’, ‘both’}, default None
    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, unless return_type is None, in which case a NumPy array of axes is returned with the same shape as layout. See the prose documentation for more.
  kwds: other plotting keyword arguments to be passed to matplotlib boxplot

Returns lines: dict
  ax: matplotlib Axes
  (ax, lines): namedtuple

Notes

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.

pandas.DataFrame.clip

DataFrame.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)

Trim values at input threshold(s).
**Parameters**

- **lower**: float or array_like, default None
- **upper**: float or array_like, default None
- **axis**: int or string axis name, optional

Align object with lower and upper along the given axis.

- **inplace**: boolean, default False

Whether to perform the operation in place on the data. New in version 0.21.0.

**Returns**

- **clipped**: Series

**Examples**

```python
>>> df
   0      1
0 0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967

>>> df.clip(-1.0, 0.5)
   0      1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000

>>> t
   0   1
0  0.3  1.2
1  1.2  2.1
2  2.1  3.0
3  3.0  4.1
dtype: float64

>>> df.clip(t, t + 1, axis=0)
   0      1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```

**pandas.DataFrame.clip_lower**

DataFrame.clip_lower(threshold, axis=None, inplace=False)

Return copy of the input with values below given value(s) truncated.

**Parameters**

- **threshold**: float or array_like
- **axis**: int or string axis name, optional
Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns clipped** : same type as input

See also:

clip

**pandas.DataFrame.clip_upper**

DataFrame.clip_upper(threshold, axis=None, inplace=False)

Return copy of input with values above given value(s) truncated.

**Parameters threshold** : float or array_like

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns clipped** : same type as input

See also:

clip

**pandas.DataFrame.combine**

DataFrame.combine(other, func, fill_value=None, overwrite=True)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame’s value (which might be NaN as well)

**Parameters other** : DataFrame

**func** : function

**fill_value** : scalar value

**overwrite** : boolean, default True

If True then overwrite values for common keys in the calling frame

**Returns result** : DataFrame

**pandas.DataFrame.combine_first**

DataFrame.combine_first(other)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

**Parameters other** : DataFrame

**Returns combined** : DataFrame
Examples

a’s values prioritized, use values from b to fill holes:

```python
>>> a.combine_first(b)
```

**pandas.DataFrame.compound**

DataFrame.**compound**(axis=None, skipna=None, level=None)

Return the compound percentage of the values for the requested axis

**Parameters**

- **axis**: {index (0), columns (1)}
- **skipna**: boolean, default True
  
  Exclude NA/null values. If an entire row/column is NA or empty, 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
- **numeric_only**: boolean, 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**

- **compounded**: Series or DataFrame (if level specified)

**pandas.DataFrame.consolidate**

DataFrame.**consolidate**(inplace=False)

DEPRECATED: consolidate will be an internal implementation only.

**pandas.DataFrame.convert_objects**

DataFrame.**convert_objects**(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)

Deprecated. Attempt to infer better dtype for object columns

**Parameters**

- **convert_dates**: boolean, default True
  
  If True, convert to date where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.
- **convert_numeric**: boolean, default False
  
  If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.
- **convert_timedeltas**: boolean, default True
  
  If True, convert to timedelta where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.
- **copy**: boolean, default True
If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

**Returns** converted : same as input object

**See also:**

- `pandas.to_datetime` Convert argument to datetime.
- `pandas.to_timedelta` Convert argument to timedelta.
- `pandas.to_numeric` Return a fixed frequency timedelta index, with day as the default.

### pandas.DataFrame.copy

**DataFrame.copy**(deep=True)

Make a copy of this objects data.

**Parameters** deep : boolean or string, default True

Make a deep copy, including a copy of the data and the indices. With deep=False neither the indices or the data are copied.

Note that when deep=True data is copied, 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.

**Returns** copy : type of caller

### pandas.DataFrame.corr

**DataFrame.corr**(method='pearson', min_periods=1)

Compute pairwise correlation of columns, excluding NA/null values

**Parameters** method : {'pearson', 'kendall', 'spearman'}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

**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** y : DataFrame

### pandas.DataFrame.corrwith

**DataFrame.corrwith**(other, axis=0, drop=False)

Compute pairwise correlation between rows or columns of two DataFrame objects.

**Parameters** other : DataFrame

**axis** : {0 or ‘index’, 1 or ‘columns’}, default 0

0 or ‘index’ to compute column-wise, 1 or ‘columns’ for row-wise
**drop** : boolean, default False

Drop missing indices from result, default returns union of all

**Returns** **correls** : Series

---

**pandas.DataFrame.count**

DataFrame.**count**(axis=0, level=None, numeric_only=False)

Return Series with number of non-NA/null observations over requested axis. Works with non-floating point data as well (detects NaN and None)

**Parameters** **axis** : 0 or ‘index’, 1 or ‘columns’, default 0

0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric_only** : boolean, default False

Include only float, int, boolean data

**Returns** **count** : Series (or DataFrame if level specified)

---

**pandas.DataFrame.cov**

DataFrame.**cov**(min_periods=None)

Compute pairwise covariance of columns, excluding NA/null values

**Parameters** **min_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result.

**Returns** **y** : DataFrame

**Notes**

y contains the covariance matrix of the DataFrame’s time series. The covariance is normalized by N-1 (unbiased estimator).

---

**pandas.DataFrame.cummax**

DataFrame.**cummax**(axis=None, skipna=True, *args, **kwargs)

Return cumulative max over requested axis.

**Parameters** **axis** : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**Returns** **cummax** : Series

**See also:**
**pandas.core.window.Expanding.max**  Similar functionality but ignores NaN values.

**pandas.DataFrame.cummin**

DataFrame.cummin(\(axis=None, \)skipna=True, *args, **kwargs)

Return cumulative minimum over requested axis.

- **Parameters**
  - axis: {index (0), columns (1)}
  - skipna: boolean, default True

    Exclude NA/null values. If an entire row/column is NA, the result will be NA

- **Returns**
  - cummin: Series

See also:

**pandas.core.window.Expanding.min**  Similar functionality but ignores NaN values.

**pandas.DataFrame.cumprod**

DataFrame.cumprod(\(axis=None, \)skipna=True, *args, **kwargs)

Return cumulative product over requested axis.

- **Parameters**
  - axis: {index (0), columns (1)}
  - skipna: boolean, default True

    Exclude NA/null values. If an entire row/column is NA, the result will be NA

- **Returns**
  - cumprod: Series

See also:

**pandas.core.window.Expanding.prod**  Similar functionality but ignores NaN values.

**pandas.DataFrame.cumsum**

DataFrame.cumsum(\(axis=None, \)skipna=True, *args, **kwargs)

Return cumulative sum over requested axis.

- **Parameters**
  - axis: {index (0), columns (1)}
  - skipna: boolean, default True

    Exclude NA/null values. If an entire row/column is NA, the result will be NA

- **Returns**
  - cumsum: Series

See also:

**pandas.core.window.Expanding.sum**  Similar functionality but ignores NaN values.
pandas: powerful Python data analysis toolkit, Release 0.21.0

pandas.DataFrame.describe

DataFrame.describe(percentiles=None, include=None, exclude=None)

Generates descriptive statistics 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(include=['O'])). To exclude pandas categorical columns, use 'category'
• None (default) : The result will exclude nothing.

Returns

summary: Series/DataFrame of summary statistics

See also:

DataFrame.count, DataFrame.max, DataFrame.min, DataFrame.mean, DataFrame.std, DataFrame.select_dtypes

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. Times-tamps 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
```

Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count    4
unique   3
top      a
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()
count    3
unique   2
top    2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({'object': ['a', 'b', 'c'],
                    'numeric': [1, 2, 3],
                    'categorical': pd.Categorical(['d', 'e', 'f'])})
>>> df.describe()
numeric
```

34.4. DataFrame 1497
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      c
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  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 numeric columns in a DataFrame description.

```python
>>> df.describe(include=[np.number])
numeric
   count  mean  std  min  25%  50%  75%  max
   3.0   2.0  1.0  1.0  1.5  2.0  2.5  3.0
```

Including only string columns in a DataFrame description.

```python
>>> df.describe(include=[np.object])
object
   count  unique
   3      3
```
Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
categorical
   count   top  freq
      3     f    1
```

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])
categorical  object
   count   top  freq
      3     f   1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.object])
categorical    numeric
   count   top  mean     std
      3     f  NaN     NaN
```

### pandas.DataFrame.diff

DataFrame\[**diff**(periods=1, axis=0)

1st discrete difference of object

**Parameters**

- **periods**: int, default 1
  
  Periods to shift for forming difference

- **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
  
  Take difference over rows (0) or columns (1).

**Returns**

- **diffed**: DataFrame

### pandas.DataFrame.div

DataFrame\[**div**(other, axis='columns', level=None, fill_value=None)

Floating division of dataframe and other, element-wise (binary operator \textit{truediv}).
Equivalent to \texttt{dataframe / other}, but with support to substitute a \texttt{fill\_value} for missing data in one of the inputs.

**Parameters** \texttt{other} : Series, DataFrame, or constant

- \texttt{axis} : \{0, 1, \texttt{‘index’}, \texttt{‘columns’}\}
  
  For Series input, axis to match Series index on

- \texttt{fill\_value} : None or float value, default None
  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

- \texttt{level} : int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns** \texttt{result} : DataFrame

**See also:**

\texttt{DataFrame.rtruediv}

**Notes**

Mismatched indices will be unioned together

\texttt{pandas.DataFrame.divide}

\texttt{DataFrame.divide(\texttt{other}, \texttt{axis=\textquoteleftcolumns\textquoteright}, \texttt{level=None, fill_value=None})}

Floating division of dataframe and other, element-wise (binary operator \texttt{truediv}).

Equivalent to \texttt{dataframe / other}, but with support to substitute a \texttt{fill\_value} for missing data in one of the inputs.

**Parameters** \texttt{other} : Series, DataFrame, or constant

- \texttt{axis} : \{0, 1, \texttt{‘index’}, \texttt{‘columns’}\}
  
  For Series input, axis to match Series index on

- \texttt{fill\_value} : None or float value, default None
  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

- \texttt{level} : int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns** \texttt{result} : DataFrame

**See also:**

\texttt{DataFrame.rtruediv}

**Notes**

Mismatched indices will be unioned together
pandas.DataFrame.dot

DataFrame.dot(other)
Matrix multiplication with DataFrame or Series objects

Parameters
other : DataFrame or Series

Returns
dot_product : DataFrame or Series

pandas.DataFrame.drop

DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')
Return new object with labels in requested axis removed.

Parameters
labels : single label or list-like
Index or column labels to drop.

axis : int or axis name
Whether to drop labels from the index (0 / ‘index’) or columns (1 / ‘columns’).

index, columns : single label or list-like
Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels).

New in version 0.21.0.

level : int or level name, default None
For MultiIndex

inplace : bool, default False
If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'
If ‘ignore’, suppress error and existing labels are dropped.

Returns
dropped : type of caller

Notes
Specifying both labels and index or columns will raise a ValueError.

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

>>> df.drop(columns=['B', 'C'])
   A  D
0  0  3
1  4  7
2  8 11

Drop a row by index

>>> df.drop([0, 1])
   A  B  C  D  
2  8  9 10  11
```

**pandas.DataFrame.drop_duplicates**

DataFrame. `drop_duplicates` (subset=None, keep='first', inplace=False)
Return DataFrame with duplicate rows removed, optionally only considering certain columns

**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'
  - first : Drop duplicates except for the first occurrence.
  - last : Drop duplicates except for the last occurrence.
  - False : Drop all duplicates.
- **inplace** : boolean, default False
  Whether to drop duplicates in place or to return a copy

**Returns**
- deduplicated : DataFrame

**pandas.DataFrame.dropna**

DataFrame. `dropna` (axis=0, how='any', thresh=None, subset=None, inplace=False)
Return object with labels on given axis omitted where alternately any or all of the data are missing

**Parameters**
- **axis** : {0 or ‘index’, 1 or ‘columns’}, or tuple/list thereof
  Pass tuple or list to drop on multiple axes
- **how** : {'any', 'all'}
  - any : if any NA values are present, drop that label
  - all : if all values are NA, drop that label
- **thresh** : int, default None
  int value : require that many non-NA values
subset : array-like

Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include

inplace : boolean, default False

If True, do operation in-place and return None.

Returns dropped : DataFrame

Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                      [3, 4, np.nan, 1],
                      [np.nan, np.nan, np.nan, 5]],
                      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  5.0

Drop the columns where all elements are nan:

```python
>>> df.dropna(axis=1, how='all')
   A    B    D
0 NaN  2.0  0.0
1  3.0  4.0  1.0
2  NaN  NaN  5.0
```

Drop the columns where any of the elements is nan:

```python
>>> df.dropna(axis=1, how='any')
   D
0  0
1  1
2  5
```

Drop the rows where all of the elements are nan (there is no row to drop, so df stays the same):

```python
>>> df.dropna(axis=0, how='all')
   A    B    C    D
0 NaN  2.0  NaN  0.0
1  3.0  4.0  NaN  1.0
2  NaN  NaN  NaN  5.0
```

Keep only the rows with at least 2 non-na values:

```python
>>> df.dropna(thresh=2)
   A    B    C    D
0 NaN  2.0  NaN  0.0
1  3.0  4.0  NaN  1.0
```

pandas.DataFrame.duplicated

DataFrame.duplicated(subset=None, keep='first')

Return boolean Series denoting duplicate rows, optionally only considering certain columns
**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’
  - **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**

- **duplicated**: Series

---

**pandas.DataFrame.eq**

`DataFrame.eq(\texttt{other, axis=}'columns', level=None)`

Wrapper for flexible comparison methods eq

**pandas.DataFrame.equals**

`DataFrame.equals(\texttt{other})`

Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

**pandas.DataFrame.eval**

`DataFrame.eval(\texttt{expr, inplace=False, **kwargs})`

Evaluate an expression in the context of the calling DataFrame instance.

**Parameters**

- **expr**: string
  - The expression string to evaluate.

- **inplace**: bool, default False
  - If the expression contains an assignment, whether to perform the operation in-place and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

**kwargs**: dict
  - See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

**Returns**

- **ret**: ndarray, scalar, or pandas object

**See also:**

- `pandas.DataFrame.query`, `pandas.DataFrame.assign`, `pandas.eval`

**Notes**

For more details see the API documentation for `eval()`. For detailed examples see *enhancing performance with eval*.
Examples

```python
>>> from numpy.random import randn
>>> from pandas import DataFrame

>>> df = DataFrame(randn(10, 2), columns=list('ab'))

>>> df.eval('a + b')

>>> df.eval('c = a + b')
```

`pandas.DataFrame.ewm`

DataFrame.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, freq=None, adjust=True, ignore_na=False, axis=0)

Provides exponential weighted functions

New in version 0.18.0.

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, optional
  Specify decay in terms of half-life, \( \alpha = 1 - exp(log(0.5)/\text{halflife}) \), for \( \text{halflife} > 0 \)

- **alpha** : float, optional
  Specify smoothing factor \( \alpha \) directly, \( 0 < \alpha \leq 1 \)
  New in version 0.18.0.

- **min_periods** : int, default 0
  Minimum number of observations in window required to have a value (otherwise result is NA).

- **freq** : None or string alias / date offset object, default=None
  Deprecated since version 0.18.0: Frequency to conform to before computing statistic

- **adjust** : boolean, default True
  Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

- **ignore_na** : boolean, default False
  Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns

A Window sub-classed for the particular operation
Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

The freq keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of resample() (i.e. using the mean).

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), ..., 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

Examples

```python
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
   B
0  0
1  1
2  2
3  NaN
4  4

>>> df.ewm(com=0.5).mean()
   B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

pandas.DataFrame.expanding

DataFrame.expanding (min_periods=1, freq=None, center=False, axis=0)

Provides expanding transformations.

New in version 0.18.0.

Parameters min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)
Deprecated since version 0.18.0: Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**center** : boolean, default False

Set the labels at the center of the window.

**axis** : int or string, default 0

**Returns**  a Window sub-classed for the particular operation

**Notes**

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The freq keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the `mean`).

**Examples**

```python
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

>>> df.expanding(2).sum()
   B
0 NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

**pandas.DataFrame.ffill**

`Dataframe.ffill` *(axis=None, inplace=False, limit=None, downcast=None)*

Synonym for `Dataframe.fillna(method='ffill')`

**pandas.DataFrame.fillna**

`Dataframe.fillna` *(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)*

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 / fill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** : {0 or ‘index’, 1 or ‘columns’}

**inplace** : boolean, 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 filled** : DataFrame

**See also:**

* reindex,
  * asfreq

**Examples**

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                      ...                     [3, 4, np.nan, 1],
                      ...                     [np.nan, np.nan, np.nan, 5],
                      ...                     [np.nan, 3, np.nan, 4]],
                      ...                     columns=list('ABCD'))
```  
Replace all NaN elements with 0s.

```python
>>> df.fillna(0)
```

We can also propagate non-null values forward or backward.

```python
>>> df.fillna(method='ffill')
```
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
2  0.0  1.0  2.0  5.0
3  0.0  3.0  2.0  4.0
```

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  Na  1.0
2  Na  1.0  Na  5.0
3  Na  3.0  Na  4.0
```

**pandas.DataFrame.filter**

`DataFrame.filter(items=None, like=None, regex=None, axis=None)`

Subset rows or columns of dataframe according to labels in the specified index.

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
  List of info axis to restrict to (must not all be present)

- `like` : string
  Keep info axis where “arg in col == True”

- `regex` : string (regular expression)
  Keep info axis with re.search(regex, col) == True

- `axis` : int or string axis name
  The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame

**Returns**

same type as input object

**See also:**

`pandas.DataFrame.loc`

**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 `[]`.  

---

34.4. DataFrame
Examples

```python
>>> 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*)
Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters
- *offset* : string, DateOffset, dateutil.relativedelta

Returns
- *subset* : type of caller

Examples

ts.first(‘10D’) -> First 10 days

`pandas.DataFrame.first_valid_index`

Dataframe `first_valid_index`()
Return index for first non-NA/null value.

Returns
- *scalar* : type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty DataFrame.
**pandas.DataFrame.floordiv**

`DataFrame.floordiv(other, axis='columns', level=None, fill_value=None)`

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.

**Parameters**

- **other**: Series, DataFrame, or constant
  
  - **axis**: `{0, 1, 'index', 'columns'}`
    
    For Series input, axis to match Series index on
  
  - **fill_value**: None or float value, default None
    
    Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
  
  - **level**: int or name
    
    Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- **result**: DataFrame

**See also:**

- `DataFrame.rfloordiv`

**Notes**

Mismatched indices will be unioned together

**pandas.DataFrame.from_csv**

classmethod `DataFrame.from_csv(path, header=0, sep=\'\', index_col=0, parse_dates=True, encoding=None, tupleize_cols=None, infer_datetime_format=False)`

Read CSV file (DEPRECATED, please use `pandas.read_csv()` instead).

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- `index_col` is 0 instead of None (take first column as index by default)
- `parse_dates` is True instead of False (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

**Parameters**

- **path**: string file path or file handle / StringIO
  
  - **header**: int, default 0
    
    Row to use as header (skip prior rows)
  
  - **sep**: string, default ‘\’
    
    Field delimiter
index_col : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates : boolean, default True

Parse dates. Different default from read_table

tupleize_cols : boolean, default False

write multi_index columns as a list of tuples (if True) or new (expanded format) if False

infer_datetime_format: boolean, default False

If True and parse_dates is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns y : DataFrame

See also:

pandas.read_csv

pandas.DataFrame.from_dict

classmethod DataFrame.from_dict(data, orient='columns', dtype=None)

Construct DataFrame from dict of array-like or dicts

Parameters data : dict

{field : array-like} or {field : dict}

orient : {'columns', 'index'}, 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’.

dtype : dtype, default None

Data type to force, otherwise infer

Returns DataFrame

pandas.DataFrame.from_items

classmethod DataFrame.from_items(items, columns=None, orient='columns')

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

Parameters items : sequence of (key, value) pairs

Values should be arrays or Series.

columns : sequence of column labels, optional

Must be passed if orient='index'.

orient : {'columns', 'index'}, default 'columns'
The “orientation” of the data. If the keys of the input correspond to column labels, pass ‘columns’ (default). Otherwise if the keys correspond to the index, pass ‘index’.

Returns frame : DataFrame

**pandas.DataFrame.from_records**

classmethod DataFrame.from_records(data, index=None, exclude=None, columns=None, coerce=True, coerce_float=False, nrows=None)

Convert structured or record ndarray to DataFrame

Parameters data : ndarray (structured dtype), list of tuples, dict, or DataFrame

index : string, 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 : boolean, default False

Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

Returns df : DataFrame

**pandas.DataFrame.ge**

DataFrame.ge(other, axis='columns', level=None)

Wrapper for flexible comparison methods ge

**pandas.DataFrame.get**

DataFrame.get(key, default=None)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters key : object

Returns value : type of items contained in object

**pandas.DataFrame.get_dtype_counts**

DataFrame.get_dtype_counts()

Return the counts of dtypes in this object.
pandas: powerful Python data analysis toolkit, Release 0.21.0

**pandas.DataFrame.get_ftype_counts**

DataFrame.get_ftype_counts()
Return the counts of ftypes in this object.

**pandas.DataFrame.get_value**

DataFrame.get_value(index, col, takeable=False)
Quickly retrieve single value at passed column and index
Deprecated since version 0.21.0.
Please use .at[] or .iat[] accessors.

Parameters:
- **index**: row label
- **col**: column label
- **takeable**: interpret the index/col as indexers, default False

Returns:
- **value**: scalar value

**pandas.DataFrame.get_values**

DataFrame.get_values()
same as values (but handles sparseness conversions)

**pandas.DataFrame.groupby**

DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, **kwargs)
Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

Parameters:
- **by**: mapping, function, str, or iterable
  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 an ndarray is passed, the values are used as-is determine the groups. A str or list of strs may be passed to group by the columns in self
- **axis**: int, default 0
- **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**: boolean, 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**: boolean, 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** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**Returns** GroupBy object

**Examples**

**DataFrame results**

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

**DataFrame with hierarchical index**

```
>>> data.groupby(['col1', 'col2']).mean()
```

**pandas.DataFrame.gt**

DataFrame.gt (other, axis='columns', level=None)

Wrapper for flexible comparison methods gt

**pandas.DataFrame.head**

DataFrame.head (n=5)

Return the first n rows.

**Parameters**

n : int, default 5

Number of rows to select.

**Returns**

obj_head : type of caller

The first n rows of the caller object.

**pandas.DataFrame.hist**

DataFrame.hist (data, 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, **kwds)

Draw histogram of the DataFrame’s series using matplotlib / pylab.

**Parameters**

data : DataFrame

column : string or sequence

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 : boolean, 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
ax : matplotlib axes object, default None
sharex : boolean, 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 subplots in a figure!
sharey : boolean, default False
    In case subplots=True, share y axis and set some y axis labels to invisible
figsize : tuple
    The size of the figure to create in inches by default
layout : tuple, optional
    Tuple of (rows, columns) for the layout of the histograms
bins : integer, default 10
    Number of histogram bins to be used
kwds : other plotting keyword arguments
    To be passed to hist function

pandas.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
        0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise
    skipna : boolean, default True
        Exclude NA/null values. If an entire row/column is NA, the result will be first index.
**Returns** `idxmax` : Series

**See also:**

`Series.idxmax`

**Notes**

This method is the DataFrame version of `ndarray.argmax`.

**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  
0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise

`skipna` : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**Returns** `idxmin` : Series

**See also:**

`Series.idxmin`

**Notes**

This method is the DataFrame version of `ndarray.argmin`.

**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.

New in version 0.21.0.

**Returns** `converted` : same type as input object

**See also:**

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to numeric type
Examples

```python
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
0  1
1  2
2  3
```

```python
>>> df.dtypes
A  object
dtype: object
```

```python
>>> df.infer_objects().dtypes
A  int64
dtype: object
```

**pandas.DataFrame.info**

DataFrame.info(verbos=None, buf=None, max_cols=None, memory_usage=None, null_counts=None)

Concise summary of a DataFrame.

**Parameters**

- **verbose** : {None, True, False}, optional
  Whether to print the full summary. None follows the `display.max_info_columns` setting. True or False overrides the `display.max_info_columns` setting.

- **buf** : writable buffer, defaults to `sys.stdout`

- **max_cols** : int, default None
  Determines whether full summary or short summary is printed. None follows the `display.max_info_columns` setting.

- **memory_usage** : boolean/string, default None
  Specifies whether total memory usage of the DataFrame elements (including index) should be displayed. None follows the `display.memory_usage` setting. True or False overrides the `display.memory_usage` setting. A value of `deep` is equivalent of True, with deep introspection. Memory usage is shown in human-readable units (base-2 representation).

- **null_counts** : boolean, default None
  Whether to show the non-null counts
  - If None, then only show if the frame is smaller than `max_info_rows` and `max_info_columns`.
  - If True, always show counts.
  - If False, never show counts.
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 : string, number, or hashable object
     label of the inserted column

 value : int, Series, or array-like

 allow_duplicates : bool, optional

pandas.DataFrame.interpolate

DataFrame.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', downcast=None, **kwargs)

Interpolate values according to different methods.

 Please note that only method='linear' is supported for DataFrames/Series with a MultiIndex.

 **Parameters**

 method : {'linear', 'time', 'index', 'values', 'nearest', 'zero',
 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline',
 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

 • 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
 • 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
 • 'index', 'values': use the actual numerical values of the index
 • 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to scipy.interpolate.interp1d. Both 'polynomial' and 'spline' require that you also specify an order (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
 • 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
 • 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

 New in version 0.18.1: Added support for the ‘akima’ method Added interpolate method ‘from_derivatives’ which replaces ‘piecewise_polynomial’ interpolation method in scipy 0.18; backwards-compatible with scipy < 0.18

 axis : {0, 1}, default 0
- 0: fill column-by-column
- 1: fill row-by-row

**limit**: int, default None.

Maximum number of consecutive NaNs to fill. Must be greater than 0.

**limit_direction**: {'forward', 'backward', 'both'}, default ‘forward’

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.17.0.

**inplace**: bool, default False

Update the DataFrame in place if possible.

**downcast**: optional, ‘infer’ or None, defaults to None

Downcast dtypes if possible.

**kwargs**: keyword arguments to pass on to the interpolating function.

**Returns** Series or DataFrame of same shape interpolated at the NaNs

**See also:** reindex, replace, fillna

**Examples**

Filling in NaNs

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0 0
1 1
2 2
3 3
dtype: float64
```

**pandas.DataFrame.isin**

**DataFrame.isin(values)**

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

**Parameters** values: iterable, Series, DataFrame or dictionary

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 dictionary, 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 of booleans

**Examples**

When values is a list:
```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
   A  B
0  True  True
1  False  False
2  True  False

When values is a dict:
```
```
>>> df = DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
   A  B
0  True  False  # Note that B didn't match the 1 here.
1  False  True
2  True  True

When values is a Series or DataFrame:
```
```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
   A  B
0  True  False  # Column A in 'other' has a 3, but not at index 1.
1  False  False
2  True  True
```

**pandas.DataFrame.isna**

DataFrame.isna()

Return a boolean same-sized object indicating if the values are NA.

See also:

- DataFrame.notna  boolean inverse of isna
- DataFrame.isnull  alias of isna
- isna  top-level isna

**pandas.DataFrame.isnull**

DataFrame.isnull()

Return a boolean same-sized object indicating if the values are NA.

See also:

- DataFrame.notna  boolean inverse of isna
- DataFrame.isnull  alias of isna
- isna  top-level isna
pandas: powerful Python data analysis toolkit, Release 0.21.0

pandas.DataFrame.items

DataFrame.items()  
Iterator over (column name, Series) pairs.

See also:

iterrows  Iterate over DataFrame rows as (index, Series) pairs.

itertuples  Iterate over DataFrame rows as namedtuples of the values.

pandas.DataFrame.iteritems

DataFrame.iteritems()  
Iterator over (column name, Series) pairs.

See also:

iterrows  Iterate over DataFrame rows as (index, Series) pairs.

itertuples  Iterate over DataFrame rows as namedtuples of the values.

pandas.DataFrame.iterrows

DataFrame.iterrows()  
Iterate over DataFrame rows as (index, Series) pairs.

Returns  it : generator

A generator that iterates over the rows of the frame.

See also:

itertuples  Iterate over DataFrame rows as namedtuples of the values.

iteritems  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, with index value as first element of the tuple.

**Parameters**
- **index**: boolean, default True
  - If True, return the index as the first element of the tuple.
- **name**: string, default “Pandas”
  - The name of the returned namedtuples or None to return regular tuples.

**See also:**
- *iterrows* Iterate over DataFrame rows as (index, Series) pairs.
- *iteritems* 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. With a large number of columns (>255), regular tuples are returned.

**Examples**

```python
>>> df = pd.DataFrame({"col1": [1, 2], 'col2': [0.1, 0.2]},
                    index=['a', 'b'])
>>> df
  col1 col2
a  1  0.1
b  2  0.2
>>> for row in df.itertuples():
...    print(row)
...    print(row)
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

**pandas.DataFrame.join**

DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)

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 with name field set, 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**: column name, tuple/list of column names, or array-like
Column(s) in the caller to join on the index in other, otherwise joins index-on-index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if 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 frame’s index
- outer: form union of calling frame’s index (or column if on is specified) with other frame’s index, and sort it lexicographically
- inner: form intersection of calling frame’s index (or column if on is specified) with other frame’s index, preserving the order of the calling’s one

**lsuffix**: string

Suffix to use from left frame’s overlapping columns

**rsuffix**: string

Suffix to use from right frame’s overlapping columns

**sort**: boolean, 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 joined**: DataFrame

See also:

**DataFrame.merge** For column(s)-on-column(s) operations

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Examples

```python
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})

>>> caller
   A  key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5

>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...    'B': ['B0', 'B1', 'B2']})
```
Join DataFrames using their indexes.

```python
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The
joined DataFrame will have key as its index.

```python
>>> caller.set_index('key').join(other.set_index('key'))
```

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 the caller. This method preserves the original caller’s index in
the result.

```python
>>> caller.join(other.set_index('key'), on='key')
```

### pandas.DataFrame.keys

DataFrame.keys()

Get the ‘info axis’ (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.
DataFrame.kurt

DataFrame.kurt (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters
axis : {index (0), columns (1)}
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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
numeric_only : boolean, 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
kurt : Series or DataFrame (if level specified)

DataFrame.kurtosis

DataFrame.kurtosis (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters
axis : {index (0), columns (1)}
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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
numeric_only : boolean, 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
kurt : Series or DataFrame (if level specified)

DataFrame.last

DataFrame.last (offset)
Convenience method for subsetting final periods of time series data based on a date offset.

Parameters
offset : string, DateOffset, dateutil.relativedelta

Returns
subset : type of caller
Examples

```python
ts.last('5M') -> Last 5 months
```

**pandas.DataFrame.last_valid_index**

```python
DataFrame.last_valid_index()
```
Return index for first non-NA/null value.

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty DataFrame.

**pandas.DataFrame.le**

```python
DataFrame.le(other, axis='columns', level=None)
```
Wrapper for flexible comparison methods le

**pandas.DataFrame.lookup**

```python
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.

**Parameters**
- `row_labels`: sequence
  The row labels to use for lookup
- `col_labels`: sequence
  The column labels to use for lookup

**Notes**

Akin to:

```python
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

**Examples**

```python
values [ndarray] The found values
```
**pandas.DataFrame.lt**

DataFrame\.lt (other, axis='columns', level=None)

Wrapper for flexible comparison methods lt

**pandas.DataFrame.mad**

DataFrame\.mad (axis=None, skipna=None, level=None)

Return the mean absolute deviation of the values for the requested axis

**Parameters**

axis : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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**

mad : Series or DataFrame (if level specified)

**pandas.DataFrame.mask**

DataFrame\.mask (cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

**Parameters**

cond : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as cond.

other : scalar, NDFrame, or callable

Entries where cond is True are replaced with corresponding value from other. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data
axis : alignment axis if needed, default None
level : alignment level if needed, default None
errors : str, {‘raise’, ‘ignore’}, default ‘raise’
  • raise: allow exceptions to be raised
  • ignore: suppress exceptions. On error return original object
  Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

try_cast : boolean, default False
  try to cast the result back to the input type (if possible),
raise_on_error : boolean, default True
  Whether to raise on invalid data types (e.g. trying to where on strings)
  Deprecated since version 0.21.0.

Returns wh : same type as caller

See also:
DataFrame.where()

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   1   2   3   4
0   NaN  1.0  2.0  3.0  4.0
```

```python
>>> s.mask(s > 0)
   0   1   2   3   4
0   0.0 NaNNaNNaN
1   NaN NaN NaN NaN
2   NaN NaN NaN NaN
3   NaN NaN NaN NaN
4   NaN NaN NaN NaN
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

```python
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

```python
>>> 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
```

```python
>>> 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=None, skipna=None, level=None, numeric_only=None, **kwargs)`

This method returns the maximum of the values in the object. 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)}
- `skipna` : boolean, default True
  Exclude NA/null values. If an entire row/column is NA or empty, 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
- `numeric_only` : boolean, 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**

- `max` : Series or DataFrame (if level specified)
pandas.DataFrame.mean

DataFrame.mean (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the mean of the values for the requested axis

Parameters
axis: {index (0), columns (1)}
skipna: boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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
numeric_only: boolean, 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
mean: Series or DataFrame (if level specified)

pandas.DataFrame.median

DataFrame.median (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the median of the values for the requested axis

Parameters
axis: {index (0), columns (1)}
skipna: boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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
numeric_only: boolean, 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
median: 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)
“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables 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’.

New in version 0.20.0.
Parameters frame : DataFrame

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 string, optional
If columns are a MultiIndex then use this level to melt.

See also:
melt, pivot_table, DataFrame.pivot

Examples

```python
>>> import pandas as pd
>>> 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
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')
   A myVarname  myValname
0  a          B   1
```

```
If you have multi-index columns:

```python
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
    A  B  C
0  D  E  F
1  a  b  c
2  1  2  3
3  4  5  6
```

```python
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A_variable value
0    a        B  1
1    b        B  3
2    c        B  5
```

```python
>>> df.melt(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.DataFrame.memory_usage

DataFrame._memory_usage_(index=True, deep=False)

Memory usage of DataFrame columns.

**Parameters**

- **index**: bool
  
  Specifies whether to include memory usage of DataFrame’s index in returned Series. If `index=True` (default is False) the first index of the Series is `Index`.

- **deep**: bool

  Introspect the data deeply, interrogate object dtypes for system-level memory consumption

**Returns**

- **sizes**: Series

  A series with column names as index and memory usage of columns with units of bytes.

**See also**

- `numpy.ndarray.nbytes`

**Notes**

Memory usage does not include memory consumed by elements that are not components of the array if `deep=False`
pandas.DataFrame.merge

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 objects by performing a database-style join operation by 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.

Parameters right: DataFrame

how : {'left', 'right', 'outer', 'inner'}, default 'inner'
  - 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

on : label or list
  Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

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_index : boolean, 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 : boolean, default False
  Use the index from the right DataFrame as the join key. Same caveats as left_index

sort : boolean, 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 : 2-length sequence (tuple, list, ...)
  Suffix to apply to overlapping column names in the left and right side, respectively

copy : boolean, default True
  If False, do not copy data unnecessarily

indicator : boolean or string, default False
If True, adds a column to output DataFrame called “_merge” with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of “left_only” for observations whose merge key only appears in ‘left’ DataFrame, “right_only” for observations whose merge key only appears in ‘right’ DataFrame, and “both” if the observation’s merge key is found in both.

New in version 0.17.0.

validate : string, default None

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.

New in version 0.21.0.

Returns merged : DataFrame

The output type will the be same as ‘left’, if it is a subclass of DataFrame.

See also:

merge_ordered, merge_asof

Examples

```python
>>> A
lkey value
0 foo 1
1 bar 2
2 baz 3
3 foo 4

>>> B
rkey value
0 foo 5
1 bar 6
2 qux 7

>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
lkey value_x  rkey value_y
0 foo 1      foo 5
1 foo 4      foo 5
2 bar 2      bar 6
3 bar 2      bar 8
4 baz 3      NaN NaN
5 NaN  NaN    qux 7
```

pandas.DataFrame.min

DataFrame.min(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

This method returns the minimum of the values in the object. If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.
pandas.DataFrame.mod

DataFrame.mod (other, axis='columns', level=None, fill_value=None)
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.

Parameters

other : Series, DataFrame, or constant
axis : {0, 1, ‘index’, ‘columns’}
For Series input, axis to match Series index on
fill_value : None or float value, default None
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
level : int or name
Broadcast across a level, matching Index values on the passed MultiIndex level

Returns
result : DataFrame

See also:
DataFrame.rmod

Notes
Mismatched indices will be unioned together

pandas.DataFrame.mode

DataFrame.mode (axis=0, numeric_only=False)
Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe df, you can just do this: df.fillna(df.mode().iloc[0])
Parameters

- **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
  - 0 or ‘index’: get mode of each column
  - 1 or ‘columns’: get mode of each row
- **numeric_only**: boolean, default False
  - if True, only apply to numeric columns

Returns **modes**: DataFrame (sorted)

Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
    A
0  1
1  2
```

**pandas.DataFrame.mul**

DataFrame.mul (other, axis=’columns’, level=None, fill_value=None)

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.

- **other**: Series, DataFrame, or constant
- **axis**: {0, 1, ‘index’, ‘columns’}
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result**: DataFrame

See also:

- `DataFrame.rmul`

Notes

Mismatched indices will be unioned together

**pandas.DataFrame.multiply**

DataFrame.multiply (other, axis=’columns’, level=None, fill_value=None)

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.

**Parameters**

- **other**: Series, DataFrame, or constant
- **axis**: 0, 1, ‘index’, ‘columns’
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- **result**: DataFrame

**See also:**

- `DataFrame.rmul`

**Notes**

Mismatched indices will be unioned together

**pandas.DataFrame.ne**

- **DataFrame.ne**(other, axis='columns', level=None)
  - Wrapper for flexible comparison methods ne

**pandas.DataFrame.nlargest**

- **DataFrame.nlargest**(n, columns, keep='first')
  - Get the rows of a DataFrame sorted by the n largest values of `columns`.
  - New in version 0.17.0.

**Parameters**

- **n**: int
  - Number of items to retrieve
- **columns**: list or str
  - Column name or names to order by
- **keep**: ‘first’, ‘last’, False, default ‘first’
  - Where there are duplicate values: - `first`: take the first occurrence. - `last`: take the last occurrence.

**Returns**

- **DataFrame**

**Examples**
```python
>>> df = DataFrame({'a': [1, 10, 8, 11, -1],
...                 'b': list('abdce'),
...                 'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nlargest(3, 'a')
    a  b  c
3  11  c  3
1  10  b  2
2   8  d  NaN
```

**pandas.DataFrame.notna**

DataFrame.notna()

Return a boolean same-sized object indicating if the values are not NA.

See also:

DataFrame.isna boolean inverse of notna

DataFrame.notnull alias of notna

notna top-level notna

**pandas.DataFrame.notnull**

DataFrame.notnull()

Return a boolean same-sized object indicating if the values are not NA.

See also:

DataFrame.isna boolean inverse of notna

DataFrame.notnull alias of notna

notna top-level notna

**pandas.DataFrame.nsmallest**

DataFrame.nsmallest(n, columns, keep='first')

Get the rows of a DataFrame sorted by the n smallest values of columns.

New in version 0.17.0.

Parameters

- **n**: int
  Number of items to retrieve

- **columns**: list or str
  Column name or names to order by

- **keep**: {'first', 'last', False}, default ‘first’
  Where there are duplicate values: - first: take the first occurrence. - last: take the last occurrence.

Returns

DataFrame
Examples

```python
>>> df = DataFrame({'a': [1, 10, 8, 11, -1],
                 'b': list('abdce'),
                 'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a   b   c
0  1  a  1
2  8  d  NaN
4 -1  e  4
```

**pandas.DataFrame.nunique**

DataFrame.nunique(*axis=0, dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

- **Parameters**
  - **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
  - **dropna**: boolean, default True

Don’t include NaN in the counts.

- **Returns**
  - **nunique**: Series

**Examples**

```python
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A 3
B 1
```

```python
>>> df.nunique(axis=1)
0 1
1 2
2 2
```

**pandas.DataFrame.pct_change**

DataFrame.pct_change(*periods=1, fill_method='pad', limit=None, freq=None, **kwargs*)

Percent change over given number of periods.

- **Parameters**
  - **periods**: int, default 1
  - **fill_method**: str, default ‘pad’
  - **limit**: int, default None
  - **freq**: DateOffset, timedelta, or offset alias string, optional
Increment to use from time series API (e.g. ‘M’ or BDay())

Returns chg : NDFrame

Notes

By default, the percentage change is calculated along the stat axis: 0, or Index, for DataFrame and 1, or minor for Panel. You can change this with the axis keyword argument.

**pandas.DataFrame.pipe**

DataFrame.pipe(func, *args, **kwargs)

Apply func(self, *args, **kwargs)

Parameters

- **func** : function
  - function to apply to the NDFrame. 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 NDFrame.

- **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:


Notes

Use .pipe when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```python
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
... )
```

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((f, 'arg2'), arg1=a, arg3=c)
... )
```
pandas.DataFrame.pivot

DataFrame.pivot(index=None, columns=None, values=None)
Reshape data (produce a “pivot” table) based on column values. Uses unique values from index / columns to form axes of the resulting DataFrame.

Parameters
index : string or object, optional
    Column name to use to make new frame’s index. If None, uses existing index.

columns : string or object
    Column name to use to make new frame’s columns.

values : string or object, optional
    Column name 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
pivoted : DataFrame

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

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods

Examples

```python
def = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
    'baz': [1, 2, 3, 4, 5, 6]})

def
foo  bar  baz
0    one  A  1
1    one  B  2
2    one  C  3
3    two  A  4
4    two  B  5
5    two  C  6

>>> df.pivot(index='foo', columns='bar', values='baz')
      A  B  C
one  1  2  3
two  4  5  6

>>> df.pivot(index='foo', columns='bar')['baz']
      A  B  C
one  1  2  3
two  4  5  6
```
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')

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 or list of functions, 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)

fill_value : scalar, default None

Value to replace missing values with

margins : boolean, default False

Add all row / columns (e.g. for subtotal / grand totals)

dropna : boolean, default True

Do not include columns whose entries are all NaN

margins_name : string, default ‘All’

Name of the row / column that will contain the totals when margins is True.

Returns

table : DataFrame

See also:

DataFrame.pivot pivot without aggregation that can handle non-numeric data

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", "small", "small",
    "large"],
    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
```
>>> df
   A     B   C     D
0 foo  one  small  1
1 foo  one  large  2
2 foo  one  large  2
3 foo  two  small  3
4 foo  two  small  3
5 bar  one  large  4
6 bar  one  small  5
7 bar  two  small  6
8 bar  two  large  7

>>> table = pivot_table(df, values='D', index=['A', 'B'],
                      columns=['C'], aggfunc=np.sum)

>>> table
     C     large      small
   A     B
bar one  4.0     5.0
    two  7.0     6.0
foo one  4.0     1.0
    two  NaN     6.0

pandas.DataFrame.plot

DataFrame.plot(x=None, y=None, kind='line', ax=None, subplots=False, sharex=None, sharey=False, layout=None, figsize=None, use_index=True, title=None, grid=None, legend=True, style=None, logx=False, logy=False, loglog=False, xticks=None, yticks=None, xlim=None, ylim=None, rot=None, fontsize=None, colormap=None, table=False, yerr=None, xerr=None, secondary_y=False, sort_columns=False, **kwds)

Make plots of DataFrame using matplotlib / pylab.

New in version 0.17.0: Each plot kind has a corresponding method on the DataFrame.plot accessor: df.plot(kind='line') is equivalent to df.plot.line().

Parameters data : DataFrame

x : label or position, default None

y : label or position, default None

   Allows plotting of one column versus another

kind : str

   ‘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
• ‘hexbin’ : hexbin plot

**ax** : matplotlib axes object, default None

**subplots** : boolean, default False

Make separate subplots for each column

**sharex** : boolean, 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** : boolean, 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

**use_index** : boolean, default True

Use index as ticks for x axis

**title** : string 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** : boolean, default None (matlab style default)

Axis grid lines

**legend** : False/True/’reverse’

Place legend on axis subplots

**style** : list or dict

matplotlib line style per column

**logx** : boolean, default False

Use log scaling on x axis

**logy** : boolean, default False

Use log scaling on y axis

**loglog** : boolean, default False

Use log scaling on both x and y axes

**xticks** : sequence

Values to use for the xticks

**yticks** : sequence

Values to use for the yticks
**xlim** : 2-tuple/list
**ylim** : 2-tuple/list
**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** : boolean, 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** : boolean, 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** : same types as yerr.
**stacked** : boolean, default False in line and bar plots, and True in area plot. If True, create stacked plot.
**sort_columns** : boolean, default False
  Sort column names to determine plot ordering
**secondary_y** : boolean 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** : boolean, default True
  When using a secondary_y axis, automatically mark the column labels with “(right)” in the legend
**kwds** : keywords
  Options to pass to matplotlib plotting method

**Returns axes** : matplotlib.AxesSubplot or np.array of them

**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)

• If `kind` = ‘scatter’ and the argument `c` is the name of a dataframe column, the values of that column are used to color each point.

• If `kind` = ‘hexbin’, you can control the size of the bins with the `gridsize` argument. 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`).

pandas.DataFrame.pop

DataFrame.pop(item)
Return item and drop from frame. Raise KeyError if not found.

Parameters item : str
Column label to be popped

Returns popped : 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'])
>>> df
          name  class  max_speed
0       falcon   bird       389.0
1       parrot   bird        24.0
2        lion  mammal       80.5
3      monkey  mammal         NaN

>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object

>>> df
          name  max_speed
0       falcon       389.0
1       parrot       24.0
2        lion       80.5
3      monkey 9 NaN
```
pandas.DataFrame.pow

DataFrame.pow(other, axis='columns', level=None, fill_value=None)

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.

Parameters
other : Series, DataFrame, or constant

axis : {0, 1, ‘index’, ‘columns’}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns
result : DataFrame

See also:

DataFrame.rpow

Notes

Mismatched indices will be unioned together

pandas.DataFrame.prod

DataFrame.prod(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return the product of the values for the requested axis

Parameters
axis : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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
prod : Series or DataFrame (if level specified)
pandas.DataFrame.product

DataFrame.product(\text{axis=None, skipna=None, level=None, numeric_only=None, **kwargs})

Return the product of the values for the requested axis

\textbf{Parameters} \text{axis: \{index (0), columns (1)\}}

\text{skipna: boolean, default True}

Exclude NA/null values. If an entire row/column is NA or empty, the result will be NA

\text{level: int or level name, default None}

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

\text{numeric_only: boolean, default None}

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

\textbf{Returns} \text{prod: Series or DataFrame (if level specified)}

pandas.DataFrame.quantile

DataFrame.quantile(q=0.5, \text{axis=0, numeric_only=True, interpolation='linear'})

Return values at the given quantile over requested axis, a la numpy.percentile.

\textbf{Parameters} \text{q: float or array-like, default 0.5 (50\% quantile)}

0 <= q <= 1, the quantile(s) to compute

\text{axis: \{0, 1, ‘index’, ‘columns’\} (default 0)}

0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise


New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points \(i\) and \(j\):

\begin{itemize}
  \item linear: \(i + (j - i) \times \text{fraction}\), where \text{fraction} is the fractional part of the index surrounded by \(i\) and \(j\).
  \item lower: \(i\).
  \item higher: \(j\).
  \item nearest: \(i\) or \(j\) whichever is nearest.
  \item midpoint: \((i + j)/2\).
\end{itemize}

\textbf{Returns} \text{quantiles: Series or DataFrame}

\begin{itemize}
  \item 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.
  \item If \(q\) is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.
\end{itemize}
Examples

```python
>>> df = DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                  columns=['a', 'b'])
>>> df.quantile(.1)
a  1.3
b  3.7
dtype: float64
>>> df.quantile([.1, .5])
   a   b
0.1  1.3  3.7
0.5  2.5 55.0
```

**pandas.DataFrame.query**

DataFrame.query(expr, inplace=False, **kwargs)

Query the columns of a frame with a boolean expression.

**Parameters**

- **expr**: string
  The query string to evaluate. You can refer to variables in the environment by prefixing them with an `@` character like `@a + b`.

- **inplace**: bool
  Whether the query should modify the data in place or return a modified copy

- **kwargs**: dict
  See the documentation for pandas.eval() for complete details on the keyword arguments accepted by DataFrame.query().

**Returns**

- **q**: DataFrame

**See also**:

pandas.eval, DataFrame.eval

**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 pandas.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.

For further details and examples see the `query` documentation in `indexing`.

**Examples**

```python
>>> from numpy.random import randn
>>> from pandas import DataFrame

>>> df = DataFrame(randn(10, 2), columns=list('ab'))

>>> df.query('a > b')

>>> df[df.a > df.b]  # same result as the previous expression
```

**pandas.DataFrame.radd**

DataFrame `.radd` *(other*, axis=`'columns'`, level=None, fill_value=None) 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.

**Parameters**

- **other**: Series, DataFrame, or constant
- **axis**: {0, 1, ‘index’, ‘columns’}
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- **result**: DataFrame

**See also:**

`DataFrame.add`

**Notes**

Mismatched indices will be unioned together

**pandas.DataFrame.rank**

DataFrame `.rank` *(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)*

Compute numerical data ranks (1 through n) along axis. 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'\}
- 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

numeric_only : boolean, default None
- Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option : \{'keep', 'top', 'bottom'\}
- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending : boolean, default True
- False for ranks by high (1) to low (N)

pct : boolean, default False
- Computes percentage rank of data

Returns ranks : same type as caller

pandas.DataFrame.rdiv

DataFrame.rdiv(other, axis='columns', level=None, fill_value=None)
- 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.

Parameters other : Series, DataFrame, or constant
- axis : \{0, 1, ‘index’, ‘columns’\}
  - For Series input, axis to match Series index on
- fill_value : None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- level : int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : DataFrame

See also:

DataFrame.truediv
Notes

Mismatched indices will be unioned together

**pandas.DataFrame.reindex**

DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=nan, limit=None, tolerance=None)

Conform DataFrame to new index with optional filling logic, placing 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**

- **labels**: array-like, optional
  
  New labels / index to conform the axis specified by ‘axis’ to.

- **index, columns**: array-like, optional (should be specified using keywords)
  
  New labels / index to conform to. Preferably an Index object to avoid duplicating data

- **axis**: int or str, optional
  
  Axis to target. Can be either the axis name (‘index’, ‘columns’) or number (0, 1).

- **method**: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}, optional
  
  method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
  
  - 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**: boolean, 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 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.

New in version 0.17.0.

New in version 0.21.0: (list-like tolerance)

**Returns** reindexed: 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       200.00       0.040
Chrome        200.00       0.020
Safari        404.00       0.070
IE10          404.00       0.080
Konqueror     301.00       1.000
```

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)
http_status  response_time
Safari       404.00       0.070
Iceweasel    NaN           NaN
Comodo Dragon NaN           NaN
IE10         404.00       0.080
Chrome       200.00       0.020
```

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       404.00       0.070
Iceweasel    0.000         0.000
Comodo Dragon 0.000        0.000
IE10         404.00       0.080
Chrome       200.00       0.020
```
We can also reindex the columns.

```python
df.reindex(columns=['http_status', 'user_agent'])
```

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

Or we can use “axis-style” keyword arguments

```python
df.reindex(['http_status', 'user_agent'], axis="columns")
```

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

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.reindex(date_index2)
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2009-12-29</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-30</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-31</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-02</td>
<td>101</td>
</tr>
<tr>
<td>2010-01-03</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>89</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>88</td>
</tr>
</tbody>
</table>
```

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)
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2009-12-29</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-30</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-31</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-02</td>
<td>101</td>
</tr>
<tr>
<td>2010-01-03</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>89</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>88</td>
</tr>
<tr>
<td>2010-01-07</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```
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 backpropagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.

```python
cpy2.reindex(date_index2, method='bfill')
prices
2009-12-29    100
2009-12-30    100
2009-12-31    100
2010-01-01    100
2010-01-02    101
2010-01-03    NaN
2010-01-04    100
2010-01-05    89
2010-01-06    88
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.DataFrame.reindex_axis**

DataFrame.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=nan)

Conform input object to new index with optional filling logic, placing 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**

labels : array-like

New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or ‘index’, 1 or ‘columns’}


Method to use for filling holes in reindexed DataFrame:

• 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 : boolean, 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

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 $|\text{index}[\text{indexer}] - \text{target}| \leq \text{tolerance}$. 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.

New in version 0.17.0.
New in version 0.21.0: (list-like tolerance)

**Returns** *reindexed*: DataFrame

**See also:**
*reindex, reindex_like*

### Examples

```python
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

**pandas.DataFrame.reindex_like**

DataFrame.**reindex_like**(other, method=None, copy=True, limit=None, tolerance=None)
Return an object with matching indices to myself.

**Parameters**

*other*: Object

*method*: string or None

*copy*: boolean, default True

*limit*: int, default None

  Maximum number of consecutive labels to fill for inexact matches.

*tolerance*: optional

  Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.17.0.
New in version 0.21.0: (list-like tolerance)

**Returns** *reindexed*: same as input

**Notes**

Like calling s.reindex(index=other.index, columns=other.columns, method=...)
DataFrame.rename

DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None)

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, index, columns : dict-like or function, optional
dict-like or functions transformations to apply to that axis’ values. Use either mapper and axis to specify the axis to target with mapper, or index and columns.

axis : int or str, optional
Axis to target with mapper. Can be either the axis name (‘index’, ‘columns’) or number (0, 1). The default is ‘index’.

copy : boolean, default True
Also copy underlying data

inplace : boolean, 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.

Returns renamed : DataFrame

See also:
pandas.DataFrame.rename_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.

>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6

>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
Using axis-style parameters

```python
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6

>>> df.rename({1: 2, 2: 4}, axis='index')
   0  1  4
   2  2  5
   4  3  6
```

**pandas.DataFrame.rename_axis**

The `pandas.DataFrame.rename_axis` function is used to alter the name of the index or columns.

```python
df.rename_axis("foo")
```

### Parameters
- **mapper**: scalar, list-like, optional
  - Value to set the axis name attribute.
- **axis**: int or string, default 0
- **copy**: boolean, default True
  - Also copy underlying data
- **inplace**: boolean, default False

### Returns
- **renamed**: type of caller or None if inplace=True

### See also:
- `pandas.Series.rename`
- `pandas.DataFrame.rename`
- `pandas.Index.rename`

### Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

### Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
>>> df.rename_axis("foo")
   A  B
foo 0  1  4
    1  2  5
    2  3  6

>>> df.rename_axis("bar", axis="columns")
   A  B
bar 0  1  4
```
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**: int
  - Where to reorder levels.

Returns

type of caller (new object)

pandas.DataFrame.replace

DataFrame.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)
Replace values given in 'to_replace' with 'value'.

Parameters

- **to_replace**: str, regex, list, dict, Series, numeric, or None
  - str or regex:
    - 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 and regex rules apply as above.
  - dict:
    - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.
    - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - 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 use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace**: boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**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. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method**: string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a list.

**Returns filled**: NDFrame

**Raises**

AssertionError

- If regex is not a bool and `to_replace` is not None.

TypeError

- 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.

ValueError

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

**See also:**

NDFrame.reindex, NDFrame.asfreq, NDFrame.fillna

**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.
pandas.DataFrame.resample

DataFrame.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

**Parameters**

- **rule**: string
  - the offset string or object representing target conversion
- **axis**: int, optional, default 0
- **closed**: {'right', 'left'}
  - 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'}
  - 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'}
  - For PeriodIndex only, controls whether to use the start or end of rule
- **loffset**: timedelta
  - Adjust the resampled time labels
- **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
- **on**: string, optional
  - For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.
  - New in version 0.19.0
- **level**: string or int, optional
  - For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.
  - New in version 0.19.0

**Notes**

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
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
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.

```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(array_like):
...     return np.sum(array_like)+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.

```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

Resample by month using `start` convention. Values are assigned to the first month of the period.

```python
>>> s.resample('M', convention='start').asfreq().head()
2012-01  1.0
2012-02  NaN
2012-03  NaN
2012-04  NaN
2012-05  NaN
Freq: M, dtype: float64
```

Resample by month using `end` convention. Values are assigned to the last month of the period.

```python
>>> s.resample('M', convention='end').asfreq()
2012-12  1.0
2013-01  NaN
2013-02  NaN
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

2013-03  NaN
2013-04  NaN
2013-05  NaN
2013-06  NaN
2013-07  NaN
2013-08  NaN
2013-09  NaN
2013-10  NaN
2013-11  NaN
2013-12  2.0
Freq: M,  dtype: float64

For DataFrame objects, the keyword **on** can be used to specify the column instead of the index for resampling.

```python
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()

   a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword **level** can be used to specify on level the resampling needs to take place.

```python
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)], columns=['a', 'b', 'c', 'd'], index=pd.MultiIndex.from_product([time, [1, 2]])
>>> df2.resample('3T', level=0).sum()

   a  b  c  d
2000-01-01 00:00:00  0  6 12 18
2000-01-01 00:03:00  0  4  8 12
```

### pandas.DataFrame.reset_index

DataFrame.reset_index (level=None, drop=False, inplace=False, col_level=0, col_fill='')
For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to ‘level_0’, ‘level_1’, etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default ‘index’ or ‘level_0’ (if ‘index’ is already taken) will be used.

**Parameters**

- **level**: int, str, tuple, or list, default None
  - Only remove the given levels from the index. Removes all levels by default
- **drop**: boolean, default False
  - Do not try to insert index into dataframe columns. This resets the index to the default integer index.
- **inplace**: boolean, 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** resetted : 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'))
>>> df
    class  max_speed
falcon  bird    389.0
parrot  bird     24.0
lion    mammal   80.5
monkey  mammal    NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```python
>>> df.reset_index()
   index     class  max_speed
0   falcon  bird    389.0
1   parrot  bird     24.0
2    lion  mammal   80.5
3  monkey  mammal    NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```python
>>> df.reset_index(drop=True)
   class  max_speed
0  bird    389.0
1  bird     24.0
2 mammal   80.5
3 mammal    NaN
```

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)
```

```
0      bird 389.0
1      bird  24.0
2  mammal  80.5
3  mammal    NaN
```
If the index has multiple levels, we can reset a subset of them:

```python
>>> df.reset_index(level='class')
  class  speed   species
       max     type
  name
  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:

```python
>>> df.reset_index(level='class', col_level=1)
  speed   species
  class   max     type
  name
  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`:

```python
>>> df.reset_index(level='class', col_level=1, col_fill='species')
  species   speed  species
  class     max     type
  name
  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:

```python
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
  genus   speed  species
  class   max     type
  name
  falcon     bird  389.0   fly
  parrot     bird   24.0   fly
  lion       mammal  80.5   run
  monkey     mammal   NaN   jump
```
pandas.DataFrame.rfloordiv

Dataframe.rfloordiv(other, axis='columns', level=None, fill_value=None)

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.

**Parameters**

- **other**: Series, DataFrame, or constant
- **axis**: `0, 1, 'index', 'columns'`
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- **result**: DataFrame

**See also**

Dataframe.floordiv

**Notes**

Mismatched indices will be unioned together

pandas.DataFrame.rmod

Dataframe.rmod(other, axis='columns', level=None, fill_value=None)

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.

**Parameters**

- **other**: Series, DataFrame, or constant
- **axis**: `0, 1, 'index', 'columns'`
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- **level**: int or name
  - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- **result**: DataFrame

**See also**

Dataframe.mod
Notes

Mismatched indices will be unioned together

pandas.DataFrame.rmul

DataFrame.rmul(other, axis='columns', level=None, fill_value=None)

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.

Parameters
other : Series, DataFrame, or constant

axis : {0, 1, ‘index’, ‘columns’}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns
result : DataFrame

See also:
DataFrame.mul

Notes

Mismatched indices will be unioned together

pandas.DataFrame.rolling

DataFrame.rolling(window, min_periods=None, freq=None, center=False, win_type=None, on=None, axis=0, closed=None)

Provides rolling window calculations.

New in version 0.18.0.

Parameters
window : int, or offset

Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be 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. This is new in 0.19.0

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

freq : string or DateOffset object, optional (default None)
**center** : boolean, default False

Set the labels at the center of the window.

**win_type** : string, default None

Provide a window type. See the notes below.

**on** : string, optional

For a DataFrame, column on which to calculate the rolling window, rather than the index

**closed** : string, default None

Make the interval closed on the ‘right’, ‘left’, ‘both’ or ‘neither’ endpoints. For offset-based windows, it defaults to ‘right’. For fixed windows, defaults to ‘both’. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

**axis** : int or string, default 0

**Returns** a Window or Rolling sub-classed for the particular operation

**Notes**

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The `freq` keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the `mean`).

To learn more about the offsets & frequency strings, please see this link.

The recognized `win_types` are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).
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

Rolling sum with a window length of 2, using the 'triang' window type.

```python
>>> df.rolling(2, win_type='triang').sum()
   B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```python
>>> df.rolling(2).sum()
   B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```python
>>> df.rolling(2, min_periods=1).sum()
   B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```python
>>> df = 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
   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
```

34.4. DataFrame
Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```python
>>> df.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
```

**pandas.DataFrame.round**

DataFrames and Series provide a method for coping with different types of data. For example, dates are often used in financial data. When dealing with financial data, having columns that contain date information as well as numeric values can make it easier to perform calculations. Pandas allows for this by providing methods that can perform calculations on columns that contain dates. For example, the above code shows how to calculate the rolling sum of a DataFrame containing date information and numeric values.

## Examples

```python
>>> df = pd.DataFrame(np.random.random([3, 3]),
...                   columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
   A     B     C
first 0.028208 0.992815 0.173891
second 0.038683 0.645646 0.577595
third 0.877076 0.149370 0.491027
>>> df.round(2)
   A     B     C
first 0.03 0.99 0.17
second 0.04 0.65 0.58
third 0.88 0.15 0.49
>>> df.round({'A': 1, 'C': 2})
   A     B     C
first 0.0 0.992815 0.17
second 0.0 0.645646 0.58
third 0.9 0.149370 0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
   A     B     C
first 0.0 1.0 0.17
**pandas.DataFrame.rpow**

DataFrame\[rpow\](other, axis='columns', level=None, fill_value=None)  
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.  

**Parameters**  
other : Series, DataFrame, or constant  
axis : {0, 1, ‘index’, ‘columns’}  
For Series input, axis to match Series index on  
fill_value : None or float value, default None  
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing  
level : int or name  
Broadcast across a level, matching Index values on the passed MultiIndex level  

**Returns**  
result : DataFrame  

**See also:**  
DataFrame\[pow\]

**Notes**  
Mismatched indices will be unioned together

**pandas.DataFrame.rsub**

DataFrame\[rsub\](other, axis='columns', level=None, fill_value=None)  
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.  

**Parameters**  
other : Series, DataFrame, or constant  
axis : {0, 1, ‘index’, ‘columns’}  
For Series input, axis to match Series index on  
fill_value : None or float value, default None  
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing  
level : int or name  
Broadcast across a level, matching Index values on the passed MultiIndex level  

**Returns**  
result : DataFrame
See also:

```
DataFrame.sub
```

Notes

Mismatched indices will be unioned together

**pandas.DataFrame.rtruediv**

```
DataFrame.rtruediv(other, axis='columns', level=None, fill_value=None)
```

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.

**Parameters**

- `other` : Series, DataFrame, or constant
- `axis` : {0, 1, ‘index’, ‘columns’}
  
  For Series input, axis to match Series index on

- `fill_value` : None or float value, default None
  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

- `level` : int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

- `result` : DataFrame

See also:

```
DataFrame.truediv
```

Notes

Mismatched indices will be unioned together

**pandas.DataFrame.sample**

```
DataFrame.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)
```

Returns a random sample of items from an axis of object.

**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` : boolean, optional
  
  Sample with or without replacement. Default = False.
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. inf and -inf values not allowed.

random_state : int or numpy.random.RandomState, optional

Seed for the random number generator (if int), or numpy RandomState object.

axis : int or string, optional

Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns A new object of same type as caller.

Examples

Generate an example Series and DataFrame:

```python
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0    -0.038497
1     1.820773
2     -0.972766
3     -1.598270
4     -1.095526
dtype: float64

>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
   A          B          C          D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the Series:

```python
>>> s.sample(n=3)
-0.994689
-1.049016
-0.224565
dtype: float64
```

And a random 10% of the DataFrame with replacement:

```python
>>> df.sample(frac=0.1, replace=True)
   A          B          C          D
35  1.981780  0.142106  1.817165  -0.290805
49 -1.336199 -0.448634  0.789640   0.217116
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0.823173</td>
<td>-0.078816</td>
<td>1.009536</td>
<td>1.015108</td>
</tr>
<tr>
<td>15</td>
<td>1.421154</td>
<td>-0.055301</td>
<td>-1.922594</td>
<td>-0.019696</td>
</tr>
<tr>
<td>6</td>
<td>-0.148339</td>
<td>0.832938</td>
<td>1.787600</td>
<td>-1.383767</td>
</tr>
</tbody>
</table>

### pandas.DataFrame.select

**DataFrame.select** *(crit, axis=0)*

Return data corresponding to axis labels matching criteria

DEPRECATED: use df.loc[df.index.map(crit)] to select via labels

**Parameters**

- **crit**: function
  
  To be called on each index (label). Should return True or False

- **axis**: int

**Returns**

- **selection**: type of caller

### pandas.DataFrame.select_dtypes

**DataFrame.select_dtypes** *(include=None, exclude=None)*

Return a subset of a DataFrame including/excluding columns based on their dtype.

**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**

- **subset**: 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.

**Notes**

- To select all numeric types use the numpy dtype `numpy.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 a ‘datetime64[ns, tz]’ string
Examples

```python
>>> df = pd.DataFrame({'a': np.random.randn(6).astype('f4'),
                    'b': [True, False] * 3,
                    'c': [1.0, 2.0] * 3})
>>> df
   a      b      c
0  0.3962  True  1.0
1  0.1459 False  2.0
2  0.2623  True  1.0
3  0.0764 False  2.0
4 -0.9703  True  1.0
5 -1.2094 False  2.0

>>> df.select_dtypes(include='bool')
   c
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=['floating'])
   b
0  True
1 False
2 True
3 False
4 True
5 False
```

**pandas.DataFrame.sem**

DataFrame.sem(axis=None, skipna=None, 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**: boolean, 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
  - degrees of freedom
numeric_only : boolean, 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 sem : Series or DataFrame (if level specified)

pandas.DataFrame.set_axis

DataFrame.set_axis(labels, axis=0, inplace=None)
Assign desired index to given axis

Parameters labels: list-like or Index

The values for the new index

axis : int or string, default 0

inplace : boolean, default None

Whether to return a new NDFrame instance.

WARNING: inplace=None currently falls back to True, but in a future ver-
version, will default to False. Use inplace=True explicitly rather than relying on the
default.

.. versionadded:: 0.21.0

The signature is make consistent to the rest of the API. Previously, the “axis” and
“labels” arguments were respectively the first and second positional arguments.

Returns renamed : NDFrame or None

An object of same type as caller if inplace=False, None otherwise.

See also:

pandas.NDFrame.rename

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, inplace=False)
a  1
b  2
c  3
dtype: int64
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
>>> df.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
>>> df.set_axis(['I', 'II'], axis=1, inplace=False)
   I  II
```

```
0 1 4
1 2 5
2 3 6
>>> df.set_axis(['i', 'ii'], axis=1, inplace=True)
>>> df
  i   ii
0 1 4
1 2 5
2 3 6
```

### pandas.DataFrame.set_index

DataFrame.set_index(*keys*, *drop=True, append=False, inplace=False, verify_integrity=False)*

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

**Parameters**

- **keys**: column label or list of column labels / arrays
- **drop**: boolean, default True
  
  Delete columns to be used as the new index
- **append**: boolean, default False
  
  Whether to append columns to existing index
- **inplace**: boolean, default False
  
  Modify the DataFrame in place (do not create a new object)
- **verify_integrity**: boolean, 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**: DataFrame

### Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
```

```
 month sale year
 0 1  55  2012
 1 4  40  2014
 2 7  84  2013
 3 10 31  2014
```

Set the index to become the ‘month’ column:

```
>>> df.set_index('month')
```

```
    sale year
month
 1  55  2012
 4  40  2014
 7  84  2013
 10 31  2014
```
Create a multi-index 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 multi-index using a set of values and a column:

```python
>>> df.set_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
```

`pandas.DataFrame.set_value`

Dataframe.

set_value

*DataFrame.set_value(index, col, value, takeable=False)*

Put single value at passed column and index

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

**Parameters**

- **index**: row label
- **col**: column label
- **value**: scalar value
- **takeable**: interpret the index/col as indexers, default False

**Returns**

- **frame**: DataFrame

If label pair is contained, will be reference to calling DataFrame, otherwise a new object

`pandas.DataFrame.shift`

Dataframe.

shift

*DataFrame.shift(periods=1, freq=None, axis=0)*

Shift index by desired number of periods with an optional time freq

**Parameters**

- **periods**: int
  - Number of periods to move, can be positive or negative
- **freq**: DateOffset, timedelta, or time rule string, optional
  - Increment to use from the tseries module or time rule (e.g. ‘EOM’). See Notes.
- **axis**: [0 or ‘index’, 1 or ‘columns’]

**Returns**

- **shifted**: DataFrame
Notes

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.

**pandas.DataFrame.skew**

DataFrame.skew(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return unbiased skew over requested axis Normalized by N-1

**Parameters**

- **axis**: {index (0), columns (1)}
  - `skipna` : boolean, default True
    - Exclude NA/null values. If an entire row/column is NA or empty, 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
  - `numeric_only` : boolean, 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**

- `skew` : Series or DataFrame (if level specified)

**pandas.DataFrame.slice_shift**

DataFrame.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**

DataFrame.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, by=None)

Sort object by labels (along an axis)

**Parameters**

- **axis** : index, columns to direct sorting
  - `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 : boolean, default True
    Sort ascending vs. descending
inplace : bool, default False
    if True, perform operation in-place
kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
    Choice of sorting algorithm. See also nd.array.sort for more information. 
    mergesort is the only stable algorithm. For DataFrames, this option is only applied 
    when sorting on a single column or label.
na_position : {'first', 'last'}, default 'last'
    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

Returns sorted_obj : DataFrame

pandas.DataFrame.sort_values

Dataframe.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', 
na_position='last')
Sort by the values along either axis
New in version 0.17.0.

Parameters by : str or list of str
    Name or list of names which refer to the axis items.
axis : {0 or ‘index’, 1 or ‘columns’}, default 0
    Axis to direct sorting
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'}, default ‘quicksort’
    Choice of sorting algorithm. See also nd.array.sort for more information. 
    mergesort is the only stable algorithm. For DataFrames, this option is only applied 
    when sorting on a single column or label.
na_position : {'first', 'last'}, default ‘last’
    first puts NaNs at the beginning, last puts NaNs at the end

Returns sorted_obj : DataFrame
### 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],
... })
```

```text
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Sort by `col1`

```python
>>> df.sort_values(by='col1')
```

```text
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Sort by multiple columns

```python
>>> df.sort_values(by=['col1', 'col2'])
```

```text
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Sort Descending

```python
>>> df.sort_values(by='col1', ascending=False)
```

```text
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>D</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
```

Putting NAs first

```python
>>> df.sort_values(by='col1', ascending=False, na_position='first')
```

```text
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>NaN</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```

34.4. DataFrame
### pandas.DataFrame.sortlevel

**DataFrame.sortlevel** \( (\text{level}=0, \text{axis}=0, \text{ascending}=\text{True}, \text{inplace}=\text{False}, \text{sort_remaining}=\text{True}) \)

DEPRECATED: use **DataFrame.sort_index()**

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

**Parameters**
- **level**: int
- **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
- **ascending**: boolean, default True
- **inplace**: boolean, default False
  - Sort the DataFrame without creating a new instance
- **sort_remaining**: boolean, default True
  - Sort by the other levels too.

**Returns**
- **sorted**: DataFrame

See also:
- **DataFrame.sort_index**

### pandas.DataFrame.squeeze

**DataFrame.squeeze** \( (\text{axis}=\text{None}) \)

Squeeze length 1 dimensions.

**Parameters**
- **axis**: None, integer or string axis name, optional
  - The axis to squeeze if 1-sized.
  - New in version 0.20.0.

**Returns**
- scalar if 1-sized, else original object

### pandas.DataFrame.stack

**DataFrame.stack** \( (\text{level}=-1, \text{dropna}=\text{True}) \)

Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels. The level involved will automatically get sorted.

**Parameters**
- **level**: int, string, or list of these, default last level
  - Level(s) to stack, can pass level name
- **dropna**: boolean, default True
  - Whether to drop rows in the resulting Frame/Series with no valid values

**Returns**
- **stacked**: DataFrame or Series
Examples

```python
>>> s
   a  b
one 1  2.
two 3  4.

>>> s.stack()
   a  1
   b  2
   a  3
   b  4
```

`pandas.DataFrame.std`

DataFrame.std (axis=None, skipna=None, 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**: boolean, 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
  - degrees of freedom
- **numeric_only**: boolean, 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**

- **std**: Series or DataFrame (if level specified)

`pandas.DataFrame.sub`

DataFrame.sub (other, axis='columns', level=None, fill_value=None)

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.

**Parameters**

- **other**: Series, DataFrame, or constant
- **axis**: {0, 1, ‘index’, ‘columns’}
  - For Series input, axis to match Series index on
- **fill_value**: None or float value, default None
  - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result**: DataFrame

See also:

*DataFrame.rsub*

Notes

Mismatched indices will be unioned together

### pandas.DataFrame.subtract

DataFrame\.subtract\(\text{other, axis='columns', level=None, fill_value=None}\)

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.

Parameters **other**: Series, DataFrame, or constant

- **axis**: {0, 1, ‘index’, ‘columns’}
  
  For Series input, axis to match Series index on

- **fill_value**: None or float value, default None
  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

- **level**: int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result**: DataFrame

See also:

*DataFrame.rsub*

Notes

Mismatched indices will be unioned together

### pandas.DataFrame.sum

DataFrame\.sum\(\text{axis=None, skipna=None, level=None, numeric_only=None, **kwargs}\)

Return the sum of the values for the requested axis

Parameters **axis**: {index (0), columns (1)}

- **skipna**: boolean, default True
  
  Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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 sum : Series or DataFrame (if level specified)

pandas.DataFrame.swapaxes

DataFrame.swapaxes(axis1, axis2, copy=True)

Interchange axes and swap values axes appropriately

Returns y : same as input

pandas.DataFrame.swaplevel

DataFrame.swaplevel(i=-2, j=-1, axis=0)

Swap levels i and j in a MultiIndex on a particular axis

Parameters i, j : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns swapped : type of caller (new object)

Changed in version 0.18.1: The indexes i and j are now optional, and default to the two
innermost levels of the index.

pandas.DataFrame.tail

DataFrame.tail(n=5)

Return the last n rows.

Parameters n : int, default 5

Number of rows to select.

Returns obj_tail : type of caller

The last n rows of the caller object.

pandas.DataFrame.take

DataFrame.take(indices, axis=0, convert=None, is_copy=True, **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 : int, default 0

The axis on which to select elements. “0” means that we are selecting rows, “1” means that we are selecting columns, etc.

convert : bool, default True

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

Whether to convert negative indices into positive ones. For example, –1 would map to the len(axis) – 1. The conversions are similar to the behavior of indexing a regular Python list.

is_copy : bool, default True

Whether to return a copy of the original object or not.

Returns taken : type of caller

An array-like containing the elements taken from the object.

See also:

numpy.ndarray.take, numpy.take

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
name
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.

>>> df.take([0, 3])
    name    class   max_speed
name
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.DataFrame.to_clipboard**

`Dataframe.to_clipboard(excel=None, sep=None, **kwargs)`

Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

- **Parameters**
  - `excel`: boolean, defaults to True
    - if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard
  - `sep`: optional, defaults to tab
  - other keywords are passed to `to_csv`

**Notes**

**Requirements for your platform**

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

**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='
', chunksize=None, tupleize_cols=None, date_format=None, doublequote=True, escapechar=None, decimal='.')`

Write DataFrame to a comma-separated values (csv) file

- **Parameters**
  - `path_or_buf`: string or file handle, default None
  - `sep`: character, default ‘,’
    - Field delimiter for the output file.
  - `na_rep`: string, default “”
    - Missing data representation
  - `float_format`: string, default None
    - Format string for floating point numbers
  - `columns`: sequence, optional
Columns to write

**header** : boolean or list of string, 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** : boolean, default True
Write row names (index)

**index_label** : string 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 DataFrame 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** : string, optional
A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

**compression** : string, optional
A string representing the compression to use in the output file, allowed values are ‘gzip’, ‘bz2’, ‘xz’, only used when the first argument is a filename

**line_terminator** : string, default ‘
’
The newline character or character sequence to use in the output file

**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** : string (length 1), default ‘”’
character used to quote fields

**doublequote** : boolean, default True
Control quoting of quotechar inside a field

**escapechar** : string (length 1), default None
character used to escape sep and quotechar when appropriate

**chunksize** : int or None
rows to write at a time

**tupleize_cols** : boolean, default False
Depreciated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

**date_format** : string, default None
Format string for datetime objects

**decimal: string, default ‘.’**

Character recognized as decimal separator. E.g. use ‘,’ for European data

**pandas.DataFrame.to_dense**

DataFrame.to_dense()

Return dense representation of NDFrame (as opposed to sparse)

**pandas.DataFrame.to_dict**

DataFrame.to_dict(orient='dict', into=<class 'dict'>)

Convert DataFrame to dictionary.

**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]}
  - records : list like [{column -> value}, ... , {column -> value}]
  - index : dict like {index -> {column -> value}}

  New in version 0.17.0.

  Abbreviations are allowed. s indicates series and sp indicates split.

- **into**: class, default dict

  The collections.Mapping subclass used for all Mappings in the return value. 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.

  New in version 0.21.0.

**Returns**

- **result**: collections.Mapping like {column -> {index -> value}}

**Examples**

```python
>>> df = pd.DataFrame(
    {'col1': [1, 2], 'col2': [0.5, 0.75]}, index=['a', 'b'])
>>> df
coll  col2
a    1   0.5
b    2   0.7

>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.
>>> df.to_dict('series')
{'col1': 1
b 2
Name: col1, dtype: int64, 'col2': a 0.50
b 0.75
Name: col2, dtype: float64}

>>> df.to_dict('split')
{'columns': ['col1', 'col2'],
'data': [[1.0, 0.5], [2.0, 0.75]],
'index': ['a', 'b']}

>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]

>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}

You can also specify the mapping type.

```python
>>> from collections import OrderedDict, defaultdict
```

```python
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
              ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a `defaultdict`, you need to initialize it:

```python
>>> dd = defaultdict(list)
```

```python
>>> df.to_dict('records', into=dd)
[defaultdict(<type 'list'>, {'col2': 0.5, 'col1': 1.0}),
defaultdict(<type 'list'>, {'col2': 0.75, 'col1': 2.0})]
```

### 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)

Write DataFrame to an excel sheet.

**Parameters**

- **excel_writer**: string or ExcelWriter object
  - File path or existing ExcelWriter
- **sheet_name**: string, default ‘Sheet1’
  - Name of sheet which will contain DataFrame
- **na_rep**: string, default “
  - Missing data representation
- **float_format**: string, default None
  - Format string for floating point numbers
- **columns**: sequence, optional
  - Columns to write
- **header**: boolean or list of string, 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**: boolean, default True

Write row names (index).

**index_label**: string or sequence, 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 DataFrame uses MultiIndex.

**startrow**: upper left cell row to dump data frame

**startcol**: upper left cell column to dump data frame

**engine**: string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge_cells**: boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.

**encoding**: string, default None

encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf_rep**: string, default ‘inf’

Representation for infinity (there is no native representation for infinity in Excel)

**freeze_panes**: tuple of integer (length 2), default None

Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

### Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```python
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

### pandas.DataFrame.to_feather

**DataFrame.to_feather**(filename)

write out the binary feather-format for DataFrames

New in version 0.20.0.
**Parameters**

- **fname**: str
  - string file path

**pandas.DataFrame.to_gbq**

```python
DataFrame.to_gbq(destination_table, project_id, chunksize=10000, verbose=True, reauth=False, if_exists='fail', private_key=None)
```

Write a DataFrame to a Google BigQuery table.

The main method a user calls to export pandas DataFrame contents to Google BigQuery table. Google BigQuery API Client Library v2 for Python is used. Documentation is available here

Authentication to the Google BigQuery service is via OAuth 2.0.

- If “private_key” is not provided:
  - By default “application default credentials” are used.
  - If default application credentials are not found or are restrictive, user account credentials are used.
  - In this case, you will be asked to grant permissions for product name ‘pandas GBQ’.
- If “private_key” is provided:
  - Service account credentials will be used to authenticate.

**Parameters**

- **dataframe**: DataFrame
  - DataFrame to be written

- **destination_table**: string
  - Name of table to be written, in the form ‘dataset.tablename’

- **project_id**: str
  - Google BigQuery Account project ID.

- **chunksize**: int (default 10000)
  - Number of rows to be inserted in each chunk from the dataframe.

- **verbose**: boolean (default True)
  - Show percentage complete

- **reauth**: boolean (default False)
  - Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

- **if_exists**: {'fail', 'replace', 'append'}, default ‘fail’
  - ‘fail’: If table exists, do nothing. ‘replace’: If table exists, drop it, recreate it, and insert data. ‘append’: If table exists, insert data. Create if does not exist.

- **private_key**: str (optional)
  - Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (e.g. jupyter iPython notebook on remote host)
DataFrame.to_hdf

DataFrame.to_hdf(path_or_buf, key, **kwargs)

Write the contained data to an HDF5 file using HDFStore.

Parameters:

path_or_buf : the path (string) or HDFStore object

key : string

identifier for the group in the store

mode : optional, {'a', 'w', 'r+'}, default 'a'

'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+' It is similar to 'a', but the file must already exist.

format : {'fixed(f)|table(t)'}, default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [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 : boolean, default False

For Table formats, 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.

Applicable only to format='table'.

complevel : int, 0-9, default None

Specifies a compression level for data. A value of 0 disables compression.

complib : {'zlib', 'lz4', 'gzip2', '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.

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.
pandas: powerful Python data analysis toolkit, Release 0.21.0

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, bold_rows=True, classes=None, escape=True, max_rows=None, max_cols=None, show_dimensions=False, notebook=False, decimal='.', border=None)

Render a DataFrame as an HTML table.

to_html-specific options:

- **bold_rows** [boolean, 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** [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.
- **max_rows** [int, optional] Maximum number of rows to show before truncating. If None, show all.
- **max_cols** [int, optional] Maximum number of columns to show before truncating. If None, show all.
- **decimal** [string, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe
    
    New in version 0.18.0.

- **border** [int] A border=border attribute is included in the opening <table> tag. Default pd.options.html.border.
    
    New in version 0.19.0.

**Parameters**

- **buf** : StringIO-like, optional
    
    buffer to write to

- **columns** : sequence, optional
    
    the subset of columns to write; default None writes all columns

- **col_space** : int, optional
    
    the minimum width of each column

- **header** : bool, optional
    
    whether to print column labels, default True

- **index** : bool, optional
    
    whether to print index (row) labels, default True

- **na_rep** : string, optional
    
    string representation of NaN to use, default ‘NaN’

- **formatters** : list or dict of one-parameter functions, optional
    
    formatter functions to apply to columns’ elements by position or name, default None. 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, optional
    
    formatter function to apply to columns’ elements if they are floats, default None. The result of this function must be a unicode string.

- **sparsify** : bool, optional
Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**index_names**: bool, optional
Prints the names of the indexes, default True

**line_width**: int, optional
Width to wrap a line in characters, default no wrap

How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box.

**Returns**

**formatted**: string (or unicode, depending on data and options)

### 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=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**: the path or buffer to write the result string
if this is None, return the converted string

**orient**: string
- Series
  - default is ‘index’
  - allowed values are: {‘split’,‘records’,‘index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,‘records’,‘index’,‘columns’,‘values’}
- 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, and the data component is like orient=’records’.

  Changed in version 0.20.0.

**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** : 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** : string, 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 re-
spectively.

**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** : boolean, default False
If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError
if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

**compression** : {None, ‘gzip’, ‘bz2’, ‘xz’}
A string representing the compression to use in the output file, only used when
the first argument is a filename

New in version 0.21.0.

**Returns** same type as input object with filtered info axis

**See also:**
pd.read_json

**Examples**

```python
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])

>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
 "index":["row 1","row 2"],
 "data":[['a","b"],["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"}}'
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not pre-
erved with this encoding.

```python
>>> df.to_json(orient='records')
'[["col 1":"a","col 2":"b"],"col 1":"c","col 2":"d"]'
```
Encoding with Table Schema

```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": "0.20.0"},
"data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
    {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

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, sparse=None, index_names=True, bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multicolumn_format=None, multirow=None)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series
to_latex-specific options:

- **bold_rows** [boolean, default False] Make the row labels bold in the output
- **column_format** [str, default None] The columns format as specified in LaTeX table format e.g. ‘rcl’ for 3 columns
- **longtable** [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.
- **escape** [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.
- **encoding** [str, default None] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.
- **decimal** [string, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.
  
  New in version 0.18.0.
- **multicolumn** [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.
  
  New in version 0.20.0.
- **multicolumn_format** [str, default ‘l’] The alignment for multicolumns, similar to column_format. The default will be read from the config module.
  
  New in version 0.20.0.
- **multirow** [boolean, 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.
  
  New in version 0.20.0.
pandas.DataFrame.to_msgpack

DataFrame.to_msgpack(path_or_buf=None, encoding='utf-8', **kwargs)

msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters:
- path : string
  - File path, buffer-like, or None
  - if None, return generated string
- append : boolean
  - whether to append to an existing msgpack
  - (default is False)
- compress : type of compressor (zlib or blosc), default to None (no compression)

pandas.DataFrame.to_panel

DataFrame.to_panel()

Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns:
- panel : Panel

pandas.DataFrame.to_parquet

DataFrame.to_parquet(fname, engine='auto', compression='snappy', **kwargs)

Write a DataFrame to the binary parquet format.

New in version 0.21.0.

Parameters:
- fname : str
  - string file path
- engine : {'auto', 'pyarrow', 'fastparquet'}, default 'auto'
  - Parquet reader library to use. If 'auto', then the option 'io.parquet.engine' is used.
  - If 'auto', then the first library to be installed is used.
- compression : str, optional, default 'snappy'
  - compression method, includes {'gzip', 'snappy', 'brotli'}
- kwargs : Additional keyword arguments passed to the engine

pandas.DataFrame.to_period

DataFrame.to_period(freq=None, axis=0, copy=True)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters:
- freq : string, default
  - axis : {0 or 'index', 1 or 'columns'}, default 0
The axis to convert (the index by default)

**copy** : boolean, default True

If False then underlying input data is not copied

**Returns**

**ts** : TimeSeries with PeriodIndex

### pandas.DataFrame.to_pickle

DataFrame.to_pickle (*path*, *compression*='infer', *protocol*=4)

Pickle (serialize) object to input file path.

**Parameters**

**path** : string

File path

**compression** : {'infer', 'gzip', 'bz2', 'xz', None}, default 'infer'

a string representing the compression to use in the output file

New in version 0.20.0.

**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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

### pandas.DataFrame.to_records

DataFrame.to_records (*index*=True, *convert_datetime64*=True)

Convert DataFrame to record array. Index will be put in the ‘index’ field of the record array if requested

**Parameters**

**index** : boolean, default True

Include index in resulting record array, stored in ‘index’ field

**convert_datetime64** : boolean, default True

Whether to convert the index to datetime.datetime if it is a DatetimeIndex

**Returns**

**y** : recarray

### pandas.DataFrame.to_sparse

DataFrame.to_sparse (*fill_value*=None, *kind*='block')

Convert to SparseDataFrame

**Parameters**

**fill_value** : float, default NaN

**kind** : {'block', 'integer'}

**Returns**

**y** : SparseDataFrame
A class for writing Stata binary dta files from array-like objects

**Parameters**

- **fname**: str or buffer
  
  String path of file-like object

- **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.

- **encoding**: str
  
  Default is latin-1. Unicode is not supported.

- **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.

- **dataset_label**: str
  
  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.

New in version 0.19.0.

**Raises**

- `NotImplementedError`
  
  - If datetimes contain timezone information
  - Column dtype is not representable in Stata

- `ValueError`
  
  - Columns listed in convert_dates are not datetime64[ns] or datet ime.datetime
  - Column listed in convert_dates is not in DataFrame
  - Categorical label contains more than 32,000 characters

New in version 0.19.0.
Examples

```python
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

Or with dates

```python
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

### pandas.DataFrame.to_string

DataFrame.to_string

```python
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, line_width=None, max_rows=None, max_cols=None, show_dimensions=False)
```

Render a DataFrame to a console-friendly tabular output.

**Parameters**

- **buf**: StringIO-like, optional
  
  buffer to write to

- **columns**: sequence, optional
  
  the subset of columns to write; default None writes all columns

- **col_space**: int, optional
  
  the minimum width of each column

- **header**: bool, 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
  
  whether to print index (row) labels, default True

- **na_rep**: string, optional
  
  string representation of NAN to use, default ‘NaN’

- **formatters**: list or dict of one-parameter functions, optional
  
  formatter functions to apply to columns’ elements by position or name, default None. 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, optional
  
  formatter function to apply to columns’ elements if they are floats, default None. The result of this function must be a unicode string.

- **sparsify**: bool, optional
  
  Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

- **index_names**: bool, optional
  
  Prints the names of the indexes, default True
line_width : int, optional

Width to wrap a line in characters, default no wrap

justify : {'left', 'right', 'center', 'justify',
     'justify-all', 'start', 'end', 'inherit', 'match-parent', 'initial', 'unset'}, 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.

Returns formatted : string (or unicode, depending on data and options)

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: string, 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 : boolean, default True

If false then underlying input data is not copied

Returns df : DataFrame with DatetimeIndex

pandas.DataFrame.to_xarray

DataFrame.to_xarray()

Return an xarray object from the pandas object.

Returns a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```python
>>> df = pd.DataFrame({'A' : [1, 1, 2],
         'B' : ['foo', 'bar', 'foo'],
         'C' : np.arange(4.,7))
>>> df
```
A  B  C
0  1 foo 4.0
1  1 bar 5.0
2  2 foo 6.0

```python
>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (index: 3)
Coordinates:
  * index (index) int64 0 1 2
Data variables:
  A (index) int64 1 1 2
  B (index) object 'foo' 'bar' 'foo'
  C (index) float64 4.0 5.0 6.0

>>> df = pd.DataFrame({'A' : [1, 1, 2],
                     'B' : ['foo', 'bar', 'foo'],
                     'C' : np.arange(4.,7)}.{set_index('B', 'A'))

>>> df
    C
   B A
foo 1 4.0
bar 1 5.0
foo 2 6.0

>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (A: 2, B: 2)
Coordinates:
  * B (B) object 'bar' 'foo'
  * A (A) int64 1 2
Data variables:
  C (B, A) float64 5.0 nan 4.0 6.0

>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                items=list('ABCD'),
                major_axis=pd.date_range('20130101', periods=3),
                minor_axis=['first', 'second'])

>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second

>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)
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]])
pandas.DataFrame.transform

DataFrame.transform (func, *args, **kwargs)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

Parameters func : callable, string, dictionary, or list of string/callables

To apply to column

Accepted Combinations are:

• string function name
• function
• list of functions
• dict of column names -> functions (or list of functions)

Returns transformed : NDFrame

See also:
pandas.NDFrame.aggregate, pandas.NDFrame.apply

Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                   index=pd.date_range('1/1/2000', periods=10))
>>> df.iloc[3:7] = np.nan

>>> df.transform(lambda x: (x - x.mean()) / x.std())
   A          B          C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
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 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```
pandas.DataFrame.transpose

DataFrame.transpose(*args, **kwargs)
Transpose index and columns

pandas.DataFrame.truediv

DataFrame.truediv(other, axis='columns', level=None, fill_value=None)
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.

Parameters
- other : Series, DataFrame, or constant
- axis : {0, 1, ‘index’, ‘columns’}
  For Series input, axis to match Series index on
- fill_value : None or float value, default None
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
- level : int or name
  Broadcast across a level, matching Index values on the passed MultiIndex level

Returns
- result : DataFrame

See also:
DataFrame.rtruediv

Notes
Mismatched indices will be unioned together

pandas.DataFrame.truncate

DataFrame.truncate(before=None, after=None, axis=None, copy=True)
Truncates a sorted DataFrame/Series before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters
- before : date, string, int
  Truncate all rows before this index value
- after : date, string, int
  Truncate all rows after this index value
- axis : {0 or ‘index’, 1 or ‘columns’}
  • 0 or ‘index’: apply truncation to rows
  • 1 or ‘columns’: apply truncation to columns
  Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels)
- copy : boolean, default is True,
pandas: powerful Python data analysis toolkit, Release 0.21.0

return a copy of the truncated section

Returns  truncated : type of caller

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.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

```python
>>> df = pd.DataFrame({'A': [1, 2, 3, 4, 5],
                    'B': [6, 7, 8, 9, 10],
                    'C': [11, 12, 13, 14, 15],
                    'index':['a', 'b', 'c', 'd', 'e'])
>>> df.truncate(before='b', after='d')
   A  B  C
  b 2 7 12
  c 3 8 13
  d 4 9 14
```

The index values in `truncate` can be datetimes or string dates. 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
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.truncate('2016-01-05', '2016-01-10').tail()
   A
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
```

```python
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
   A
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.

Parameters  periods : int

Number of periods to move, can be positive or negative

freq : DateOffset, timedelta, or time rule string, default None
Increment to use from the tseries module or time rule (e.g. ‘EOM’)

**axis**: int or basestring

Corresponds to the axis that contains the Index

**Returns shifted**: NDFrame

**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: string or pytz.timezone object

axis: the axis to convert

level: int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy: boolean, default True

Also make a copy of the underlying data

**Raises** TypeError

If the axis is tz-naive.

**pandas.DataFrame.tz_localize**

DataFrame.tz_localize (tz, axis=0, level=None, copy=True, ambiguous=’raise’)

Localize tz-naive TimeSeries to target time zone.

**Parameters** tz: string or pytz.timezone object

axis: the axis to localize

level: int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy: boolean, default True

Also make a copy of the underlying data

ambiguous: ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’

- ‘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
infer_dst : boolean, default False

Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order

Raises TypeError

If the TimeSeries is tz-aware and tz is not None.

pandas.DataFrame.unstack

DataFrame.unstack (level=-1, fill_value=None)

Pivot a level of the (necessarily hierarchical) index labels, returning 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). The level involved will automatically get sorted.

Parameters level : int, string, or list of these, default -1 (last level)

Level(s) of index to unstack, can pass level name

fill_value : replace NaN with this value if the unstack produces missing values

Returns unstacked : DataFrame or Series

See also:

DataFrame.pivot Pivot a table based on column values.

DataFrame.stack Pivot a level of the column labels (inverse operation from unstack).

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

>>> s.unstack(level=-1)
a b
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
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

```python
>>> df = s.unstack(level=0)
>>> df.unstack()
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, raise_conflict=False)
Modify DataFrame in place using non-NA values from passed DataFrame. Aligns on indices

**Parameters**
- **other**: DataFrame, or object coercible into a DataFrame
- **join**: {'left'}, default ‘left’
- **overwrite**: boolean, default True
  If True then overwrite values for common keys in the calling frame
- **filter_func**: callable(1d-array) -> 1d-array<boolean>, default None
  Can choose to replace values other than NA. Return True for values that should be updated
- **raise_conflict**: boolean
  If True, will raise an error if the DataFrame and other both contain data in the same place.

**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
```

```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
```

```python
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                     'B': ['x', 'y', 'z']})
```
```python
>>> 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
```

```
>>> df = pd.DataFrame({"A": ["a", "b", "c"],
...                    "B": ["x", "y", "z"]})
>>> new_df = pd.DataFrame({"B": ["d", "e"]}, index=[1, 2])
>>> df.update(new_df)
>>> df
  A  B
0  a  x
1  b  d
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.var**

DataFrame.var (axis=None, skipna=None, 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**: boolean, 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  
  degrees of freedom
- **numeric_only**: boolean, 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**

- **var**: Series or DataFrame (if level specified)
pandas.DataFrame.where

DataFrame.where (cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters cond : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as cond.

other : scalar, NDFrame, or callable

Entries where cond is False are replaced with corresponding value from other. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

errors : str, {'raise', 'ignore'}, default ‘raise’

• raise : allow exceptions to be raised

• ignore : suppress exceptions. On error return original object

Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns wh : same type as caller

See also:
DataFrame.mask()

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
def df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0 -1 0
1 3 -2
2 -5 2
3 -7 3
4 -9 4
```
DataFrame.xs

DataFrame.xs(key, axis=0, level=None, drop_level=True)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

**Parameters**

- **key**: object
  
  Some label contained in the index, or partially in a MultiIndex

- **axis**: int, 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**: boolean, default True
  
  If False, returns object with same levels as self.

**Returns**

xs : Series or DataFrame

**Notes**

xs is only for getting, not setting values.

Multilevel Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see *Multilevel Slicers*

**Examples**

```python
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
   A  B  C
  4  5  2
Name: a
>>> df.xs('C', axis=1)
   A  B  C
a  4  5  2
b  9  7  3
Name: C
```

```
>>> df
   A  B  C  D
first second third
bar  one  1  4  1  8  9
two  1  7  5  5  0
baz  one  1  6  6  8  0
two  2  5  3  5  3
>>> df.xs(('baz', 'three'))
```

### 34.4.2 Attributes and underlying data

**Axes**

- **index**: row labels
- **columns**: column labels

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.as_matrix(columns)</code></td>
<td>Convert the frame to its Numpy-array representation.</td>
</tr>
<tr>
<td><code>DataFrame.dtypes</code></td>
<td>Return the dtypes in this object.</td>
</tr>
<tr>
<td><code>DataFrame.ftypes</code></td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<tr>
<td><code>DataFrame.get_dtypes()</code></td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td><code>DataFrame.get_ftypes()</code></td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td><code>DataFrame.select_dtypes(include, exclude)</code></td>
<td>Return a subset of a DataFrame including/excluding columns based on their dtype.</td>
</tr>
<tr>
<td><code>DataFrame.values</code></td>
<td>Numpy representation of DataFrame</td>
</tr>
<tr>
<td><code>DataFrame.axes</code></td>
<td>Return a list with the row axis labels and column axis labels as the only members.</td>
</tr>
<tr>
<td><code>DataFrame.ndim</code></td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td><code>DataFrame.size</code></td>
<td>Number of elements in the DataFrame</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(index, deep)</code></td>
<td>Memory usage of DataFrame columns.</td>
</tr>
</tbody>
</table>

### 34.4.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.astype(dtype, copy, errors)</code></td>
<td>Cast a pandas object to a specified dtype dtype.</td>
</tr>
<tr>
<td><code>DataFrame.convert_objects(convert_dates, ...)</code></td>
<td>Deprecated.</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([deep])</code></td>
<td>Make a copy of this objects data.</td>
</tr>
<tr>
<td><code>DataFrame.isna()</code></td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td><code>DataFrame.notna()</code></td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
</tbody>
</table>

### 34.4.4 Indexing, iteration
DataFrame.head([n])  Return the first n rows.

DataFrame.at  Fast label-based scalar accessor.

DataFrame.iat  Fast integer location scalar accessor.

DataFrame.loc  Purely label-location based indexer for selection by label.

DataFrame.iloc  Purely integer-location based indexing for selection by position.

DataFrame.insert(loc, column, value[, ...]) Insert column into DataFrame at specified location.

DataFrame.__iter__() Iterate over infor axis.

DataFrame.iteritems() Iterator over (column name, Series) pairs.

DataFrame.iterrows() Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.lookup(row_labels, col_labels) Label-based “fancy indexing” function for DataFrame.

DataFrame.pop(item) Return item and drop from frame.

DataFrame.tail([n]) Return the last n rows.

DataFrame.xs(key[, axis, level, drop_level]) Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

DataFrame.isin(values) Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

DataFrame.where(cond[, other, inplace, ...]) Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

DataFrame.mask(cond[, other, inplace, axis, ...]) Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

DataFrame.query(expr[, inplace]) Query the columns of a frame with a boolean expression.

34.4.4.1 pandas.DataFrame.__iter__

DataFrame.__iter__() Iterate over infor axis

For more information on .at, .iat, .loc, and .iloc, see the indexing documentation.

34.4.5 Binary operator functions

DataFrame.add(other[, axis, level, fill_value]) Addition of dataframe and other, element-wise (binary operator add).

DataFrame.sub(other[, axis, level, fill_value]) Subtraction of dataframe and other, element-wise (binary operator sub).

DataFrame.mul(other[, axis, level, fill_value]) Multiplication of dataframe and other, element-wise (binary operator mul).

DataFrame.div(other[, axis, level, fill_value]) Floating division of dataframe and other, element-wise (binary operator truediv).

DataFrame.truediv(other[, axis, level, ...]) Floating division of dataframe and other, element-wise (binary operator truediv).

DataFrame.floor_divide(other[, axis, level, ...]) Integer division of dataframe and other, element-wise (binary operator floordiv).

DataFrame.mod(other[, axis, level, fill_value]) Modulo of dataframe and other, element-wise (binary operator mod).
Table 34.59 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.pow()</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.radd()</code></td>
<td>Addition of dataframe and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rsub()</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmul()</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rdiv()</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rtruediv()</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rfloordiv()</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmod()</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rpow()</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.lt()</code></td>
<td>Wrapper for flexible comparison methods <code>lt</code>.</td>
</tr>
<tr>
<td><code>DataFrame.gt()</code></td>
<td>Wrapper for flexible comparison methods <code>gt</code>.</td>
</tr>
<tr>
<td><code>DataFrame.le()</code></td>
<td>Wrapper for flexible comparison methods <code>le</code>.</td>
</tr>
<tr>
<td><code>DataFrame.ge()</code></td>
<td>Wrapper for flexible comparison methods <code>ge</code>.</td>
</tr>
<tr>
<td><code>DataFrame.ne()</code></td>
<td>Wrapper for flexible comparison methods <code>ne</code>.</td>
</tr>
<tr>
<td><code>DataFrame.eq()</code></td>
<td>Wrapper for flexible comparison methods <code>eq</code>.</td>
</tr>
<tr>
<td><code>DataFrame.combine()</code></td>
<td>Add two DataFrame objects and do not propagate NaN values, so if for a</td>
</tr>
<tr>
<td><code>DataFrame.combine_first()</code></td>
<td>Combine two DataFrame objects and default to non-null values in frame calling the method.</td>
</tr>
</tbody>
</table>

### 34.4.6 Function application, GroupBy & Window

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.apply()</code></td>
<td>Applies function along input axis of DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.applymap()</code></td>
<td>Apply a function to a DataFrame that is intended to operate elementwise, i.e.</td>
</tr>
<tr>
<td><code>DataFrame.aggregate()</code></td>
<td>Aggregate using callable, string, dict, or list of string/callables</td>
</tr>
<tr>
<td><code>DataFrame.transform()</code></td>
<td>Call function producing a like-indexed NDFrame</td>
</tr>
<tr>
<td><code>DataFrame.groupby()</code></td>
<td>Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.</td>
</tr>
<tr>
<td><code>DataFrame.rolling()</code></td>
<td>Provides rolling window calculations.</td>
</tr>
<tr>
<td><code>DataFrame.expanding()</code></td>
<td>Provides expanding transformations.</td>
</tr>
<tr>
<td><code>DataFrame.ewm()</code></td>
<td>Provides exponential weighted functions</td>
</tr>
</tbody>
</table>

### 34.4.7 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.abs()</code></td>
<td>Return an object with absolute value taken–only applicable to objects that are all numeric.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.61 – continued from previous page

DataFrame.all([axis, bool_only, skipna, level])  Return whether all elements are True over requested axis
DataFrame.any([axis, bool_only, skipna, level])  Return whether any element is True over requested axis
DataFrame.clip([lower, upper, axis, inplace])  Trim values at input threshold(s).
DataFrame.clip_lower(threshold[, axis, inplace])  Return copy of the input with values below given value(s) truncated.
DataFrame.clip_upper(threshold[, axis, inplace])  Return copy of input with values above given value(s) truncated.
DataFrame.corr([method, min_periods])  Compute pairwise correlation of columns, excluding NA/null values
DataFrame.corrwith(other[, axis, drop])  Compute pairwise correlation between rows or columns of two DataFrame objects.
DataFrame.count([axis, level, numeric_only])  Return Series with number of non-NA/null observations over requested axis.
DataFrame.cov([min_periods])  Compute pairwise covariance of columns, excluding NA/null values
DataFrame.cummax([axis, skipna])  Return cumulative max over requested axis.
DataFrame.cummin([axis, skipna])  Return cumulative minimum over requested axis.
DataFrame.cumprod([axis, skipna])  Return cumulative product over requested axis.
DataFrame.cumsum([axis, skipna])  Return cumulative sum over requested axis.
DataFrame.describe([percentiles, include, ...])  Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.
DataFrame.diff([periods, axis])  1st discrete difference of object
DataFrame.eval(expr[, inplace])  Evaluate an expression in the context of the calling DataFrame instance.
DataFrame.kurt([axis, skipna, level, ...])  Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).
DataFrame.mac([axis, skipna, level])  Return the mean absolute deviation of the values for the requested axis
DataFrame.max([axis, skipna, level, ...])  This method returns the maximum of the values in the object.
DataFrame.mean([axis, skipna, level, ...])  Return the mean of the values for the requested axis
DataFrame.median([axis, skipna, level, ...])  Return the median of the values for the requested axis
DataFrame.min([axis, skipna, level, ...])  This method returns the minimum of the values in the object.
DataFrame.mode([axis, numeric_only])  Gets the mode(s) of each element along the axis selected.
DataFrame.pct_change([periods, fill_method, ...])  Percent change over given number of periods.
DataFrame.prod([axis, skipna, level, ...])  Return the product of the values for the requested axis
DataFrame.quantile([q, axis, numeric_only, ...])  Return values at the given quantile over requested axis, a la numpy.percentile.
DataFrame.rank([axis, method, numeric_only, ...])  Compute numerical data ranks (1 through n) along axis.
DataFrame.round([decimals])  Round a DataFrame to a variable number of decimal places.
DataFrame.sem([axis, skipna, level, ddof, ...])  Return unbiased standard error of the mean over requested axis.
DataFrame.skew([axis, skipna, level, ...])  Return unbiased skew over requested axis
DataFrame.sum([axis, skipna, level, ...])  Return the sum of the values for the requested axis
DataFrame.std([axis, skipna, level, ddof, ...])  Return sample standard deviation over requested axis.
DataFrame.var([axis, skipna, level, ddof, ...])  Return unbiased variance over requested axis.

34.4.8 Reindexing / Selection / Label manipulation
DataFrame.add_prefix(prefix) | Concatenate prefix string with panel items names.
---|---
DataFrame.add_suffix(suffix) | Concatenate suffix string with panel items names.
DataFrame.align(other[, join, axis, level, ...]) | Align two objects on their axes with the
DataFrame.drop([labels, axis, index, ...]) | Return new object with labels in requested axis removed.
DataFrame.drop_duplicates([subset, keep, ...]) | Return DataFrame with duplicate rows removed, optionally only
DataFrame.duplicated([subset, keep]) | Return boolean Series denoting duplicate rows, optionally only
DataFrame.equals(other) | Determines if two NDFrame objects contain the same elements.
DataFrame.filter([items, like, regex, axis]) | Subset rows or columns of dataframe according to labels in the specified index.
DataFrame.first(offset) | Convenience method for subsetting initial periods of time series data based on a date offset.
DataFrame.head([n]) | Return the first n rows.
DataFrame.idxmax([axis, skipna]) | Return index of first occurrence of maximum over requested axis.
DataFrame.idxmin([axis, skipna]) | Return index of first occurrence of minimum over requested axis.
DataFrame.last(offset) | Convenience method for subsetting final periods of time series data based on a date offset.
DataFrame.reindex([labels, index, columns, ...]) | Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
DataFrame.reindex_axis(labels[, axis, ...]) | Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
DataFrame.reindex_like(other[, method, ...]) | Return an object with matching indices to myself.
DataFrame.rename([mapper, index, columns, ...]) | Alter axes labels.
DataFrame.rename_axis(mapper[, axis, copy, ...]) | Alter the name of the index or columns.
DataFrame.reset_index([level, drop, ...]) | For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to ‘level_0’, ‘level_1’, etc.
DataFrame.sample([n, frac, replace, ...]) | Returns a random sample of items from an axis of object.
DataFrame.select(crit[, axis]) | Return data corresponding to axis labels matching criteria
DataFrame.set_index(keys[, drop, append, ...]) | Set the DataFrame index (row labels) using one or more existing columns.
DataFrame.tail([n]) | Return the last n rows.
DataFrame.take(indices[, axis, convert, is_copy]) | Return the elements in the given positional indices along an axis.
DataFrame.truncate([before, after, axis, copy]) | Truncates a sorted DataFrame/Series before and/or after some particular index value.

### 34.4.9 Missing data handling

DataFrame.dropna([axis, how, thresh, ...]) | Return object with labels on given axis omitted where alternately any
DataFrame.fillna([value, method, axis, ...]) | Fill NA/NaN values using the specified method
DataFrame.replace([to_replace, value, ...]) | Replace values given in ‘to_replace’ with ‘value’.
### 34.4.10 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.pivot</td>
<td>Reshape data (produce a “pivot” table) based on column values.</td>
</tr>
<tr>
<td>DataFrame.reorder_levels</td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td>DataFrame.sort_values</td>
<td>Sort by the values along either axis.</td>
</tr>
<tr>
<td>DataFrame.sort_index</td>
<td>Sort object by labels (along an axis).</td>
</tr>
<tr>
<td>DataFrame.nlargest</td>
<td>Get the rows of a DataFrame sorted by the n largest values of columns.</td>
</tr>
<tr>
<td>DataFrame.nsmallest</td>
<td>Get the rows of a DataFrame sorted by the n smallest values of columns.</td>
</tr>
<tr>
<td>DataFrame.swaplevel</td>
<td>Swap levels i and j in a MultiIndex on a particular axis.</td>
</tr>
<tr>
<td>DataFrame.stack</td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.</td>
</tr>
<tr>
<td>DataFrame.unstack</td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.</td>
</tr>
<tr>
<td>DataFrame.melt</td>
<td>“Unpivots” a DataFrame from wide format to long format, optionally.</td>
</tr>
<tr>
<td>DataFrame.T</td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td>DataFrame.to_panel</td>
<td>Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.</td>
</tr>
<tr>
<td>DataFrame.to_xarray</td>
<td>Return an xarray object from the pandas object.</td>
</tr>
<tr>
<td>DataFrame.transpose</td>
<td>Transpose index and columns</td>
</tr>
</tbody>
</table>

### 34.4.11 Combining / joining / merging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.append</td>
<td>Append rows of other to the end of this frame, returning a new object.</td>
</tr>
<tr>
<td>DataFrame.assign</td>
<td>Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.</td>
</tr>
<tr>
<td>DataFrame.join</td>
<td>Join columns with other DataFrame either on index or on a key column.</td>
</tr>
<tr>
<td>DataFrame.merge</td>
<td>Merge DataFrame objects by performing a database-style join operation by columns or indexes.</td>
</tr>
<tr>
<td>DataFrame.update</td>
<td>Modify DataFrame in place using non-NA values from passed DataFrame.</td>
</tr>
</tbody>
</table>

### 34.4.12 Time series-related

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.asfreq</td>
<td>Convert TimeSeries to specified frequency.</td>
</tr>
<tr>
<td>DataFrame.asof</td>
<td>The last row without any NaN is taken (or the last row without</td>
</tr>
<tr>
<td>DataFrame.shift</td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
</tbody>
</table>
### 34.4.13 Plotting

DataFrame.plot is both a callable method and a namespace attribute for specific plotting methods of the form DataFrame.plot.<kind>.

#### DataFrame.plot((x, y, kind, ax, ...))
DataFrame plotting accessor and method

<table>
<thead>
<tr>
<th>DataFrame.plot.area(x=None, y=None, **kwds)</th>
<th>Area plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.plot.bar(x=None, y=None, **kwds)</td>
<td>Vertical bar plot</td>
</tr>
<tr>
<td>DataFrame.plot.barh(x=None, y=None, **kwds)</td>
<td>Horizontal bar plot</td>
</tr>
<tr>
<td>DataFrame.plot.box(by=None, **kwds)</td>
<td>Boxplot</td>
</tr>
<tr>
<td>DataFrame.plot.density(**kwds)</td>
<td>Kernel Density Estimate plot</td>
</tr>
<tr>
<td>DataFrame.plot.hexbin(x, y[, C, ...])</td>
<td>Hexbin plot</td>
</tr>
<tr>
<td>DataFrame.plot.hist(by=None, bins=None, **kwds)</td>
<td>Histogram</td>
</tr>
<tr>
<td>DataFrame.plot.kde(**kwds)</td>
<td>Kernel Density Estimate plot</td>
</tr>
<tr>
<td>DataFrame.plot.line(x=None, y=None, **kwds)</td>
<td>Line plot</td>
</tr>
<tr>
<td>DataFrame.plot.pie(y=None, **kwds)</td>
<td>Pie chart</td>
</tr>
<tr>
<td>DataFrame.plot.scatter(x, y[, s, c])</td>
<td>Scatter plot</td>
</tr>
</tbody>
</table>

#### 34.4.13.1 pandas.DataFrame.plot.area

DataFrame.plot.area(x=None, y=None, **kwds)
Area plot

New in version 0.17.0.

**Parameters**
- x, y : label or position, optional
  Coordinates for each point.

**kwds** : optional
Keyword arguments to pass on to pandas.DataFrame.plot().

**Returns**
- axes : matplotlib.AxesSubplot or np.array of them

#### 34.4.13.2 pandas.DataFrame.plot.bar

DataFrame.plot.bar(x=None, y=None, **kwds)
Vertical bar plot
New in version 0.17.0.

Parameters x, y : label or position, optional
  Coordinates for each point.
**kwds : optional
  Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.4.13.3 pandas.DataFrame.plot.barh

DataFrame.plot.barh(x=None, y=None, **kwds)
Horizontal bar plot
New in version 0.17.0.

Parameters x, y : label or position, optional
  Coordinates for each point.
**kwds : optional
  Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.4.13.4 pandas.DataFrame.plot.box

DataFrame.plot.box(by=None, **kwds)
Boxplot
New in version 0.17.0.

Parameters by : string or sequence
  Column in the DataFrame to group by.
**kwds : optional
  Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.4.13.5 pandas.DataFrame.plot.density

DataFrame.plot.density(**kwds)
Kernel Density Estimate plot
New in version 0.17.0.

Parameters **kwds : optional
  Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them
34.13.6 pandas.DataFrame.plot.hexbin

DataFrame.plot.hexbin(x, y, C=None, reduce_C_function=None, gridsize=None, **kwds)

Hexbin plot

New in version 0.17.0.

Parameters x, y : label or position, optional
    Coordinates for each point.

C : label or position, optional
    The value at each (x, y) point.

reduce_C_function : callable, optional
    Function of one argument that reduces all the values in a bin to a single number (e.g. mean, max, sum, std).

gridsize : int, optional
    Number of bins.

**kwds : optional
    Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.13.7 pandas.DataFrame.plot.hist

DataFrame.plot.hist(by=None, bins=10, **kwds)

Histogram

New in version 0.17.0.

Parameters by : string or sequence
    Column in the DataFrame to group by.

bins: integer, default 10
    Number of histogram bins to be used

**kwds : optional
    Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.13.8 pandas.DataFrame.plot.kde

DataFrame.plot.kde(**kwds)

Kernel Density Estimate plot

New in version 0.17.0.

Parameters **kwds : optional
    Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them
34.4.13.9 pandas.DataFrame.plot.line

DataFrame.plot.line(x=None, y=None, **kwds)

Line plot

New in version 0.17.0.

Parameters x, y : label or position, optional

Coordinates for each point.

**kwds : optional

Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.4.13.10 pandas.DataFrame.plot.pie

DataFrame.plot.pie(y=None, **kwds)

Pie chart

New in version 0.17.0.

Parameters y : label or position, optional

Column to plot.

**kwds : optional

Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

34.4.13.11 pandas.DataFrame.plot.scatter

DataFrame.plot.scatter(x, y, s=None, c=None, **kwds)

Scatter plot

New in version 0.17.0.

Parameters x, y : label or position, optional

Coordinates for each point.

s : scalar or array_like, optional

Size of each point.

c : label or position, optional

Color of each point.

**kwds : optional

Keyword arguments to pass on to pandas.DataFrame.plot().

Returns axes : matplotlib.AxesSubplot or np.array of them

Dataframe.boxplot([column, by, ax, ...])

Make a box plot from DataFrame column optionally grouped by some columns or

Dataframe.hist(data[, column, by, grid, ...])

Draw histogram of the DataFrame’s series using matplotlib / pylab.
34.4.14 Serialization / IO / Conversion

**DataFrame.from_csv**(path[, header, sep, ...])  
Read CSV file (DEPRECATED, please use pandas.read_csv() instead).

**DataFrame.from_dict**(data[, orient, dtype])  
Construct DataFrame from dict of array-like or dicts

**DataFrame.from_items**(items[, columns, orient])  
Convert (key, value) pairs to DataFrame.

**DataFrame.from_records**(data[, index, ...])  
Convert structured or record ndarray to DataFrame

**DataFrame.info**(verbose, buf, max_cols, ...)  
Concise summary of a DataFrame.

**DataFrame.to_pickle**(path[, compression, ...])  
Pickle (serialize) object to input file path.

**DataFrame.to_csv**(path_or_buf[, sep, na_rep, ...])  
Write DataFrame to a comma-separated values (csv) file

**DataFrame.to_hdf**(path_or_buf, key, **kwargs)  
Write the contained data to an HDF5 file using HDFStore.

**DataFrame.to_sql**(name, con[, flavor, ...])  
Write records stored in a DataFrame to a SQL database.

**DataFrame.to_dict**(orient, into)  
Convert DataFrame to dictionary.

**DataFrame.to_excel**(excel_writer[, ...])  
Write DataFrame to an excel sheet

**DataFrame.to_json**(path_or_buf[, orient, ...])  
Convert the object to a JSON string.

**DataFrame.to_html**(buf, columns, col_space, ...)  
Render a DataFrame as an HTML table.

**DataFrame.to_feather**(fname)  
Write out the binary feather-format for DataFrames

**DataFrame.to_stata**(fname[, convert_dates, ...])  
A class for writing Stata binary dta files from array-like objects

**DataFrame.to_msgpack**(path_or_buf, encoding)  
msgpack (serialize) object to input file path

**DataFrame.to_gbq**(destination_table, project_id)  
Write a DataFrame to a Google BigQuery table.

**DataFrame.to_records**(index, convert_datetime64)  
Convert DataFrame to record array.

**DataFrame.to_sparse**(fill_value, kind)  
Convert to SparseDataFrame

**DataFrame.to_dense()**  
Return dense representation of NDFrame (as opposed to sparse)

**DataFrame.to_string**(buf, columns, ...)  
Render a DataFrame to a console-friendly tabular output.

**DataFrame.to_clipboard**(excel, sep)  
Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

34.4.15 Sparse

**SparseDataFrame.to_coo()**  
Return the contents of the frame as a sparse SciPy COO matrix.

34.4.15.1 pandas.SparseDataFrame.to_coo

SparseDataFrame.to_coo()  
Return the contents of the frame as a sparse SciPy COO matrix.

New in version 0.20.0.

**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.

34.5 Panel

34.5.1 Constructor

```
Panel([data, items, major_axis, minor_axis, ...]) Represents wide format panel data, stored as 3-dimensional array
```

34.5.1.1 pandas.Panel

```
class pandas.Panel (data=None, items=None, major_axis=None, minor_axis=None, copy=False, dtype=None)
```

Represents wide format panel data, stored as 3-dimensional array

**Parameters**

- `data` : ndarray (items x major x minor), or dict of DataFrames
- `items` : Index or array-like axis=0
- `major_axis` : Index or array-like axis=1
- `minor_axis` : Index or array-like axis=2
- `dtype` : dtype, default None Data type to force, otherwise infer
- `copy` : boolean, default False Copy data from inputs. Only affects DataFrame / 2d ndarray input

**Attributes**

- `at` : Fast label-based scalar accessor
- `axes` : Return index label(s) of the internal NDFrame
- `blocks` : Internal property, property synonym for as_blocks()
- `dtypes` : Return the dtypes in this object.
- `empty` : True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.
- `ftypes` : Return the ftypes (indication of sparse/dense and dtype) in this object.
Table 34.73 – continued from previous page

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iat</td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td>iloc</td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td>is_copy</td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td>ix</td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td>loc</td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td>shape</td>
<td>Return a tuple of axis dimensions</td>
</tr>
<tr>
<td>size</td>
<td>number of elements in the NDFrame</td>
</tr>
<tr>
<td>values</td>
<td>Numpy representation of NDFrame</td>
</tr>
</tbody>
</table>

**pandas.Panel.at**

Panel.at

Fast label-based scalar accessor

Similarly to loc, at provides label based scalar lookups. You can also set using these indexers.

**pandas.Panel.axes**

Panel.axes

Return index label(s) of the internal NDFrame

**pandas.Panel.blocks**

Panel.blocks

Internal property, property synonym for as_blocks()

Depreciated since version 0.21.0.

**pandas.Panel.dtypes**

Panel.dtypes

Return the dtypes in this object.

**pandas.Panel.empty**

Panel.empty

True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.

See also:

pandas.Series.dropna, pandas.DataFrame.dropna

**Notes**

If NDFrame 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
```

**pandas.Panel.ftypes**

Panel.ftypes

Return the ftypes (indication of sparse/dense and dtype) in this object.

**pandas.Panel.iat**

Panel.iat

Fast integer location scalar accessor.

Similarly to iloc, iat provides integer based lookups. You can also set using these indexers.

**pandas.Paneliloc**

Panel.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, DataFrame or Panel) and that returns valid output for indexing (one of the above)
.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

pandas.Panel.is_copy

Panel.is_copy = None

pandas.Panel.ix

Panel.ix
A primarily label-location based indexer, with integer position fallback.

.ix[] supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

.ix is the most general indexer and will support any of the inputs in .loc and .iloc. ix also supports floating point label schemes. ix is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it’s usually better to be explicit and use .iloc or .loc.

See more at Advanced Indexing.

pandas.Panel.loc

Panel.loc
Purely label-location based indexer for selection by label.

.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'].

• A slice object with labels, e.g. 'a':'f' (note that contrary to usual python slices, both the start and the stop are included!).

• A boolean array.

• A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

.loc will raise a KeyError when the items are not found.

See more at Selection by Label

pandas.Panel.ndim

Panel.ndim
Number of axes / array dimensions
pandas.Panel.shape

Panel.shape
Return a tuple of axis dimensions

pandas.Panel.size

Panel.size
number of elements in the NDFrame

pandas.Panel.values

Panel.values
Numpy representation of NDFrame

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.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Return an object with absolute value taken–only applicable to objects that are all numeric.</td>
</tr>
<tr>
<td>add(other[, axis])</td>
<td>Addition of series and other, element-wise (binary operator add).</td>
</tr>
<tr>
<td>add_prefix(prefix)</td>
<td>Concatenate prefix string with panel items names.</td>
</tr>
<tr>
<td>add_suffix(suffix)</td>
<td>Concatenate suffix string with panel items names.</td>
</tr>
<tr>
<td>agg(func, *args, **kwargs)</td>
<td>aggregate(func, *args, **kwargs)</td>
</tr>
<tr>
<td>align(other, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>all([axis, bool_only, skipna, level])</td>
<td>Return whether all elements are True over requested axis</td>
</tr>
<tr>
<td>any([axis, bool_only, skipna, level])</td>
<td>Return whether any element is True over requested axis</td>
</tr>
<tr>
<td>apply(func[, axis])</td>
<td>Applies function along axis (or axes) of the Panel</td>
</tr>
<tr>
<td>as_blocks([copy])</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has a homogeneous dtype.</td>
</tr>
<tr>
<td>as_matrix()</td>
<td></td>
</tr>
<tr>
<td>asfreq(freq[, method, how, normalize, ...])</td>
<td>Convert TimeSeries to specified frequency.</td>
</tr>
<tr>
<td>asof(where[, subset])</td>
<td>The last row without any NaN is taken (or the last row without</td>
</tr>
<tr>
<td>astype(dtype[, copy, errors])</td>
<td>Cast a pandas object to a specified dtype dtype.</td>
</tr>
<tr>
<td>at_time(time[, asof])</td>
<td>Select values at particular time of day (e.g.</td>
</tr>
</tbody>
</table>

Continued on next page
# Table 34.74 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>between_time</code></td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
<td><code>(start_time, end_time[, ...])</code></td>
</tr>
<tr>
<td><code>bfill</code></td>
<td>Synonym for <code>DataFrame.fillna(method='bfill')</code>.</td>
<td><code>([axis, inplace, limit, downcast()])</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td>Return the bool of a single element PandasObject.</td>
<td></td>
</tr>
<tr>
<td><code>clip</code></td>
<td>Trim values at input threshold(s).</td>
<td><code>([lower, upper, axis, inplace])</code></td>
</tr>
<tr>
<td><code>clip_lower</code></td>
<td>Return copy of the input with values below given value(s) truncated.</td>
<td><code>(threshold[, axis, inplace])</code></td>
</tr>
<tr>
<td><code>clip_upper</code></td>
<td>Return copy of input with values above given value(s) truncated.</td>
<td><code>(threshold[, axis, inplace])</code></td>
</tr>
<tr>
<td><code>compound</code></td>
<td>Return the compound percentage of the values for the requested axis</td>
<td><code>([axis, skipna, level])</code></td>
</tr>
<tr>
<td><code>conform</code></td>
<td>Conform input DataFrame to align with chosen axis pair.</td>
<td><code>([frame[, axis]])</code></td>
</tr>
<tr>
<td><code>consolidate</code></td>
<td>DEPRECATED: consolidate will be an internal implementation only.</td>
<td></td>
</tr>
<tr>
<td><code>convert_objects</code></td>
<td>Deprecated.</td>
<td><code>([convert_dates, ...])</code></td>
</tr>
<tr>
<td><code>copy</code></td>
<td>Make a copy of this objects data.</td>
<td><code>([deep])</code></td>
</tr>
<tr>
<td><code>count</code></td>
<td>Return number of observations over requested axis.</td>
<td><code>([axis])</code></td>
</tr>
<tr>
<td><code>cummax</code></td>
<td>Return cumulative max over requested axis.</td>
<td><code>([axis, skipna])</code></td>
</tr>
<tr>
<td><code>cummin</code></td>
<td>Return cumulative minimum over requested axis.</td>
<td><code>([axis, skipna])</code></td>
</tr>
<tr>
<td><code>cumprod</code></td>
<td>Return cumulative product over requested axis.</td>
<td><code>([axis, skipna])</code></td>
</tr>
<tr>
<td><code>cumsom</code></td>
<td>Return cumulative sum over requested axis.</td>
<td><code>([axis, skipna])</code></td>
</tr>
<tr>
<td><code>describe</code></td>
<td>Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.</td>
<td><code>([percentiles, include, exclude])</code></td>
</tr>
<tr>
<td><code>div</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
<td><code>([other[, axis])</code></td>
</tr>
<tr>
<td><code>divide</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
<td><code>([other[, axis])</code></td>
</tr>
<tr>
<td><code>drop</code></td>
<td>Return new object with labels in requested axis removed.</td>
<td><code>([labels, axis, index, columns, level, ...])</code></td>
</tr>
<tr>
<td><code>dropna</code></td>
<td>Drop 2D from panel, holding passed axis constant</td>
<td><code>([axis, how, inplace])</code></td>
</tr>
<tr>
<td><code>eq</code></td>
<td>Wrapper for comparison method <code>eq</code>.</td>
<td><code>([other[, axis])</code></td>
</tr>
<tr>
<td><code>equals</code></td>
<td>Determines if two NDFrame objects contain the same elements.</td>
<td><code>([other])</code></td>
</tr>
<tr>
<td><code>ffill</code></td>
<td>Synonym for <code>DataFrame.fillna(method='ffill')</code>.</td>
<td><code>([axis, inplace, limit, downcast()])</code></td>
</tr>
<tr>
<td><code>fillna</code></td>
<td>Fill NA/NaN values using the specified method.</td>
<td><code>([value, method, axis, inplace, ...])</code></td>
</tr>
<tr>
<td><code>filter</code></td>
<td>Subset rows or columns of dataframe according to labels in the specified index.</td>
<td><code>([items, like, regex, axis])</code></td>
</tr>
<tr>
<td><code>first</code></td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
<td><code>(</code>offset<code>)</code></td>
</tr>
<tr>
<td><code>floordiv</code></td>
<td>Integer division of series and other, element-wise (binary operator <code>floordiv</code>).</td>
<td><code>([other[, axis])</code></td>
</tr>
<tr>
<td><code>fromDict</code></td>
<td>Construct Panel from dict of DataFrame objects.</td>
<td><code>([data[, intersect, orient, dtype]])</code></td>
</tr>
<tr>
<td><code>from_dict</code></td>
<td>Construct Panel from dict of DataFrame objects.</td>
<td><code>([data[, intersect, orient, dtype]])</code></td>
</tr>
<tr>
<td><code>ge</code></td>
<td>Wrapper for comparison method <code>ge</code>.</td>
<td><code>([other[, axis])</code></td>
</tr>
<tr>
<td><code>get</code></td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.).</td>
<td><code>([key[, default])</code></td>
</tr>
<tr>
<td><code>get_dtype_counts</code></td>
<td>Return the counts of dtypes in this object.</td>
<td></td>
</tr>
</tbody>
</table>
Table 34.74 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_ftype_counts()</code></td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td><code>get_value(*args,**kwargs)</code></td>
<td>Quickly retrieve single value at (item, major, minor) location</td>
</tr>
<tr>
<td><code>get_values()</code></td>
<td>same as values (but handles sparseness conversions)</td>
</tr>
<tr>
<td><code>groupby(function[, axis])</code></td>
<td>Group data on given axis, returning GroupBy object</td>
</tr>
<tr>
<td><code>gt(other[, axis])</code></td>
<td>Wrapper for comparison method gt</td>
</tr>
<tr>
<td><code>head([n])</code></td>
<td></td>
</tr>
<tr>
<td><code>infer_objects()</code></td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td><code>interpolate([method,axis,limit,inplace,...])</code></td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td><code>isna()</code></td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td><code>isnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are NA.</td>
</tr>
<tr>
<td><code>iteritems()</code></td>
<td>Iterate over (label, values) on info axis</td>
</tr>
<tr>
<td><code>join(other[,how,lsuffix,rsuffix])</code></td>
<td>Join items with other Panel either on major and minor axes column</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Get the ‘info axis’ (see Indexing for more)</td>
</tr>
<tr>
<td><code>kurt([axis,skipna,level,numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td><code>kurtosis([axis,skipna,level,numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).</td>
</tr>
<tr>
<td><code>last(offset)</code></td>
<td>Convenience method for subsetting final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>le(other[,axis])</code></td>
<td>Wrapper for comparison method le</td>
</tr>
<tr>
<td><code>lt(other[,axis])</code></td>
<td>Wrapper for comparison method lt</td>
</tr>
<tr>
<td><code>mad([axis,skipna,level])</code></td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td><code>major_xs(key)</code></td>
<td>Return slice of panel along major axis</td>
</tr>
<tr>
<td><code>mask(cond[,other,inplace,axis,level,...])</code></td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.</td>
</tr>
<tr>
<td><code>max([axis,skipna,level,numeric_only])</code></td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td><code>mean([axis,skipna,level,numeric_only])</code></td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td><code>median([axis,skipna,level,numeric_only])</code></td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td><code>min([axis,skipna,level,numeric_only])</code></td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><code>minor_xs(key)</code></td>
<td>Return slice of panel along minor axis</td>
</tr>
<tr>
<td><code>mod(other[,axis])</code></td>
<td>Modulo of series and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td><code>mul(other[,axis])</code></td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>multiply(other[,axis])</code></td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>ne(other[,axis])</code></td>
<td>Wrapper for comparison method ne</td>
</tr>
<tr>
<td><code>notna()</code></td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are not NA.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.74 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pct_change([periods, fill_method, limit, freq])</code></td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td><code>pipe(func, *args, **kwargs)</code></td>
<td>Apply <code>func(self, *args, **kwargs)</code></td>
</tr>
<tr>
<td><code>pop(item)</code></td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td><code>pow(other[, axis])</code></td>
<td>Exponential power of series 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 for the requested axis</td>
</tr>
<tr>
<td><code>product([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>radd(other[, axis])</code></td>
<td>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>rdiv(other[, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>reindex(*args, **kwargs)</code></td>
<td>Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><code>reindex_axis(labels[, axis, method, level, ...])</code></td>
<td>Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit, ...])</code></td>
<td>Return an object with matching indices to myself.</td>
</tr>
<tr>
<td><code>rename([items, major_axis, minor_axis])</code></td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><code>rename_axis(mapper[, axis, copy, inplace])</code></td>
<td>Alter the name of the index or columns.</td>
</tr>
<tr>
<td><code>replace([to_replace, value, inplace, limit, ...])</code></td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td><code>resample(rule[, how, axis, fill_method, ...])</code></td>
<td>Convenience method for frequency conversion and re-sampling of time series.</td>
</tr>
<tr>
<td><code>rfloordiv(other[, axis])</code></td>
<td>Integer division of series and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>rmod(other[, axis])</code></td>
<td>Modulo of series and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>rmul(other[, axis])</code></td>
<td>Multiplication of series and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>round([decimals])</code></td>
<td>Round each value in Panel to a specified number of decimal places.</td>
</tr>
<tr>
<td><code>rpow(other[, axis])</code></td>
<td>Exponential power of series and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>rsub(other[, axis])</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>rtruediv(other[, axis])</code></td>
<td>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>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><code>select(crit[, axis])</code></td>
<td>Return data corresponding to axis labels matching criteria</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_value(*args, **kwargs)</code></td>
<td>Quickly set single value at (item, major, minor) location</td>
</tr>
<tr>
<td><code>shift([periods, freq, axis])</code></td>
<td>Shift index by desired number of periods with an optional time freq.</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>Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>sort_index([axis, level, ascending, ...])</code></td>
<td>Sort object by labels (along an axis)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.74 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sort_values(by, axis, ascending, inplace, ...)</code></td>
<td>NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.</td>
</tr>
<tr>
<td><code>squeeze(axis)</code></td>
<td>Squeeze length 1 dimensions.</td>
</tr>
<tr>
<td><code>std(axis, skipna, level, ddof, numeric_only)</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>sub(other[, axis])</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>subtract(other[, axis])</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>sum(axis, skipna, level, numeric_only)</code></td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2[, copy])</code></td>
<td>Interchange axes and swap values axes appropriately.</td>
</tr>
<tr>
<td><code>swaplevel([i, j, axis])</code></td>
<td>Swap levels i and j in a MultiIndex on a particular axis.</td>
</tr>
<tr>
<td><code>tail([n])</code></td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><code>toLong(*args, **kwargs)</code></td>
<td>Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.</td>
</tr>
<tr>
<td><code>to_clipboard([excel, sep])</code></td>
<td>Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.</td>
</tr>
<tr>
<td><code>to_dense()</code></td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td><code>to_excel(path[, na_rep, engine])</code></td>
<td>Write each DataFrame in Panel to a separate excel sheet</td>
</tr>
<tr>
<td><code>to_frame([filter_observations])</code></td>
<td>Transform wide format into long (stacked) format as DataFrame whose columns are the Panel’s items and whose index is a MultiIndex formed of the Panel’s major and minor axes.</td>
</tr>
<tr>
<td><code>to_hdf(path_or_buf, key, **kwargs)</code></td>
<td>Write the contained data to an HDF5 file using HDFStore.</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 an object to a tabular environment table.</td>
</tr>
<tr>
<td><code>to_long(*args, **kwargs)</code></td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_msgpack([path_or_buf, encoding])</code></td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_pickle(path[, compression, protocol])</code></td>
<td>Pickle (serialize) object to input file path.</td>
</tr>
<tr>
<td><code>to_sparse(*args, **kwargs)</code></td>
<td>NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.</td>
</tr>
<tr>
<td><code>to_sql(name, con[, flavor, schema, ...])</code></td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td><code>to_xarray()</code></td>
<td>Return an xarray object from the pandas object.</td>
</tr>
<tr>
<td><code>transpose(*args, **kwargs)</code></td>
<td>Permute the dimensions of the Panel</td>
</tr>
<tr>
<td><code>truediv(other[, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>truncate([before, after, axis, copy])</code></td>
<td>Truncates a sorted DataFrame/Series before and/or after some particular index value.</td>
</tr>
<tr>
<td><code>tshift([periods, freq, axis])</code></td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td><code>tz_convert(tz[, axis, level, copy])</code></td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
<tr>
<td><code>tz_localize(tz[, axis, level, copy, ambiguous])</code></td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
<tr>
<td><code>update(other[, join, overwrite, ...])</code></td>
<td>Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.</td>
</tr>
<tr>
<td><code>var(axis, skipna, level, ddof, numeric_only)</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.74 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>where(cond[, other, inplace, axis, level, ...])</td>
<td>Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.</td>
</tr>
<tr>
<td>xs(key[, axis])</td>
<td>Return slice of panel along selected axis</td>
</tr>
</tbody>
</table>

**pandas.Panel.abs**

Panel.abs()  
Return an object with absolute value taken—only applicable to objects that are all numeric.  

Returns abs: type of caller

**pandas.Panel.add**

Panel.add(other, axis=0)  
Addition of series and other, element-wise (binary operator add). Equivalent to panel + other.  

Parameters other : DataFrame or Panel  
axis : {items, major_axis, minor_axis}  
Axis to broadcast over  

Returns Panel  
See also:  
Panel.radd

**pandas.Panel.add_prefix**

Panel.add_prefix(prefix)  
Concatenate prefix string with panel items names.  

Parameters prefix : string  

Returns with_prefix : type of caller

**pandas.Panel.add_suffix**

Panel.add_suffix(suffix)  
Concatenate suffix string with panel items names.  

Parameters suffix : string  

Returns with_suffix : type of caller

**pandas.Panel.agg**

Panel.agg(func, *args, **kwargs)
**pandas.Panel.aggregate**

Panel.aggregate(func, *args, **kwargs)

**pandas.Panel.align**

Panel.align(other, **kwargs)

**pandas.Panel.all**

Panel.all(axis=None, bool_only=None, skipna=None, level=None, **kwargs)

Return whether all elements are True over requested axis

**Parameters**

- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True
- **level**: int or level name, default None
- **bool_only**: boolean, default None

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns**

- **all**: DataFrame or Panel (if level specified)

**pandas.Panel.any**

Panel.any(axis=None, bool_only=None, skipna=None, level=None, **kwargs)

Return whether any element is True over requested axis

**Parameters**

- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True
- **level**: int or level name, default None
- **bool_only**: boolean, default None

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns**

- **any**: DataFrame or Panel (if level specified)
pandas.Panel.apply

Panel.apply(func, axis='major', **kwargs)
Applies function along axis (or axes) of the Panel

Parameters func : function
   Function to apply to each combination of ‘other’ axes e.g. if axis = ‘items’, the
   combination of major_axis/minor_axis will each be passed as a Series; if axis =
   (‘items’, ‘major’), DataFrames of items & major axis will be passed

axis : {‘items’, ‘minor’, ‘major’}, or {0, 1, 2}, or a tuple with two
   axes

 Additional keyword arguments will be passed as keywords to the function

Returns result : Panel, DataFrame, or Series

Examples

Returns a Panel with the square root of each element

>>> p = pd.Panel(np.random.rand(4,3,2))
>>> p.apply(np.sqrt)

Equivalent to p.sum(1), returning a DataFrame

>>> p.apply(lambda x: x.sum(), axis=1)

Equivalent to previous:

>>> p.apply(lambda x: x.sum(), axis='minor')

Return the shapes of each DataFrame over axis 2 (i.e the shapes of items x major), as a Series

>>> p.apply(lambda x: x.shape, axis=(0,1))

pandas.Panel.as_blocks

Panel.as_blocks(copy=True)
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters copy : boolean, default True

Returns values : a dict of dtype -> Constructor Types

pandas.Panel.as_matrix

Panel.as_matrix()
pandas.Panel.asfreq

Panel.asfreq(freq=None, method=None, how=None, normalize=False, fill_value=None)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. resample is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters freq : DateOffset object, or string
method : {'backfill'/'bfill', 'pad'/'ffill'}, 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).

New in version 0.20.0.

Returns converted : type of caller

See also:
reindex

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.
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.Panel.asof**

```
pandas.Panel.asof
```

```
Panel.asof(where, subset=None)
```

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame).

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

**Parameters**

- `where` : date or array of dates
  - `subset` : string or list of strings, default None
    - if not None use these columns for NaN propagation

**Returns**

- where is scalar
  - value or NaN if input is Series
  - Series if input is DataFrame

- where is Index: same shape object as input

**See also:**
merge_asof

Notes

Dates are assumed to be sorted. Raises if this is not the case.

pandas.Panel.astype

Panel.astype(dtype, copy=True, errors='raise', **kwargs)

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).

dtype : {'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

New in version 0.20.0.

raise_on_error : raise on invalid input

Deprecated since version 0.20.0: Use errors instead

**kwargs : keyword arguments to pass on to the constructor

Returns
casted : type of caller

See also:

pandas.to_datetime Convert argument to datetime.
pandas.to_timedelta Convert argument to timedelta.
pandas.to_numeric Convert argument to a numeric type.
numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

```python
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0   1
1   2
dtype: int32
>>> ser.astype('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
>>> ser.astype('category', ordered=True, categories=[2, 1])
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('int', copy=False)
>>> s2[0] = 10
>>> s1
0 10
1 2
dtype: int64
```

### pandas.Panel.at_time

**Panel.at_time**(time, asof=False)
Select values at particular time of day (e.g. 9:30AM).

- **Parameters**
  - `time`: datetime.time or string
- **Returns**
  - `values_at_time`: type of caller

### pandas.Panel.between_time

**Panel.between_time**(start_time, end_time, include_start=True, include_end=True)
Select values between particular times of the day (e.g., 9:00-9:30 AM).

- **Parameters**
  - `start_time`: datetime.time or string
  - `end_time`: datetime.time or string
  - `include_start`: boolean, default True
  - `include_end`: boolean, default True
- **Returns**
  - `values_between_time`: type of caller
pandas.Panel.bfill

Panel.bfill \( (axis=None, inplace=False, limit=None, downcast=None) \)
Synonym for DataFrame.fillna(method='bfill')

pandas.Panel.bool

Panel.bool()
Return the bool of a single element PandasObject.
This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not
have exactly 1 element, or that element is not boolean

pandas.Panel.clip

Panel.clip \( (lower=None, upper=None, axis=None, inplace=False, *args, **kwargs) \)
Trim values at input threshold(s).

Parameters lower : float or array_like, default None
upper : float or array_like, default None
axis : int or string axis name, optional
Align object with lower and upper along the given axis.
inplace : boolean, default False
Whether to perform the operation in place on the data New in version 0.21.0.

Returns clipped : Series

Examples

```python
>>> df
   0   1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967

>>> df.clip(-1.0, 0.5)
   0   1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000

>>> t
   0   1   2
0 -0.3 -0.2 -0.1
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

```python
3  0.0
4  0.1
dtype: float64

>>> df.clip(t, t + 1, axis=0)
   0  1
0 0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4 1.100000  0.570967
```

**pandas.Panel.clip_lower**

**Panel.clip_lower**(threshold, axis=None, inplace=False)  
Return copy of the input with values below given value(s) truncated.

- **Parameters**
  - **threshold**: float or array_like
  - **axis**: int or string axis name, optional
    - Align object with threshold along the given axis.
  - **inplace**: boolean, default False
    - Whether to perform the operation in place on the data. New in version 0.21.0.

- **Returns**
  - clipped: same type as input

**See also:**

- clip

**pandas.Panel.clip_upper**

**Panel.clip_upper**(threshold, axis=None, inplace=False)  
Return copy of input with values above given value(s) truncated.

- **Parameters**
  - **threshold**: float or array_like
  - **axis**: int or string axis name, optional
    - Align object with threshold along the given axis.
  - **inplace**: boolean, default False
    - Whether to perform the operation in place on the data. New in version 0.21.0.

- **Returns**
  - clipped: same type as input

**See also:**

- clip
pandas.Panel.compound

Panel.compound(axis=None, skipna=None, level=None)
Return the compound percentage of the values for the requested axis

Parameters axis : {items (0), major_axis (1), minor_axis (2)}
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
numeric_only : boolean, 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 compounded : DataFrame or Panel (if level specified)

pandas.Panel.conform

Panel.conform(frame, axis='items')
Conform input DataFrame to align with chosen axis pair.

Parameters frame : DataFrame
axis : {'items', 'major', 'minor'}
Axis the input corresponds to. E.g., if axis='major', then the frame’s columns would be items, and the index would be values of the minor axis

Returns DataFrame

pandas.Panel.consolidate

Panel.consolidate(inplace=False)
DEPRECATED: consolidate will be an internal implementation only.

pandas.Panel.convert_objects

Panel.convert_objects(convert_dates=True, convert_numeric=False, convert_time deltas=True, copy=True)
Deprecated. Attempt to infer better dtype for object columns

Parameters convert_dates : boolean, default True
If True, convert to date where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.
convert_numeric : boolean, default False
If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.
convert_timedeltas: boolean, default True
If True, convert to timedelta where possible. If ‘coerce’, force conversion, with
unconvertible values becoming NaT.

copy: boolean, default True
If True, return a copy even if no copy is necessary (e.g. no conversion was done).
Note: This is meant for internal use, and should not be confused with inplace.

Returns converted: same as input object

See also:
pandas.to_datetime Convert argument to datetime.
pandas.to_timedelta Convert argument to timedelta.
pandas.to_numeric Return a fixed frequency timedelta index, with day as the default.

pandas.Panel.copy

Panel.copy (deep=True)
Make a copy of this objects data.

Parameters deep: boolean or string, default True
Make a deep copy, including a copy of the data and the indices. With
deep=False neither the indices or the data are copied.

Note that when deep=True data is copied, actual python objects will not be
copied recursively, only the reference to the object. This is in contrast to copy.
decopy in the Standard Library, which recursively copies object data.

Returns copy: type of caller

pandas.Panel.count

Panel.count (axis='major')
Return number of observations over requested axis.

Parameters axis: {'items', 'major', 'minor'} or {0, 1, 2}

Returns count: DataFrame

pandas.Panel.cummax

Panel.cummax (axis=None, skipna=True, *args, **kwargs)
Return cumulative max over requested axis.

Parameters axis: {items (0), major_axis (1), minor_axis (2)}
skipna: boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns cummax: DataFrame

See also:
pandas: powerful Python data analysis toolkit, Release 0.21.0

**pandas.core.window.Expanding.max** Similar functionality but ignores NaN values.

**pandas.Panel.cummin**

Panel.cummin(axis=None, skipna=True, *args, **kwargs)
Return cumulative minimum over requested axis.

Parameters:
- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns: cummin : DataFrame

See also:

**pandas.core.window.Expanding.min** Similar functionality but ignores NaN values.

**pandas.Panel.cumprod**

Panel.cumprod(axis=None, skipna=True, *args, **kwargs)
Return cumulative product over requested axis.

Parameters:
- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns: cumprod : DataFrame

See also:

**pandas.core.window.Expanding.prod** Similar functionality but ignores NaN values.

**pandas.Panel.cumsum**

Panel.cumsum(axis=None, skipna=True, *args, **kwargs)
Return cumulative sum over requested axis.

Parameters:
- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns: cumsum : DataFrame

See also:

**pandas.core.window.Expanding.sum** Similar functionality but ignores NaN values.
pandas.Panel.describe

Panel.describe(percentiles=None, include=None, exclude=None)
Generates descriptive statistics 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(include=['O'])). To exclude pandas categorical columns, use 'category'

• None (default) : The result will exclude nothing.

Returns

summary: Series/DataFrame of summary statistics

See also:

DataFrame.count, DataFrame.max, DataFrame.min, DataFrame.mean, DataFrame.std, DataFrame.select_dtypes

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
```

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()
count 3
unique 2
top 2010-01-01 00:00:00
freq 2
first 2000-01-01 00:00:00
last 2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({ 'object': [ 'a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical([ 'd', 'e', 'f'])
...                     })
>>> df.describe()
          numeric
```
Describing all columns of a DataFrame regardless of data type.

```python
>>> df.describe(include='all')
categorical     numeric     object
count            3.0          NaN          3
unique           3 NaN         3
top f            NaN c
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=[np.object])
object
count 3
unique 3
```
Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
           categorical
       count   3
       unique  3
         top   f
        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    c
        freq    1    1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.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.Panel.div**

Panel.div(other, axis=0)

Floating division of series and other, element-wise (binary operator truediv). Equivalent to panel / other.

Parameters

other : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns

Panel

See also:

Panel.rtruediv
**pandas.Panel.divide**

`Panel.divide(other, axis=0)`

Floating division of series and other, element-wise (binary operator `truediv`). Equivalent to `panel / other`.

**Parameters**

- **other**: DataFrame or Panel
- **axis**: `{items, major_axis, minor_axis}`
  
  Axis to broadcast over

**Returns**

Panel

**See also:**

`Panel.rtruediv`

**pandas.Panel.drop**

`Panel.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')`

Return new object with labels in requested axis removed.

**Parameters**

- **labels**: single label or list-like
  
  Index or column labels to drop.
- **axis**: int or axis name
  
  Whether to drop labels from the index (0 / 'index') or columns (1 / 'columns').
- **index, columns**: single label or list-like
  
  Alternative to specifying `axis` (`labels, axis=1` is equivalent to `columns=labels`).

  New in version 0.21.0.
- **level**: int or level name, default None
  
  For MultiIndex
- **inplace**: bool, default False
  
  If True, do operation inplace and return None.
- **errors**: {'ignore', 'raise'}, default 'raise'
  
  If ‘ignore’, suppress error and existing labels are dropped.

**Returns**

dropped : type of caller

**Notes**

Specifying both `labels` and `index or columns` will raise a ValueError.
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
```

**pandas.Panel.dropna**

Panel.dropna (axis=0, how='any', inplace=False)
Drop 2D from panel, holding passed axis constant

- **Parameters**
  - **axis** : int, default 0
    - Axis to hold constant. E.g. axis=1 will drop major_axis entries having a certain amount of NA data
  - **how** : {'all', 'any'}, default ‘any’
    - ‘any’: one or more values are NA in the DataFrame along the axis. For ‘all’ they all must be.
  - **inplace** : bool, default False
    - If True, do operation inplace and return None.

- **Returns**
  - **dropped** : Panel

**pandas.Panel.eq**

Panel.eq (other, axis=None)
Wrapper for comparison method eq
pandas.Panel.equals

Panel.equals(other)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

pandas.Panel.ffill

Panel.ffill(axis=None, inplace=False, limit=None, downcast=None)
Synonym for DataFrame.fillna(method='ffill')

pandas.Panel.fillna

Panel.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)
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 / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0, 1, 2, 'items', 'major_axis', 'minor_axis'}
inplace : boolean, 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
filled : Panel

See also:
reindex, asfreq
Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                     [3, 4, np.nan, 1],
...                     [np.nan, np.nan, np.nan, 5],
...                     [np.nan, 3, np.nan, 4]],
...                     columns=list('ABCD'))
```

```bash
A B C D
0 NaN 2.0 NaN 0
1 3.0 4.0 NaN 1
2 NaN NaN NaN 5
3 NaN 3.0 NaN 4
```

Replace all NaN elements with 0s.

```python
>>> df.fillna(0)
```

```bash
A B C D
0 0.0 2.0 0.0 0
1 3.0 4.0 0.0 1
2 0.0 0.0 0.0 5
3 0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```python
>>> df.fillna(method='ffill')
```

```bash
A B C D
0 NaN 2.0 NaN 0
1 3.0 4.0 NaN 1
2 3.0 4.0 NaN 5
3 3.0 3.0 NaN 4
```

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)
```

```bash
A B C D
0 0.0 2.0 2.0 0
1 3.0 4.0 2.0 1
2 0.0 1.0 2.0 5
3 0.0 3.0 2.0 4
```

Only replace the first NaN element.

```python
>>> df.fillna(value=values, limit=1)
```

```bash
A B C D
0 0.0 2.0 2.0 0
1 3.0 4.0 NaN 1
2 NaN 1.0 NaN 5
3 NaN 3.0 NaN 4
```

pandas.Panel.filter

Panel.filter(items=None, like=None, regex=None, axis=None)

Subset rows or columns of dataframe according to labels in the specified index.
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
  
  List of info axis to restrict to (must not all be present)

- **like**: string
  
  Keep info axis where “arg in col == True”

- **regex**: string (regular expression)
  
  Keep info axis with re.search(regex, col) == True

- **axis**: int or string axis name
  
  The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame

**Returns**

same type as input object

**See also:**

*pandas.DataFrame.loc*

**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
  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
mouse  1  2    3
rabbit 4  5    6
```
**pandas.Panel.first**

Panel.first(offset)
Convenience method for subsetting initial periods of time series data based on a date offset.

**Parameters**
- offset : string, DateOffset, dateutil.relativedelta

**Returns**
- subset : type of caller

**Examples**

```
 ts.first('10D') -> First 10 days
```

**pandas.Panel.floordiv**

Panel.floordiv(other, axis=0)
Integer division of series and other, element-wise (binary operator floordiv). Equivalent to panel // other.

**Parameters**
- other : DataFrame or Panel
- axis : [items, major_axis, minor_axis]
  - Axis to broadcast over

**Returns**
- Panel

**See also:**
- Panel.rfloordiv

**pandas.Panel.fromDict**

classmethod Panel.fromDict(data, intersect=False, orient='items', dtype=None)
Construct Panel from dict of DataFrame objects

**Parameters**
- data : dict
  - {field : DataFrame}
- intersect : boolean
  - Intersect indexes of input DataFrames
- orient : {'items', 'minor'}, default 'items'
  - The “orientation” of the data. If the keys of the passed dict should be the items of the result panel, pass ‘items’ (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass ‘minor’
- dtype : dtype, default None
  - Data type to force, otherwise infer

**Returns**
- Panel
pandas.Panel.from_dict

**classmethod** Panel.from_dict(**data**, **intersect=False**, **orient='items'**, **dtype=None**)  
Construct Panel from dict of DataFrame objects

**Parameters**

**data**: dict
   {field : DataFrame}

**intersect**: boolean
   Intersect indexes of input DataFrames

**orient**: {'items', 'minor'}, default 'items'
   The “orientation” of the data. If the keys of the passed dict should be the items of the result panel, pass ‘items’ (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass ‘minor’

**dtype**: dtype, default None
   Data type to force, otherwise infer

**Returns**
Panel

pandas.Panel.ge

Panel.ge(**other**, **axis=None**)  
Wrapper for comparison method ge

pandas.Panel.get

Panel.get(**key**, **default=None**)  
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

**Parameters**

**key**: object

**Returns**

**value**: type of items contained in object

pandas.Panel.get_dtype_counts

Panel.get_dtype_counts()  
Return the counts of dtypes in this object.

pandas.Panel.get_ftype_counts

Panel.get_ftype_counts()  
Return the counts of ftypes in this object.
pandas.Panel.get_value

Panel.get_value(*args, **kwargs)
  Quickly retrieve single value at (item, major, minor) location
  Deprecated since version 0.21.0.
  Please use .at[] or .iat[] accessors.
  Parameters  item : item label (panel item)
                 major : major axis label (panel item row)
                 minor : minor axis label (panel item column)
                 takeable : interpret the passed labels as indexers, default False
  Returns  value : scalar value

pandas.Panel.get_values

Panel.get_values()
  same as values (but handles sparseness conversions)

pandas.Panel.groupby

Panel.groupby(function, axis='major')
  Group data on given axis, returning GroupBy object
  Parameters  function : callable
                  axis : {'major', 'minor', 'items'}, default 'major'
  Returns  grouped : PanelGroupBy

pandas.Panel.gt

Panel.gt(other, axis=None)
  Wrapper for comparison method gt

pandas.Panel.head

Panel.head(n=5)

pandas.Panel.infer_objects

Panel.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.
  New in version 0.21.0.
Returns converted : same type as input object

See also:

- **pandas.to_datetime** Convert argument to datetime.
- **pandas.to_timedelta** Convert argument to timedelta.
- **pandas.to_numeric** Convert argument to numeric type

**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
dtype: int64
```

**pandas.Panel.interpolate**

Panel.interpolate (method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', downcast=None, **kwargs)

Interpolate values according to different methods.

Please note that only method='linear' is supported for DataFrames/Series with a MultiIndex.

**Parameters method** : {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- ‘linear’: ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- ‘time’: interpolation works on daily and higher resolution data to interpolate given length of interval
- ‘index’, ‘values’: use the actual numerical values of the index
- ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘polynomial’ is passed to scipy.interpolate.interp1d. Both ‘polynomial’ and ‘spline’ require that you also specify an order (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
• ‘krogh’, ‘piecewise_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation

• ‘from_derivatives’ refers to BPoly.from_derivatives which replaces ‘piecewise_polynomial’ interpolation method in scipy 0.18

New in version 0.18.1: Added support for the ‘akima’ method Added interpolate method ‘from_derivatives’ which replaces ‘piecewise_polynomial’ in scipy 0.18; backwards-compatible with scipy < 0.18

axis : {0, 1}, default 0
  • 0: fill column-by-column
  • 1: fill row-by-row

limit : int, default None.
  Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default ‘forward’
  If limit is specified, consecutive NaNs will be filled in this direction.
  New in version 0.17.0.

inplace : bool, default False
  Update the NDFrame in place if possible.

downcast : optional, ‘infer’ or None, defaults to None
  Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Returns
Series or DataFrame of same shape interpolated at the NaNs

See also:
reindex, replace, fillna

Examples

Filling in NaNs

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0 0
1 1
2 2
3 3
dtype: float64
```

pandas.Panel.isna

Panel.isna()
  Return a boolean same-sized object indicating if the values are NA.
See also:

```
NDFrame.notna  boolean inverse of isna
NDFrame.isnull  alias of isna
isna  top-level isna
```

**pandas.Panel.isnull**

```
Panel.isnull()  
Return a boolean same-sized object indicating if the values are NA.
```

See also:

```
NDFrame.notna  boolean inverse of isna
NDFrame.isnull  alias of isna
isna  top-level isna
```

**pandas.Panel.iteritems**

```
Panel.iteritems()  
Iterate over (label, values) on info axis
```

This is index for Series, columns for DataFrame, major_axis for Panel, and so on.

**pandas.Panel.join**

```
Panel.join(other, how='left', lsuffix='', rsuffix='')  
Join items with other Panel either on major and minor axes column
```

Parameters

- **other**: Panel or list of Panels

  Index should be similar to one of the columns in this one

- **how**: {'left', 'right', 'outer', 'inner'}

  How to handle indexes of the two objects. Default: ‘left’ for joining on index,
  None otherwise *  left: use calling frame’s index *  right: use input frame’s index
  *  outer: form union of indexes *  inner: use intersection of indexes

- **lsuffix**: string

  Suffix to use from left frame’s overlapping columns

- **rsuffix**: string

  Suffix to use from right frame’s overlapping columns

Returns

- **joined**: Panel
**pandas.Panel.keys**

Panel.keys()

Get the ‘info axis’ (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

**pandas.Panel.kurt**

Panel.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**Parameters**

**axis**: {items (0), major_axis (1), minor_axis (2)}

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame

**numeric_only**: boolean, 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**

**kurt**: DataFrame or Panel (if level specified)

**pandas.Panel.kurtosis**

Panel.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**Parameters**

**axis**: {items (0), major_axis (1), minor_axis (2)}

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame

**numeric_only**: boolean, 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**

**kurt**: DataFrame or Panel (if level specified)
pandas.Panel.last

Panel.last(offset)
Convenience method for subsetting final periods of time series data based on a date offset.

Parameters offset : string, DateOffset, dateutil.relativedelta

Returns subset : type of caller

Examples

ts.last('5M') -> Last 5 months

pandas.Panel.le

Panel.le(other, axis=None)
Wrapper for comparison method le

pandas.Panel.lt

Panel.lt(other, axis=None)
Wrapper for comparison method lt

pandas.Panel.mad

Panel.mad(axis=None, skipna=None, level=None)
Return the mean absolute deviation of the values for the requested axis

Parameters axis : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, default True
    Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame

numeric_only : boolean, 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 mad : DataFrame or Panel (if level specified)

pandas.Panel.major_xs

Panel.major_xs(key)
Return slice of panel along major axis

Parameters key : object
    Major axis label
Returns y : DataFrame

    index -> minor axis, columns -> items

Notes

major_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of major_xs functionality, see MultiIndex Slicers

pandas.Panel.mask

Panel .mask (cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters cond : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as cond.

other : scalar, NDFrame, or callable

Entries where cond is True are replaced with corresponding value from other. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

effects : str, {'raise', 'ignore'}, default ‘raise’

    • raise : allow exceptions to be raised
    • ignore : suppress exceptions. On error return original object

Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.
Returns \textit{wh} : same type as caller

See also:

\textit{DataFrame.where()}

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if \texttt{cond} is False the element is used; otherwise the corresponding element from the DataFrame \texttt{other} is used.

The signature for \textit{DataFrame.where()} differs from \textit{numpy.where()}. Roughly \texttt{df1.where(m, df2)} is equivalent to \texttt{np.where(m, df1, df2)}.

For further details and examples see the mask documentation in \textit{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

>>> s.mask(s > 0)
0  0.0
1  NaN
2  NaN
3  NaN
4  NaN

>>> s.where(s > 1, 10)
0  10.0
1  10.0
2  2.0
3  3.0
4  4.0

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
     A     B
0  0.0  -1.0
1 -2.0   3.0
2 -4.0  -5.0
3  6.0  -7.0
4 -8.0    9.0

>>> df.where(m, -df) == np.where(m, df, -df)
     A     B
0  True  True
1  True  True
2  True  True
```
pandas.Panel.max

Panel.max (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

This method returns the maximum of the values in the object. If you want the index of the maximum, use idxmax. This is the equivalent of the numpy.ndarray method argmax.

Parameters
- axis : {items (0), major_axis (1), minor_axis (2)}
- skipna : boolean, default True
  Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
- numeric_only : boolean, 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
- max : DataFrame or Panel (if level specified)

pandas.Panel.mean

Panel.mean (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return the mean of the values for the requested axis

Parameters
- axis : {items (0), major_axis (1), minor_axis (2)}
- skipna : boolean, default True
  Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
- numeric_only : boolean, 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
- mean : DataFrame or Panel (if level specified)
**pandas.Panel.median**

Panel.median(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return the median of the values for the requested axis

**Parameters**
- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True
  - Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
- **numeric_only**: boolean, 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**
- **median**: DataFrame or Panel (if level specified)

**pandas.Panel.min**

Panel.min(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

This method returns the minimum of the values in the object. If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

**Parameters**
- **axis**: {items (0), major_axis (1), minor_axis (2)}
- **skipna**: boolean, default True
  - Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
- **numeric_only**: boolean, 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**
- **min**: DataFrame or Panel (if level specified)

**pandas.Panel.minor_xs**

Panel.minor_xs(key)

Return slice of panel along minor axis

**Parameters**
- **key**: object
  - Minor axis label

**Returns**
- **y**: DataFrame
  - index -> major axis, columns -> items
Notes

minor_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of minor_xs functionality, see MultiIndex Slicers

pandas.Panel.mod

Panel\_mod\((other, axis=0)\)

Modulo of series and other, element-wise (binary operator mod). Equivalent to panel % other.

- **Parameters**
  - other: DataFrame or Panel
  - axis: \{items, major_axis, minor_axis\}
    - Axis to broadcast over

- **Returns**
  - Panel

See also:

Panel.rmod

pandas.Panel.mul

Panel\_mul\((other, axis=0)\)

Multiplication of series and other, element-wise (binary operator mul). Equivalent to panel * other.

- **Parameters**
  - other: DataFrame or Panel
  - axis: \{items, major_axis, minor_axis\}
    - Axis to broadcast over

- **Returns**
  - Panel

See also:

Panel.rmul

pandas.Panel.multiply

Panel.multiply\((other, axis=0)\)

Multiplication of series and other, element-wise (binary operator mul). Equivalent to panel * other.

- **Parameters**
  - other: DataFrame or Panel
  - axis: \{items, major_axis, minor_axis\}
    - Axis to broadcast over

- **Returns**
  - Panel

See also:

Panel.rmul
pandas.Panel.ne

Panel.ne(other, axis=None)
Wrapper for comparison method ne

pandas.Panel.notna

Panel.notna()
Return a boolean same-sized object indicating if the values are not NA.

See also:

NDFrame.isna boolean inverse of notna
NDFrame.notnull alias of notna

notna top-level notna

pandas.Panel.notnull

Panel.notnull()
Return a boolean same-sized object indicating if the values are not NA.

See also:

NDFrame.isna boolean inverse of notna
NDFrame.notnull alias of notna

notna top-level notna

pandas.Panel.pct_change

Panel.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)
Percent change over given number of periods.

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 offset alias string, optional
  Increment to use from time series API (e.g. ‘M’ or BDay())

Returns:
- chg : NDFrame
Notes

By default, the percentage change is calculated along the stat axis: 0, or Index, for DataFrame and 1, or minor for Panel. You can change this with the axis keyword argument.

pandas.Panel.pipe

Panel.pipe(func, *args, **kwargs)
Apply func(self, *args, **kwargs)

Parameters func : function
    function to apply to the NDFrame. 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 NDFrame.

    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:
pandas.DataFrame.apply, pandas.DataFrame.applymap, pandas.Series.map

Notes

Use .pipe when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```python
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(f, arg2=b, arg3=c)
... )
```

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((f, 'arg2'), arg1=a, arg3=c)
... )
```
pandas.Panel.pop

Panel.pop(item)
Return item and drop from frame. Raise KeyError if not found.

Parameters item : str
   Column label to be popped

Returns popped : 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'))
>>> df
   name class  max_speed
0   falcon   bird      389.0
1   parrot   bird       24.0
2     lion  mammal      80.5
3  monkey  mammal        NaN

>>> df.pop('class')
   name  max_speed
0   falcon      389.0
1   parrot       24.0
2     lion       80.5
3  monkey        NaN
Name: class, dtype: object

>>> df
   name  max_speed
0   falcon      389.0
1   parrot       24.0
2     lion       80.5
3  monkey        NaN
```

pandas.Panel.pow

Panel.pow(other, axis=0)
Exponential power of series and other, element-wise (binary operator pow). Equivalent to panel ** other.

Parameters other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
   Axis to broadcast over

Returns Panel

See also:
Panel.rpow
pandas.Panel.prod

Panel.prod(\texttt{axis=None, skipna=None, level=None, numeric\_only=None, **kwargs})

Return the product of the values for the requested axis

\textbf{Parameters}  \texttt{axis: \{items (0), major\_axis (1), minor\_axis (2)\}}

\texttt{skipna} : boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, the result will be NA

\texttt{level} : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

\texttt{numeric\_only} : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

\textbf{Returns}  \texttt{prod} : DataFrame or Panel (if level specified)

pandas.Panel.product

Panel.product(\texttt{axis=None, skipna=None, level=None, numeric\_only=None, **kwargs})

Return the product of the values for the requested axis

\textbf{Parameters}  \texttt{axis: \{items (0), major\_axis (1), minor\_axis (2)\}}

\texttt{skipna} : boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, the result will be NA

\texttt{level} : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

\texttt{numeric\_only} : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

\textbf{Returns}  \texttt{prod} : DataFrame or Panel (if level specified)

pandas.Panel.radd

Panel.radd(\texttt{other, axis=0})

Addition of series and other, element-wise (binary operator radd). Equivalent to \texttt{other + panel}.

\textbf{Parameters}  \texttt{other} : DataFrame or Panel

\texttt{axis} : \{items, major\_axis, minor\_axis\}

Axis to broadcast over

\textbf{Returns}  Panel
See also:

*Panel.add*

**pandas.Panel.rank**

```
Panel.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)
```

Compute numerical data ranks (1 through n) along axis. 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'}
  - 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

- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. Valid only for DataFrame or Panel objects

- **na_option**: {'keep', ‘top’, ‘bottom’}
  - keep: leave NA values where they are
  - top: smallest rank if ascending
  - bottom: smallest rank if descending

- **ascending**: boolean, default True
  - False for ranks by high (1) to low (N)

- **pct**: boolean, default False
  - Computes percentage rank of data

**Returns**

*Panel.rdiv* : same type as caller

**pandas.Panel.rdiv**

```
Panel.rdiv(other, axis=0)
```

Floating division of series and other, element-wise (binary operator `rtruediv`). Equivalent to `other / panel`.

**Parameters**

- **other**: DataFrame or Panel

- **axis**: {items, major_axis, minor_axis}
  - Axis to broadcast over

**Returns**

*Panel*
See also:

`Panel.truediv`

**pandas.Panel.reindex**

```
pandas.Panel.reindex(*args, **kwargs)
```

Conform Panel to new index with optional filling logic, placing 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**

- **items, major_axis, minor_axis**: array-like, optional (should be specified using keywords)
  - New labels / index to conform to. Preferably an Index object to avoid duplicating data
- **method**: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}, optional
  - method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
    - 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**: boolean, 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 `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.
  - New in version 0.17.0.
  - New in version 0.21.0: (list-like tolerance)

**Returns**

reindexed : Panel
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']
>>> df.reindex(new_index)
     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 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 0.07
 Iceweasel       0 0.00
Comodo Dragon    0 0.00
IE10             404 0.08
 Chrome           200 0.02
```

```python
>>> df.reindex(new_index, fill_value='missing')
     http_status  response_time
   Safari      404 0.07
 Iceweasel    missing missing
Comodo Dragon missing missing
IE10             404 0.08
 Chrome           200 0.02
```

We can also reindex the columns.
Or we can use “axis-style” keyword arguments

To further illustrate the filling functionality in reindex, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

Suppose we decide to expand the dataframe to cover a wider date range.

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 backpropagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.
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.Panel.reindex_axis`**

Panel.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=nan)

Conform input object to new index with optional filling logic, placing 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**
- **labels**: array-like
  New labels / index to conform to. Preferably an Index object to avoid duplicating data
- **axis**: {0, 1, 2, ‘items’, ‘major_axis’, ‘minor_axis’}
- **method**: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}, optional
  Method to use for filling holes in reindexed DataFrame:
  - 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**: boolean, 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
- **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 `abs(index[indexer] - target) <= tolerance`. 

```
<table>
<thead>
<tr>
<th>Date</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-12-30</td>
<td>100</td>
</tr>
<tr>
<td>2009-12-31</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-02</td>
<td>101</td>
</tr>
<tr>
<td>2010-01-03</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>100</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>89</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>88</td>
</tr>
<tr>
<td>2010-01-07</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```
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.

New in version 0.17.0.
New in version 0.21.0: (list-like tolerance)

Returns reindexed : Panel

See also:
reindex, reindex_like

Examples

```python
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

pandas.Panel.reindex_like

Panel.reindex_like (other, method=None, copy=True, limit=None, tolerance=None)
Return an object with matching indices to myself.

Parameters other : Object
method : string or None
          Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.
          New in version 0.17.0.
          New in version 0.21.0: (list-like tolerance)

Returns reindexed : same as input

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

pandas.Panel.rename

Panel.rename (items=None, major_axis=None, minor_axis=None, **kwargs)
Alter axes input function or functions. 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 (Series only).

Parameters items, major_axis, minor_axis : scalar, list-like, dict-like or function, optional
Scalar or list-like will alter the Series.name attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis’ values

**copy**: boolean, default True

Also copy underlying data

**inplace**: boolean, default False

Whether to return a new Panel. 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.

**Returns renamed**: Panel (new object)

**See also**:

pandas.NDFrame.rename_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 2
4 3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0 1
3 2
5 3
dtype: int64
```

Since DataFrame doesn’t have a .name attribute, only mapping-type arguments are allowed.

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

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.
pandas: powerful Python data analysis toolkit, Release 0.21.0

```python
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
    a  c
0  1  4
1  2  5
2  3  6
```

```python
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
    a  B
0  1  4
1  2  5
2  3  6
```

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
```

See the user guide for more.

pandas.Panel.rename_axis

Panel.rename_axis(mapper, axis=0, copy=True, inplace=False)

Alter the name of the index or columns.

Parameters mapper : scalar, list-like, optional

Value to set the axis name attribute.

axis : int or string, default 0

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Returns renamed : type of caller or None if inplace=True

See also:

pandas.Series.rename, pandas.DataFrame.rename, pandas.Index.rename

Notes

Prior to version 0.21.0, rename_axis could also be used to change the axis labels by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use rename instead.
Examples

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
   A  B
foo
0  1  4
1  2  5
2  3  6
```

```python
>>> df.rename_axis("bar", axis="columns")
   bar  A  B
0   1  4
1   2  5
2   3  6
```

**pandas.Panel.replace**

Panel.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)
Replace values given in ‘to_replace’ with ‘value’.

**Parameters**

to_replace : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - 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 regexs 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 and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {‘a’: {‘b’: nan}}, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- 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 use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace**: boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**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. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method**: string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a list.

**Returns** filled : NDFrame

Raises **AssertionError**

- If `regex` is not a bool and `to_replace` is not None.

**TypeError**

- 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.

**ValueError**

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

See also:

NDFrame.reindex, NDFrame.asfreq, NDFramefillna

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.
pandas.Panel.resample

Panel.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

Parameters

- **rule**: string
  
  the offset string or object representing target conversion

- **axis**: int, optional, default 0

- **closed**: {'right', 'left'}
  
  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'}
  
  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'}
  
  For PeriodIndex only, controls whether to use the start or end of rule

- **loffset**: timedelta
  
  Adjust the resampled time labels

- **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

- **on**: string, optional
  
  For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

  New in version 0.19.0.

- **level**: string or int, optional
  
  For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

  New in version 0.19.0.

Notes

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
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
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.

```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(array_like):
...     return np.sum(array_like)+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`.

```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('M', convention='start').asfreq().head()
2012-01  1.0
2012-02  NaN
2012-03  NaN
2012-04  NaN
2012-05  NaN
Freq: M, dtype: float64
```

Resample by month using ‘start’ `convention`. Values are assigned to the first month of the period.

```python
>>> s.resample('M', convention='start').asfreq().head()
2012-01 1.0
2012-02 NaN
2012-03 NaN
2012-04 NaN
2012-05 NaN
Freq: M, dtype: float64
```

Resample by month using ‘end’ `convention`. Values are assigned to the last month of the period.

```python
>>> s.resample('M', convention='end').asfreq()
2012-12 1.0
2013-01 NaN
2013-02 NaN
```
For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```python
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
   a  b  c  d
time
2000-01-01 00:00:00 0  3  6  9
2000-01-01 00:03:00 0  3  6  9
2000-01-01 00:06:00 0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```python
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                    columns=['a', 'b', 'c', 'd'],
                    index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()
   a  b  c  d
2000-01-01 00:00:00 0  6 12 18
2000-01-01 00:03:00 0  4  8 12
```

**pandas.Panel.rfloordiv**

`Panel.rfloordiv(other, axis=0)`

Integer division of series and other, element-wise (binary operator `rfloordiv`). Equivalent to `other // panel`.

**Parameters**

- `other` : DataFrame or Panel
- `axis` : {items, major_axis, minor_axis}

  Axis to broadcast over

**Returns**

`Panel` to broadcast over

**See also:**

`Panel.floordiv`
pandas.Panel.rmod

Panel.rmod(other, axis=0)
Modulo of series and other, element-wise (binary operator rmod). Equivalent to other % panel.

Parameters  other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
    Axis to broadcast over

Returns  Panel

See also:
Panel.mod

pandas.Panel.rmul

Panel.rmul(other, axis=0)
Multiplication of series and other, element-wise (binary operator rmul). Equivalent to other * panel.

Parameters  other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
    Axis to broadcast over

Returns  Panel

See also:
Panel.mul

pandas.Panel.round

Panel.round(decimals=0, *args, **kwargs)
Round each value in Panel to a specified number of decimal places.

New in version 0.18.0.

Parameters  decimals : int
    Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns  Panel object

See also:
numpy.around

pandas.Panel.rpow

Panel.rpow(other, axis=0)
Exponential power of series and other, element-wise (binary operator rpow). Equivalent to other ** panel.

Parameters  other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
pandas.Panel.rsub

Panel.rsub(other, axis=0)
Subtraction of series and other, element-wise (binary operator rsub). Equivalent to other - panel.

Parameters other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
Axis to broadcast over

Returns Panel

See also:
Panel.pow

pandas.Panel.rtruediv

Panel.rtruediv(other, axis=0)
Floating division of series and other, element-wise (binary operator rtruediv). Equivalent to other / panel.

Parameters other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
Axis to broadcast over

Returns Panel

See also:
Panel.truediv

pandas.Panel.sample

Panel.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)
Returns a random sample of items from an axis of object.

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 : boolean, optional
Sample with or without replacement. Default = False.
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. inf and -inf values not allowed.

random_state : int or numpy.random.RandomState, optional

Seed for the random number generator (if int), or numpy RandomState object.

axis : int or string, optional

Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns A new object of same type as caller.

Examples

Generate an example Series and DataFrame:

```python
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64

>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
     A         B         C         D
0  0.016443  -2.318952  -0.566372  -1.028078
1 -1.051921   0.438836   0.658280  -0.175797
2 -1.243569  -0.364626  -0.215065   0.057736
3  1.768216   0.404512  -0.385604  -1.457834
4  1.072446  -1.137172   0.314194  -0.046661
dtype: float64
```

Next extract a random sample from both of these objects...

3 random elements from the Series:

```python
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the DataFrame with replacement:

```python
>>> df.sample(frac=0.1, replace=True)
     A         B         C         D
35  1.981780  0.142106  1.817165  -0.290805
49 -1.336199  -0.448634  -0.789640   0.217116
```
pandas.Panel.select

Panel.select(crit, axis=0)
Return data corresponding to axis labels matching criteria

DEPRECATED: use df.loc[df.index.map(crit)] to select via labels

Parameters
- crit : function
  To be called on each index (label). Should return True or False
- axis : int

Returns
- selection : type of caller

pandas.Panel.sem

Panel.sem(axis=None, skipna=None, 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 : {items (0), major_axis (1), minor_axis (2)}
  - skipna : boolean, 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 DataFrame
  - ddof : int, default 1
    degrees of freedom
  - numeric_only : boolean, 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
- sem : DataFrame or Panel (if level specified)

pandas.Panel.set_axis

Panel.set_axis(labels, axis=0, inplace=None)
Assign desired index to given axis

Parameters
- labels : list-like or Index
  The values for the new index
- axis : int or string, default 0
- inplace : boolean, default None
Whether to return a new NDFrame instance.

WARNING: inplace=None currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

.. versionadded:: 0.21.0

The signature is make consistent to the rest of the API. Previously, the “axis” and “labels” arguments were respectively the first and second positional arguments.

Returns renamed : NDFrame or None

An object of same type as caller if inplace=False, None otherwise.

See also:
pandas.NDFrame.rename

Examples

```python
>>> s = pd.Series([1, 2, 3])
```
```python
>>> s
0 1
1 2
2 3
dtype: int64
```
```python
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
```
```python
  a 1
  b 2
  c 3
dtype: int64
```
```python
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```
```python
>>> df.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
```
```python
  A   B
  a  1  4
  b  2  5
  c  3  6
```
```python
>>> df.set_axis(['I', 'II'], axis=1, inplace=False)
```
```python
  I   II
  0  1  4
  1  2  5
  2  3  6
```
```python
>>> df.set_axis(['i', 'ii'], axis=1, inplace=True)
```
```python
>>> df
```
```python
  i    ii
  0  1  4
  1  2  5
  2  3  6
```
```python
```

pandas.Panel.set_value

Panel.set_value(*args, **kwargs)

Quickly set single value at (item, major, minor) location

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.
**Parameters**

- **item**: item label (panel item)
  - **major**: major axis label (panel item row)
  - **minor**: minor axis label (panel item column)
  - **value**: scalar
  - **takeable**: interpret the passed labels as indexers, default False

**Returns**

- **panel**: Panel
  
  If label combo is contained, will be reference to calling Panel, otherwise a new object

---

**pandas.Panel.shift**

Panel.shift(periods=1, freq=None, axis='major')

Shift index by desired number of periods with an optional time freq. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original. This is different from the behavior of DataFrame.shift()

- **Parameters**
  - **periods**: int
    - Number of periods to move, can be positive or negative
  - **freq**: DateOffset, timedelta, or time rule string, optional
  - **axis**: {'items', 'major', 'minor'} or {0, 1, 2}

- **Returns**
  - **shifted**: Panel

---

**pandas.Panel.skew**

Panel.skew(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return unbiased skew over requested axis Normalized by N-1

- **Parameters**
  - **axis**: {items (0), major_axis (1), minor_axis (2)}
  - **skipna**: boolean, default True
    - Exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
  - **numeric_only**: boolean, 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**
  - **skew**: DataFrame or Panel (if level specified)

---

**pandas.Panel.slice_shift**

Panel.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.Panel.sort_index**

**Panel.sort_index**(*axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True*)

Sort object by labels (along an axis)

Parameters **axis** : axes to direct sorting

- **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** : boolean, default True
  
  Sort ascending vs. descending

- **inplace** : bool, default False
  
  if True, perform operation in-place

- **kind** : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
  
  Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

- **na_position** : {'first', 'last'}, default 'last'
  
  *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

Returns **sorted_obj** : NDFrame

**pandas.Panel.sort_values**

**Panel.sort_values**(*by=None, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')

NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.
**pandas.Panel.squeeze**

```python
Panel.squeeze(axis=None)
```

Squeeze length 1 dimensions.

**Parameters**
- **axis** : None, integer or string axis name, optional
  - The axis to squeeze if 1-sized.
  - New in version 0.20.0.

**Returns**
- scalar if 1-sized, else original object

**pandas.Panel.std**

```python
Panel.std(axis=None, skipna=None, 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** : {items (0), major_axis (1), minor_axis (2)}
  - skipna : boolean, 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 DataFrame
  - ddof : int, default 1
    - degrees of freedom
  - numeric_only : boolean, 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**
- std : DataFrame or Panel (if level specified)

**pandas.Panel.sub**

```python
Panel.sub(other, axis=0)
```

Subtraction of series and other, element-wise (binary operator sub). Equivalent to `panel - other`.

**Parameters**
- **other** : DataFrame or Panel
  - **axis** : {items, major_axis, minor_axis}

**Returns**
- Panel

See also:
- `Panel.rsub`
pandas.Panel.subtract

Panel.subtract(other, axis=0)
Subtraction of series and other, element-wise (binary operator sub). Equivalent to panel - other.

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Returns Panel
See also:
Panel.rsub

pandas.Panel.sum

Panel.sum(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the sum of the values for the requested axis

Parameters
axis : {items (0), major_axis (1), minor_axis (2)}
skipna : boolean, default True
exclude NA/null values. If an entire row/column is NA or empty, 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 DataFrame
numeric_only : boolean, 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 sum : DataFrame or Panel (if level specified)

pandas.Panel.swapaxes

Panel.swapaxes(axis1, axis2, copy=True)
Interchange axes and swap values axes appropriately

Returns y : same as input

pandas.Panel.swaplevel

Panel.swaplevel(i=-2, j=-1, axis=0)
Swap levels i and j in a MultiIndex on a particular axis

Parameters
i, j : int, string (can be mixed)
Level of index to be swapped. Can pass level name as string.
Returns swapped : type of caller (new object)

Changed in version 0.18.1: The indexes i and j are now optional, and default to the two innermost levels of the index.

pandas.Panel.tail

Panel.tail(n=5)

pandas.Panel.take

Panel.take(indices, axis=0, convert=None, is_copy=True, **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 : int, default 0

The axis on which to select elements. “0” means that we are selecting rows, “1” means that we are selecting columns, etc.

convert : bool, default True

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

Whether to convert negative indices into positive ones. For example, -1 would map to the len(axis) - 1. The conversions are similar to the behavior of indexing a regular Python list.

is_copy : bool, default True

Whether to return a copy of the original object or not.

Returns taken : type of caller

An array-like containing the elements taken from the object.

See also:

numpy.ndarray.take, numpy.take

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
```
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])
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.Panel.toLong**

`Panel.toLong(*args, **kwargs)`

**pandas.Panel.to_clipboard**

`Panel.to_clipboard(excel=None, sep=None, **kwargs)`

Attempt to write text representation of object to the system clipboard. This can be pasted into Excel, for example.

- `excel`: boolean, defaults to `True`
  - If `True`, use the provided separator, writing in a csv format for allowing easy pasting into excel. If `False`, write a string representation of the object to the clipboard
  - `sep`: optional, defaults to `tab`
  - Other keywords are passed to `to_csv`

**Notes**

**Requirements for your platform**

- Linux: `xclip`, or `xsel` (with gtk or PyQt4 modules)
- Windows: none
• OS X: none

**pandas.Panel.to_dense**

Panel.to_dense()  
Return dense representation of NDFrame (as opposed to sparse)

**pandas.Panel.to_excel**

Panel.to_excel(path, na_rep='', engine=None, **kwargs)  
Write each DataFrame in Panel to a separate excel sheet

**Parameters**  
- **path**: string or ExcelWriter object  
  File path or existing ExcelWriter
- **na_rep**: string, default ''  
  Missing data representation
- **engine**: string, default None  
  write engine to use - you can also set this via the options io.excel.xlsx.writer, io.excel.xls.writer, and io.excel.xlsm.writer.

**Other Parameters**  
- **float_format**: string, default None  
  Format string for floating point numbers
- **cols**: sequence, optional  
  Columns to write
- **header**: boolean or list of string, default True  
  Write out column names. If a list of string is given it is assumed to be aliases for the column names
- **index**: boolean, default True  
  Write row names (index)
- **index_label**: string or sequence, 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 DataFrame uses MultiIndex.
- **startrow**: upper left cell row to dump data frame
- **startcol**: upper left cell column to dump data frame

**Notes**

Keyword arguments (and na_rep) are passed to the to_excel method for each DataFrame written.
**pandas.Panel.to_frame**

Panel.to_frame(filter_observations=True)
Transform wide format into long (stacked) format as DataFrame whose columns are the Panel’s items and whose index is a MultiIndex formed of the Panel’s major and minor axes.

**Parameters**

filter_observations : boolean, default True
Drop (major, minor) pairs without a complete set of observations across all the items

**Returns**

y : DataFrame

**pandas.Panel.to_hdf**

Panel.to_hdf(path_or_buf, key, **kwargs)
Write the contained data to an HDF5 file using HDFStore.

**Parameters**

path_or_buf : the path (string) or HDFStore object
key : string
identifier for the group in the store
mode : optional, {'a', 'w', 'r+'}, default 'a'
  '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+' It is similar to 'a', but the file must already exist.
format : ‘fixed(f)|table(t)’, default is ‘fixed’
  fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable
  table(t) [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 : boolean, default False
  For Table formats, 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.
  Applicable only to format=’table’.
complevel : int, 0-9, default None
  Specifies a compression level for data. A value of 0 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.
fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

## pandas.Panel.to_json

```
pandas.Panel.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=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** : the path or buffer to write the result string
  
  if this is None, return the converted string

- **orient** : string
  
  - Series
    
    - default is ‘index’
    
    - allowed values are: {‘split’,‘records’,’index’}
  
  - DataFrame
    
    - default is ‘columns’
    
    - allowed values are: {‘split’,‘records’,’index’,’columns’,’values’}
  
  - 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, and the data component is like orient=’records’.

  Changed in version 0.20.0.

- **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** : 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** : string, default ‘ms’ (milliseconds)
pandas: powerful Python data analysis toolkit, Release 0.21.0

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 : boolean, default False
If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError
if incorrect ‘orient’ since others are not list like.
New in version 0.19.0.
compression : {None, ‘gzip’, ‘bz2’, ‘xz’}
A string representing the compression to use in the output file, only used when
the first argument is a filename
New in version 0.21.0.
Returns same type as input object with filtered info axis
See also:
pd.read_json
Examples
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...
index=['row 1', 'row 2'],
...
columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
"index":["row 1","row 2"],
"data":[["a","b"],["c","d"]]}'

Encoding/decoding a Dataframe using 'index' formatted JSON:
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'

Encoding with Table Schema
>>> 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": "0.20.0"},

1702

Chapter 34. API Reference


`pandas.Panel.to_latex`:

`Panel.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)`

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`.

Changed in version 0.20.2: Added to Series

`to_latex`-specific options:

- **bold_rows** [boolean, default False] Make the row labels bold in the output
- **column_format** [str, default None] The columns format as specified in LaTeX table format e.g. `rcl` for 3 columns
- **longtable** [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a `\usepackage{longtable}` to your LaTeX preamble.
- **escape** [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.
- **encoding** [str, default None] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.
- **decimal** [string, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.
  
  New in version 0.18.0.
- **multicolumn** [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.
  
  New in version 0.20.0.
- **multicolumn_format** [str, default ‘l’] The alignment for multicolumns, similar to `column_format` The default will be read from the config module.
  
  New in version 0.20.0.
- **multirow** [boolean, 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.

New in version 0.20.0.

`pandas.Panel.to_long`:

`Panel.to_long(*args, **kwargs)`
pandas.Panel.to_msgpack

Panel.to_msgpack(path_or_buf=None, encoding='utf-8', **kwargs)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters
- **path**: string File path, buffer-like, or None
  - if None, return generated string
- **append**: boolean whether to append to an existing msgpack
  - (default is False)
- **compress**: type of compressor (zlib or blosc), default to None (no compression)

pandas.Panel.to_pickle

Panel.to_pickle(path, compression='infer', protocol=4)
Pickle (serialize) object to input file path.

Parameters
- **path**: string

- **compression**: {'infer', 'gzip', 'bz2', 'xz', None}, default 'infer'
  - a string representing the compression to use in the output file
  - New in version 0.20.0.
- **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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.
  - New in version 0.21.0.

pandas.Panel.to_sparse

Panel.to_sparse(*args, **kwargs)
NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Convert to SparsePanel

pandas.Panel.to_sql

Panel.to_sql(name, con, flavor=None, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None)
Write records stored in a DataFrame to a SQL database.
**Parameters**

- **name** : string
  
  Name of SQL table

- **con** : SQLAlchemy engine or DBAPI2 connection (legacy mode)
  
  Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

- **flavor** : ‘sqlite’, default None
  
  Deprecated since version 0.19.0: ‘sqlite’ is the only supported option if SQLAlchemy is not used.

- **schema** : string, default None
  
  Specify the schema (if database flavor supports this). If None, use default schema.

- **if_exists** : {'fail', 'replace', 'append'}, default ‘fail’
  
  - fail: If table exists, do nothing.
  - replace: If table exists, drop it, recreate it, and insert data.
  - append: If table exists, insert data. Create if does not exist.

- **index** : boolean, default True
  
  Write DataFrame index as a column.

- **index_label** : string 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, default None
  
  If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

- **dtype** : dict of column name to SQL type, default None
  
  Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

---

**pandas.Panel.to_xarray**

*Panel.to_xarray()*

Return an xarray object from the pandas object.

- **Returns**
  
  - a DataArray for a Series
  - a Dataset for a DataFrame
  - a DataArray for higher dims

---

**Notes**

See the xarray docs
Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2],
                    'B': ['foo', 'bar', 'foo'],
                    'C': np.arange(4.,7))

>>> df
   A  B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0

>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (index: 3)
Coordinates:
 * index (index) int64 0 1 2
Data variables:
  A (index) int64 1 1 2
  B (index) object 'foo' 'bar' 'foo'
  C (index) float64 4.0 5.0 6.0

>>> df = pd.DataFrame({'A': [1, 1, 2],
                    'B': ['foo', 'bar', 'foo'],
                    'C': np.arange(4.,7))
                     .set_index(['B','A'])

>>> df
   C
  B A
foo 1 4.0
bar 1 5.0
foo 2 6.0

>>> df.to_xarray()
<xarray.Dataset>
Dimensions: (A: 2, B: 2)
Coordinates:
 * B (B) object 'bar' 'foo'
 * A (A) int64 1 2
Data variables:
  C (B, A) float64 5.0 nan 4.0 6.0

>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                items=list('ABCD'),
                major_axis=pd.date_range('20130101', periods=3),
                minor_axis=['first', 'second'])

>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second

>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7]])
```
pandas.Panel.transpose

Panel.transpose(*args, **kwargs)

Permute the dimensions of the Panel

Parameters args : three positional arguments: each one of  
\{0, 1, 2, \textquotesingle{}items\}, \textquotesingle{}major_axis\}, \textquotesingle{}minor_axis\}

- \textit{copy} \ [boolean, default False] Make a copy of the underlying data. Mixed-dtype  
data will always result in a copy

Returns y : same as input

Examples

```python
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

pandas.Panel.truediv

Panel.truediv\ (other, axis=0)  

Floating division of series and other, element-wise (binary operator \textit{truediv}). Equivalent to panel /  
other.

Parameters other : DataFrame or Panel  

- axis : \{\textit{items}, \textit{major_axis}, \textit{minor_axis}\}  

Axis to broadcast over

Returns Panel

See also:

Panel.rtruediv
pandas.Panel.truncate

Panel.truncate(before=None, after=None, axis=None, copy=True)

Truncates a sorted DataFrame/Series before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters

- **before**: date, string, int
  - Truncate all rows before this index value

- **after**: date, string, int
  - Truncate all rows after this index value

- **axis**: {0 or ‘index’, 1 or ‘columns’}
  - 0 or ‘index’: apply truncation to rows
  - 1 or ‘columns’: apply truncation to columns
  
  Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels)

- **copy**: boolean, default is True,
  - return a copy of the truncated section

Returns truncated : type of caller

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.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The index values in `truncate` can be datetimes or string dates. 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
>>> df = pd.DataFrame({'A': [1, 2, 3, 4, 5],
...                    'B': [6, 7, 8, 9, 10],
...                    'C': [11, 12, 13, 14, 15]},
...                   index=['a', 'b', 'c', 'd', 'e'])
>>> df.truncate(before='b', after='d')
   A  B  C
b  2  7  12
c  3  8  13
d  4  9  14
```

```python
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.truncate('2016-01-05', '2016-01-10').tail()
   A
2016-01-09  23:59:56  1
2016-01-09  23:59:57  1
2016-01-09  23:59:58  1
```
pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>2016-01-09 23:59:59</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-10 00:00:00</td>
<td>1</td>
</tr>
</tbody>
</table>

```python
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
   A
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.Panel.tshift**

Panel.tshift (periods=1, freq=None, axis='major')

**pandas.Panel.tz_convert**

Panel.tz_convert (tz, axis=0, level=None, copy=True)
Convert tz-aware axis to target time zone.

Parameters tz : string or pytz.timezone object
axis : the axis to convert
level : int, str, default None
   If axis ia a MultiIndex, convert a specific level. Otherwise must be None
copy : boolean, default True
   Also make a copy of the underlying data

Raises TypeError
   If the axis is tz-naive.

**pandas.Panel.tz_localize**

Panel.tz_localize (tz, axis=0, level=None, copy=True, ambiguous='raise')
Localize tz-naive TimeSeries to target time zone.

Parameters tz : string or pytz.timezone object
axis : the axis to localize
level : int, str, default None
   If axis ia a MultiIndex, localize a specific level. Otherwise must be None
copy : boolean, default True
   Also make a copy of the underlying data
ambiguous : ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’
   • ‘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

**infer_dst**: boolean, default False

Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order

**Raises** TypeError

If the TimeSeries is tz-aware and tz is not None.

**pandas.Panel.update**

Panel.update(other='left', overwrite=True, filter_func=None, raise_conflict=False)

Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

**Parameters**

other : Panel, or object coercible to Panel

join : How to join individual DataFrames

{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘left’

overwrite : boolean, default True

If True then overwrite values for common keys in the calling panel

filter_func : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict : bool

If True, will raise an error if a DataFrame and other both contain data in the same place.

**pandas.Panel.var**

Panel.var(axis=None, skipna=None, 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 : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, 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 DataFrame

ddof : int, default 1

degrees of freedom

numeric_only : boolean, 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 var : DataFrame or Panel (if level specified)

pandas.Panel.where

Panel.where (cond, other=nan, inplace=False, axis=None, level=None, errors='raise',
          try_cast=False, raise_on_error=None)

Return an object of same shape as self and whose corresponding entries are from self where cond is True
and otherwise are from other.

Parameters cond : boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as cond.

other : scalar, NDFrame, or callable

Entries where cond is False are replaced with corresponding value from other. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn’t check it).

New in version 0.18.1: A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

errors : str, {'raise', 'ignore'}, default ‘raise’

• raise : allow exceptions to be raised
• ignore : suppress exceptions. On error return original object

Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns wh : same type as caller

See also:

DataFrame.mask()
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
```

```python
>>> s.mask(s > 0)
0     0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```python
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

```python
>>> df.where(m, -df) == df.mask(~m, -df)
     A    B
0   True True
1   True True
2   True True
```
pandas.Panel.xs

Panel.xs(key, axis=1)
Return slice of panel along selected axis

Parameters

- **key**: object
  - Label
- **axis**: {'items', 'major', 'minor'}, default 1/'major'

Returns

- y : ndim(self)-1

Notes

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of xs functionality, see MultiIndex Slicers

34.5.2 Attributes and underlying data

Axes

- **items**: axis 0; each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1; the index (rows) of each of the DataFrames
- **minor_axis**: axis 2; the columns of each of the DataFrames

| Panel.values | Numpy representation of NDFrame |
| Panel.axes | Return index label(s) of the internal NDFrame |
| Panel.ndim | Number of axes / array dimensions |
| Panel.size | number of elements in the NDFrame |
| Panel.shape | Return a tuple of axis dimensions |
| Panel.dtypes | Return the dtypes in this object. |
| Panel.ftypes | Return the ftypes (indication of sparse/dense and dtype) in this object. |
| Panel.get_dtype_counts() | Return the counts of dtypes in this object. |
| Panel.get_ftype_counts() | Return the counts of ftypes in this object. |

34.5.3 Conversion

- **Panel.astype(dtype[, copy, errors])**: Cast a pandas object to a specified dtype dtype.
- **Panel.copy([deep])**: Make a copy of this objects data.
- **Panel.isna()**: Return a boolean same-sized object indicating if the values are NA.

Continued on next page
Table 34.76 – continued from previous page

Panel.notna()  Return a boolean same-sized object indicating if the values are not NA.

34.5.4 Getting and setting

Panel.get_value(*args, **kwargs)  Quickly retrieve single value at (item, major, minor) location.
Panel.set_value(*args, **kwargs)  Quickly set single value at (item, major, minor) location.

34.5.5 Indexing, iteration, slicing

Panel.at  Fast label-based scalar accessor.
Panel.iat  Fast integer location scalar accessor.
Panel.loc  Purely label-location based indexer for selection by label.
Panel.iloc  Purely integer-location based indexing for selection by position.
Panel.__iter__()  Iterate over info axis.
Panel.iteritems()  Iterate over (label, values) on info axis.
Panel.pop(item)  Return item and drop from frame.
Panel.xs(key[, axis])  Return slice of panel along selected axis.
Panel.major_xs(key)  Return slice of panel along major axis.
Panel.minor_xs(key)  Return slice of panel along minor axis.

34.5.5.1 pandas.Panel.__iter__

Panel.__iter__()  Iterate over info axis.

For more information on .at, .iat, .loc, and .iloc, see the indexing documentation.

34.5.6 Binary operator functions

Panel.add(other[, axis])  Addition of series and other, element-wise (binary operator add).
Panel.sub(other[, axis])  Subtraction of series and other, element-wise (binary operator sub).
Panel.mul(other[, axis])  Multiplication of series and other, element-wise (binary operator mul).
Panel.div(other[, axis])  Floating division of series and other, element-wise (binary operator truediv).
Panel.truediv(other[, axis])  Floating division of series and other, element-wise (binary operator truediv).
Panel.floordiv(other[, axis])  Integer division of series and other, element-wise (binary operator floordiv).
Panel.mod(other[, axis])  Modulo of series and other, element-wise (binary operator mod).

Continued on next page
Panel `pow`:
- Exponential power of series and other, element-wise (binary operator `pow`).

Panel `radd`:
- Addition of series and other, element-wise (binary operator `radd`).

Panel `rsub`:
- Subtraction of series and other, element-wise (binary operator `rsub`).

Panel `rmul`:
- Multiplication of series and other, element-wise (binary operator `rmul`).

Panel `rdiv`:
- Floating division of series and other, element-wise (binary operator `rtruediv`).

Panel `rtruediv`:
- Floating division of series and other, element-wise (binary operator `rtruediv`).

Panel `rfloordiv`:
- Integer division of series and other, element-wise (binary operator `rfloordiv`).

Panel `rmod`:
- Modulo of series and other, element-wise (binary operator `rmod`).

Panel `rpow`:
- Exponential power of series and other, element-wise (binary operator `rpow`).

Panel `lt`:
- Wrapper for comparison method `lt`.

Panel `gt`:
- Wrapper for comparison method `gt`.

Panel `le`:
- Wrapper for comparison method `le`.

Panel `ge`:
- Wrapper for comparison method `ge`.

Panel `ne`:
- Wrapper for comparison method `ne`.

Panel `eq`:
- Wrapper for comparison method `eq`.

### 34.5.7 Function application, GroupBy

- `Panel.apply(func[, axis])`: Applies function along axis (or axes) of the Panel.
- `Panel.groupby(function[, axis])`: Group data on given axis, returning GroupBy object.

### 34.5.8 Computations / Descriptive Stats

- `Panel.abs()`: Return an object with absolute value taken–only applicable to objects that are all numeric.
- `Panel.clip([lower, upper, axis, inplace])`: Trim values at input threshold(s).
- `Panel.clip_lower(threshold[, axis, inplace])`: Return copy of the input with values below given value(s) truncated.
- `Panel.clip_upper(threshold[, axis, inplace])`: Return copy of input with values above given value(s) truncated.
- `Panel.count([axis])`: Return number of observations over requested axis.
- `Panel.cummax([axis, skipna])`: Return cumulative max over requested axis.
- `Panel.cummin([axis, skipna])`: Return cumulative minimum over requested axis.
- `Panel.cumprod([axis, skipna])`: Return cumulative product over requested axis.
- `Panel.cumsum([axis, skipna])`: Return cumulative sum over requested axis.
- `Panel.max([axis, skipna, level, numeric_only])`: This method returns the maximum of the values in the object.
- `Panel.mean([axis, skipna, level, numeric_only])`: Return the mean of the values for the requested axis.
- `Panel.median([axis, skipna, level, numeric_only])`: Return the median of the values for the requested axis.
Table 34.81 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Panel.min</strong> ([axis, skipna, level, numeric_only])</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><strong>Panel.pct_change</strong> ([periods, fill_method, ...])</td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td><strong>Panel.prod</strong> ([axis, skipna, level, numeric_only])</td>
<td>Return the product of the values for the requested axis.</td>
</tr>
<tr>
<td><strong>Panel.sem</strong> ([axis, skipna, level, ddof, ...])</td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><strong>Panel.skew</strong> ([axis, skipna, level, numeric_only])</td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><strong>Panel.sum</strong> ([axis, skipna, level, numeric_only])</td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td><strong>Panel.std</strong> ([axis, skipna, level, ddof, ...])</td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><strong>Panel.var</strong> ([axis, skipna, level, ddof, ...])</td>
<td>Return unbiased variance over requested axis.</td>
</tr>
</tbody>
</table>

### 34.5.9 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Panel.add_prefix</strong> (prefix)</td>
<td>Concatenate prefix string with panel items names.</td>
</tr>
<tr>
<td><strong>Panel.add_suffix</strong> (suffix)</td>
<td>Concatenate suffix string with panel items names.</td>
</tr>
<tr>
<td><strong>Panel.drop</strong> ([labels, axis, index, columns, ...])</td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td><strong>Panel.equals</strong> (other)</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td><strong>Panel.filter</strong> ([items, like, regex, axis])</td>
<td>Subset rows or columns of dataframe according to labels in the specified index.</td>
</tr>
<tr>
<td><strong>Panel.first</strong> (offset)</td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><strong>Panel.last</strong> (offset)</td>
<td>Convenience method for subsetting final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><strong>Panel.reindex</strong> (*args, **kwargs)</td>
<td>Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><strong>Panel.reindex_axis</strong> (labels[, axis, method, ...])</td>
<td>Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.</td>
</tr>
<tr>
<td><strong>Panel.reindex_like</strong> (other[, method, copy, ...])</td>
<td>Return an object with matching indices to myself.</td>
</tr>
<tr>
<td><strong>Panel.rename</strong> ([items, major_axis, minor_axis])</td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><strong>Panel.sample</strong> ([n, frac, replace, weights, ...])</td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><strong>Panel.select</strong> (crit[, axis])</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td><strong>Panel.take</strong> (indices[, axis, convert, is_copy])</td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><strong>Panel.truncate</strong> ([before, after, axis, copy])</td>
<td>Truncates a sorted DataFrame/Series before and/or after some particular index value.</td>
</tr>
</tbody>
</table>

### 34.5.10 Missing data handling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Panel.dropna</strong> ([axis, how, inplace])</td>
<td>Drop 2D from panel, holding passed axis constant</td>
</tr>
<tr>
<td><strong>Panel.fillna</strong> ([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
</tbody>
</table>

### 34.5.11 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Panel.sort_index</strong> ([axis, level, ascending, ...])</td>
<td>Sort object by labels (along an axis)</td>
</tr>
</tbody>
</table>

Continued on next page
### 34.5.12 Combining / joining / merging

- **Panel.join**(other[, how, lslug, rsuffix])
  - Join items with other Panel either on major and minor axes
- **Panel.update**(other[, join, overwrite, ...])
  - Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.

### 34.5.13 Time series-related

- **Panel.asfreq**(freq[, method, how, normalize, ...])
  - Convert TimeSeries to specified frequency.
- **Panel.shift**([periods, freq, axis])
  - Shift index by desired number of periods with an optional time freq.
- **Panel.resample**(rule[, how, axis, ...])
  - Convenience method for frequency conversion and resampling of time series.
- **Panel.tz_convert**(tz[, axis, level, copy])
  - Convert tz-aware axis to target time zone.
- **Panel.tz_localize**(tz[, axis, level, copy])
  - Localize tz-naive TimeSeries to target time zone.

### 34.5.14 Serialization / IO / Conversion

- **Panel.from_dict**(data[, intersect, orient, dtype])
  - Construct Panel from dict of DataFrame objects
- **Panel.to_pickle**(path[, compression, protocol])
  - Pickle (serialize) object to input file path.
- **Panel.to_excel**(path[, na_rep, engine])
  - Write each DataFrame in Panel to a separate excel sheet
- **Panel.to_hdf**(path_or_buf, key, **kwargs)
  - Write the contained data to an HDF5 file using HDFStore.
- **Panel.to_sparse**(*args, **kwargs)
  - NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
- **Panel.to_frame**([filter_observations])
  - Transform wide format into long (stacked) format as DataFrame whose columns are the Panel’s items and whose index is a MultiIndex formed of the Panel’s major and minor axes.
- **Panel.to_xarray()**
  - Return an xarray object from the pandas object.
- **Panel.to_clipboard**([excel, sep])
  - Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

### 34.6 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.
34.6.1 pandas.Index

class pandas.Index

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects

Parameters:
- **data**: array-like (1-dimensional)
  - **dtype**: NumPy dtype (default: object)
  - **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,  TimedeltaIndex,  PeriodIndex,  Int64Index,  UInt64Index,  
        Float64Index
```

Notes

An Index instance can only contain hashable objects

Examples

```python
>>> pd.Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')

>>> 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>asIs</code></td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td><code>base</code></td>
<td>return the data pointer of the underlying data</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.89 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype</td>
<td>data type</td>
</tr>
<tr>
<td>dtype_str</td>
<td>string representation of the data type</td>
</tr>
<tr>
<td>empty</td>
<td>True if the index is empty</td>
</tr>
<tr>
<td>flags</td>
<td>store the bit flags</td>
</tr>
<tr>
<td>has_duplicates</td>
<td>True if there are any duplicate values</td>
</tr>
<tr>
<td>hasnans</td>
<td>True if there are any NaN values</td>
</tr>
<tr>
<td>inferred_type</td>
<td>inferred type of the underlying data</td>
</tr>
<tr>
<td>is_all_dates</td>
<td>True if the index contains only dates</td>
</tr>
<tr>
<td>is_monotonic</td>
<td>True if the index is monotonically increasing or decreasing</td>
</tr>
<tr>
<td>is_monotonic_decreasing</td>
<td>return if the index is monotonically decreasing (only equal or increasing)</td>
</tr>
<tr>
<td>is_monotonic_increasing</td>
<td>return if the index is monotonically increasing (only equal or decreasing)</td>
</tr>
<tr>
<td>is_unique</td>
<td>True if all values in the index are unique</td>
</tr>
<tr>
<td>itemsize</td>
<td>return the size of the item of the underlying data</td>
</tr>
<tr>
<td>name</td>
<td>name of the index</td>
</tr>
<tr>
<td>names</td>
<td>list of names in the index</td>
</tr>
<tr>
<td>nbytes</td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td>ndim</td>
<td>return the number of dimensions of the underlying data</td>
</tr>
<tr>
<td>shape</td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td>size</td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td>strides</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>values</td>
<td>return the underlying data as an ndarray</td>
</tr>
</tbody>
</table>

34.6.1.1 pandas.Index.T

Index.T

    return the transpose, which is by definition self

34.6.1.2 pandas.Index.asi8

Index.asi8 = None

34.6.1.3 pandas.Index.base

Index.base

    return the base object if the memory of the underlying data is shared

34.6.1.4 pandas.Index.data

Index.data

    return the data pointer of the underlying data

34.6.1.5 pandas.Index.dtype

Index.dtype = None
34.6.1.6 pandas.Index.dtype_str
Index.dtype_str = None

34.6.1.7 pandas.Index.empty
Index.empty

34.6.1.8 pandas.Index.flags
Index.flags

34.6.1.9 pandas.Index.has_duplicates
Index.has_duplicates

34.6.1.10 pandas.Index.hasnans
Index.hasnans = None

34.6.1.11 pandas.Index.inferred_type
Index.inferred_type = None

34.6.1.12 pandas.Index.is_all_dates
Index.is_all_dates = None

34.6.1.13 pandas.Index.is_monotonic
Index.is_monotonic
    alias for is_monotonic_increasing (deprecated)

34.6.1.14 pandas.Index.is_monotonic_decreasing
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
```
34.6.1.15  pandas.Index.is_monotonic_increasing

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
```

34.6.1.16  pandas.Index.is_unique

Index.is_unique = None

34.6.1.17  pandas.Index.itemsize

Index.itemsize
return the size of the dtype of the item of the underlying data

34.6.1.18  pandas.Index.name

Index.name = None

34.6.1.19  pandas.Index.names

Index.names

34.6.1.20  pandas.Index.nbytes

Index.nbytes
return the number of bytes in the underlying data

34.6.1.21  pandas.Index.ndim

Index.ndim
return the number of dimensions of the underlying data, by definition 1

34.6.1.22  pandas.Index.nlevels

Index.nlevels
34.6.1.23 pandas.Index.shape

Index.shape
return a tuple of the shape of the underlying data

34.6.1.24 pandas.Index.size

Index.size
return the number of elements in the underlying data

34.6.1.25 pandas.Index.strides

Index.strides
return the strides of the underlying data

34.6.1.26 pandas.Index.values

Index.values
return the underlying data as an ndarray

Methods

*all*(*args, **kwargs) Return whether all elements are True

*any*(*args, **kwargs) Return whether any element is True

*append*(other) Append a collection of Index options together

*argmax*[axis] return a ndarray of the maximum argument indexer

*argmin*[axis] return a ndarray of the minimum argument indexer

*argsort*(*args, **kwargs) Returns the indices that would sort the index and its underlying data.

*asof*(label) For a sorted index, return the most recent label up to and including the passed label.

*asof_locs*(where, mask) where : array of timestamps

*astype*(dtype[, copy]) Create an Index with values cast to dtypes.

*contains*(key) return a boolean if this key is IN the index

*copy*[name, deep, dtype]) Make a copy of this object.

*delete*(loc) Make new Index with passed location(-s) deleted

*difference*(other) Return a new Index with elements from the index that are not in other.

*drop*[labels[, errors]] Make new Index with passed list of labels deleted

*drop_duplicates*[keep] Return Index with duplicate values removed

*dropna*[how] Return Index without NA/NaN values

*duplicated*[keep] Return boolean np.ndarray denoting duplicate values

*equals*(other) Determines if two Index objects contain the same elements.

*factorize*[isort, na_sentinel]) Encode the object as an enumerated type or categorical variable

*fillna*[value, downcast]) Fill NA/NaN values with the specified value

*format*[isname, formatter]) Render a string representation of the Index

Continued on next page
Table 34.90 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_duplicates()</code></td>
<td></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, **kwargs)</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, equal to the length of the index.</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>get_values()</code></td>
<td>return the underlying data as an 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>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td><code>identical(other)</code></td>
<td>Make new Index inserting new item at location.</td>
</tr>
<tr>
<td><code>insert(loc, item)</code></td>
<td>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td><code>intersection(other)</code></td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td><code>is_(other)</code></td>
<td></td>
</tr>
<tr>
<td><code>is_boolean()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_categorical()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_float(bool)</code></td>
<td></td>
</tr>
<tr>
<td><code>is_integer()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_interval()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_lexsorted_for_tuple(tup)</code></td>
<td></td>
</tr>
<tr>
<td><code>is_mixed()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_numeric()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_object()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_type_compatible(kind)</code></td>
<td></td>
</tr>
<tr>
<td><code>isin(values[, level])</code></td>
<td>Compute boolean array of whether each index value is found in the passed set of values.</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</td>
</tr>
<tr>
<td><code>join(other[, how, level, return_indexers, sort])</code></td>
<td>this is an internal non-public method</td>
</tr>
<tr>
<td><code>map(mapper)</code></td>
<td>Apply mapper function to an index.</td>
</tr>
<tr>
<td><code>max()</code></td>
<td>The maximum value of the object</td>
</tr>
<tr>
<td><code>memory_usage([deep])</code></td>
<td>Memory usage of my values</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>The minimum value of the object</td>
</tr>
<tr>
<td><code>notna()</code></td>
<td>Inverse of isna</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Inverse of isna</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>
<tr>
<td><code>reindex(target[, method, level, limit, ...])</code></td>
<td>Create index with target’s values (move/add/delete values as necessary)</td>
</tr>
</tbody>
</table>
Table 34.90 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename(name[, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>repeat(repeats, *args, **kwargs)</td>
<td>Repeat elements of an Index.</td>
</tr>
<tr>
<td>reshape(*args, **kwargs)</td>
<td>NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.</td>
</tr>
<tr>
<td>searchsorted(value[, side, sorter])</td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td>set_names(names[, level, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>set_value(arr, key, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>shift([periods, freq])</td>
<td>Shift Index containing datetime objects by input number of periods and</td>
</tr>
<tr>
<td>slice_indexer([start, end, step, kind])</td>
<td>For an ordered Index, compute the slice indexer for input labels and</td>
</tr>
<tr>
<td>slice_locs([start, end, step, kind])</td>
<td>Compute slice locations for input labels.</td>
</tr>
<tr>
<td>sort(*args, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>sort_values([return_indexer, ascending])</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>sortlevel([level, ascending, sort_remaining])</td>
<td>For internal compatibility with with the Index API</td>
</tr>
<tr>
<td>str</td>
<td>alias of StringMethods</td>
</tr>
<tr>
<td>summary([name])</td>
<td></td>
</tr>
<tr>
<td>symmetric_difference(other[, result_name])</td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
<tr>
<td>take(indices[, axis, allow_fill, fill_value])</td>
<td>return a new Index of the values selected by the indices</td>
</tr>
<tr>
<td>to_datetime([dayfirst])</td>
<td>DEPRECATED: use pandas.to_datetime() instead.</td>
</tr>
<tr>
<td>to_frame(index)</td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td>to_native_types([slicer])</td>
<td>Format specified values of self and return them.</td>
</tr>
<tr>
<td>to_series(**kwargs)</td>
<td>Create a Series with both index and values equal to the index keys</td>
</tr>
<tr>
<td>tolist()</td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td>transpose(*args, **kwargs)</td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td>union(other)</td>
<td>Form the union of two Index objects and sorts if possible.</td>
</tr>
<tr>
<td>unique()</td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td>value_counts([normalize, sort, ascending, ...])</td>
<td>Returns object containing counts of unique values.</td>
</tr>
<tr>
<td>view([cls])</td>
<td></td>
</tr>
<tr>
<td>where(cond[, other])</td>
<td>New in version 0.19.0.</td>
</tr>
</tbody>
</table>

### 34.6.1.27 pandas.Index.all

Index.all(*args, **kwargs)

Return whether all elements are True

**Parameters**

All arguments to numpy.all are accepted.

**Returns**

- **all**: bool or array_like (if axis is specified)

A single element array_like may be converted to bool.

### 34.6.1.28 pandas.Index.any

Index.any(*args, **kwargs)

Return whether any element is True
Parameters  All arguments to numpy.any are accepted.

Returns  any : bool or array_like (if axis is specified)

A single element array_like may be converted to bool.

34.6.1.29  pandas.Index.append

Index.append(other)

Append a collection of Index options together

Parameters  other : Index or list/tuple of indices

Returns  appended : Index

34.6.1.30  pandas.Index.argmax

Index.argmax(axis=None)

return a ndarray of the maximum argument indexer

See also:
numpy.ndarray.argmax

34.6.1.31  pandas.Index.argmin

Index.argmin(axis=None)

return a ndarray of the minimum argument indexer

See also:
numpy.ndarray.argmin

34.6.1.32  pandas.Index.argsort

Index.argsort(*args, **kwargs)

Returns the indices that would sort the index and its underlying data.

Returns  argsorted : numpy array

See also:
numpy.ndarray.argsort

34.6.1.33  pandas.Index.asof

Index.asof(label)

For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.

See also:

get_loc  asof is a thin wrapper around get_loc with method=’pad’
34.6.1.34 pandas.Index.asof_locs

Index.asof_locs(where, mask)
where: array of timestamps
mask: array of booleans where data is not NA

34.6.1.35 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 ValueError exception is raised.

Parameters
dtype: numpy dtype or pandas type
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.

New in version 0.19.0.

34.6.1.36 pandas.Index.contains

Index.contains(key)
return a boolean if this key is IN the index

Parameters
key: object

Returns
boolean

34.6.1.37 pandas.Index.copy

Index.copy(name=None, deep=False, dtype=None, **kwargs)
Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters
name: string, optional
deep: boolean, default False
dtype: numpy dtype or pandas type

Returns
copy: Index

Notes
In most cases, there should be no functional difference from using deep, but if deep is passed it will attempt to deepcopy.

34.6.1.38 pandas.Index.delete

Index.delete(loc)
Make new Index with passed location(-s) deleted

Returns
new_index: Index
34.6.1.39 pandas.Index.difference

Index.difference(other)
Return a new Index with elements from the index that are not in other.
This is the set difference of two Index objects. It’s sorted if sorting is possible.

Parameters other : Index or array-like

Returns difference : Index

Examples

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
```

34.6.1.40 pandas.Index.drop

Index.drop(labels, errors='raise')
Make new Index with passed list of labels deleted

Parameters labels : array-like
errors : {'ignore', 'raise'}, default 'raise'
If ‘ignore’, suppress error and existing labels are dropped.

Returns dropped : Index

34.6.1.41 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

34.6.1.42 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 valid : Index
34.6.1.43 pandas.Index.duplicated

Index.duplicated(keep='first')
Return boolean np.ndarray denoting duplicate values

Parameters keep: {'first', 'last', False}, default 'first'

- 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 duplicated: np.ndarray

34.6.1.44 pandas.Index.equals

Index.equals(other)
Determines if two Index objects contain the same elements.

34.6.1.45 pandas.Index.factorize

Index.factorize(sort=False, na_sentinel=-1)
Encode the object as an enumerated type or categorical variable

Parameters sort: boolean, default False
Sort by values

na_sentinel: int, default -1
Value to mark “not found”

Returns labels: the indexer to the original array
uniques: the unique Index

34.6.1.46 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 filled: %(klass)s

34.6.1.47 pandas.Index.format

Index.format(name=False, formatter=None, **kwargs)
Render a string representation of the Index
### pandas.Index.get_duplicates

```
Index.get_duplicates()
```

### pandas.Index.get_indexer

```
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
    - 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 most 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.
    - New in version 0.17.0.
    - New in version 0.21.0: (list-like tolerance)

**Returns**

- `indexer` : ndarray of int
  - 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.

**Examples**

```
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

### pandas.Index.get_indexer_for

```
Index.get_indexer_for(target, **kwargs)
```

guaranteed return of an indexer even when non-unique This dispatches to get_indexer or get_indexer_nonunique as appropriate
34.6.1.51 pandas.Index.get_indexer_non_unique

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 : ndarray of int
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 : ndarray of int
An indexer into the target of the values not found. These correspond to the -1 in the indexer array

34.6.1.52 pandas.Index.get_level_values

Index.get_level_values(level)
Return an Index of values for requested level, 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
self, as there is only one level in the Index.

See also:

pandas.MultiIndex.get_level_values get values for a level of a MultiIndex

34.6.1.53 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 : optional
Maximum distance from index value for inexact matches. The value of the index at the matching location most satisfy the equation abs(index[loc] - key) <= 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.

New in version 0.17.0.

New in version 0.21.0: (list-like 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], dtype=bool)
```

#### 34.6.1.54 pandas.Index.get_slice_bound

**Index.get_slice_bound** *(label, side, kind)*

Calculate slice bound that corresponds to given label.

Returns leftmost (one-past-the-rightmost if `side` is `'right'`) position of given label.

**Parameters**

- `label` : object
- `side` : {'left', 'right'}
- `kind` : {'ix', 'loc', 'getitem'}

#### 34.6.1.55 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

#### 34.6.1.56 pandas.Index.get_values

**Index.get_values** ()

return the underlying data as an ndarray

#### 34.6.1.57 pandas.Index.groupby

**Index.groupby** *(values)*

Group the index labels by a given array of values.

**Parameters**

- `values` : array
  
  Values used to determine the groups.
Returns groups : dict
{group name -> group labels}

34.6.1.58 pandas.Index.holds_integer

Index.holds_integer()

34.6.1.59 pandas.Index.identical

Index.identical(other)
Similar to equals, but check that other comparable attributes are also equal

34.6.1.60 pandas.Index.insert

Index.insert(loc, item)
Make new Index inserting new item at location. Follows Python list.append semantics for negative values

Parameters loc : int
item : object

Returns new_index : Index

34.6.1.61 pandas.Index.intersection

Index.intersection(other)
Form the intersection of two Index objects.
This returns a new Index with elements common to the index and other, preserving the order of the calling index.

Parameters other : Index or array-like

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')
```

34.6.1.62 pandas.Index.is_

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 True if both have same underlying data, False otherwise : bool
34.6.1.63 pandas.Index.is_boolean
Index.is_boolean()

34.6.1.64 pandas.Index.is_categorical
Index.is_categorical()

34.6.1.65 pandas.Index.is_floating
Index.is_floating()

34.6.1.66 pandas.Index.is_integer
Index.is_integer()

34.6.1.67 pandas.Index.is_interval
Index.is_interval()

34.6.1.68 pandas.Index.is_lexsorted_for_tuple
Index.is_lexsorted_for_tuple(tup)

34.6.1.69 pandas.Index.is_mixed
Index.is_mixed()

34.6.1.70 pandas.Index.is_numeric
Index.is_numeric()

34.6.1.71 pandas.Index.is_object
Index.is_object()

34.6.1.72 pandas.Index.is_type_compatible
Index.is_type_compatible(kind)
34.6.1.73 pandas.Index.isin

Index.isin(values, level=None)
Compute boolean array of whether each index value is found in the passed set of values.

Parameters:

values : set or list-like
Sought values.
New in version 0.18.1.
Support for values as a set

level : str or int, optional
Name or position of the index level to use (if the index is a MultiIndex).

Returns:

is_contained : ndarray (boolean dtype)

Notes

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.

34.6.1.74 pandas.Index.isna

Index.isna()
Detect missing values
New in version 0.20.0.

Returns:
a boolean array of whether my values are NA

See also:

isnull alias of isna
pandas.isna top-level isna

34.6.1.75 pandas.Index.isnull

Index.isnull()
Detect missing values
New in version 0.20.0.

Returns:
a boolean array of whether my values are NA

See also:

isnull alias of isna
pandas.isna top-level isna
34.6.1.76 pandas.Index.item

Index.item()
return the first element of the underlying data as a python scalar

34.6.1.77 pandas.Index.join

Index.join(other, how='left', level=None, return_indexers=False, sort=False)
this is an internal non-public method
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 : boolean, default False
    sort : boolean, 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)
    New in version 0.20.0.

Returns join_index, (left_indexer, right_indexer)

34.6.1.78 pandas.Index.map

Index.map(mapper)
Apply mapper function to an index.

Parameters mapper : callable
Function to be applied.

Returns applied : Union[Index, MultiIndex], inferred
The output of the mapping function applied to the index. If the function returns a
tuple with more than one element a MultiIndex will be returned.

34.6.1.79 pandas.Index.max

Index.max()
The maximum value of the object

34.6.1.80 pandas.Index.memory_usage

Index.memory_usage(deep=False)
Memory usage of my values

Parameters deep : bool
    Introspect the data deeply, interrogate object dtypes for system-level memory
    consumption

Returns bytes used
See also:

numpy.ndarray.nbytes

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

34.6.1.81 pandas.Index.min

Index.min()  
The minimum value of the object

34.6.1.82 pandas.Index.notna

Index.notna()  
Inverse of isna  
New in version 0.20.0.  
Returns a boolean array of whether my values are not NA  
See also:

notnull alias of notna  
pandas.notna top-level notna

34.6.1.83 pandas.Index.notnull

Index.notnull()  
Inverse of isna  
New in version 0.20.0.  
Returns a boolean array of whether my values are not NA  
See also:

notnull alias of notna  
pandas.notna top-level notna

34.6.1.84 pandas.Index.nunique

Index.nunique(dropna=True)  
Return number of unique elements in the object.  
Excludes NA values by default.  
Parameters dropna : boolean, default True  
Don’t include NaN in the count.  
Returns nunique : int
34.6.1.85 pandas.Index.putmask

Index.putmask(mask, value)
return a new Index of the values set with the mask

See also:
numpy.ndarray.putmask

34.6.1.86 pandas.Index.ravel

Index.ravel(order='C')
return an ndarray of the flattened values of the underlying data

See also:
numpy.ndarray.ravel

34.6.1.87 pandas.Index.reindex

Index.reindex(target, method=None, level=None, limit=None, tolerance=None)
Create index with target’s values (move/add/delete values as necessary)

Parameters target : an iterable

Returns new_index : pd.Index

Resulting index

indexer : np.ndarray or None

Indices of output values in original index

34.6.1.88 pandas.Index.rename

Index.rename(name, inplace=False)
Set new names on index. Defaults to returning new index.

Parameters name : str or list

name to set

inplace : bool

if True, mutates in place

Returns new index (of same type and class...etc) [if inplace, returns None]

34.6.1.89 pandas.Index.repeat

Index.repeat(repeats, *args, **kwargs)
Repeat elements of an Index. Refer to numpy.ndarray.repeat for more information about the repeats argument.

See also:
numpy.ndarray.repeat
34.6.1.90 pandas.Index.reshape

Index.reshape(*args, **kwargs)

NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.

Reshape an Index.

34.6.1.91 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 IndexOpsMixin self such that, if the corresponding elements in value were inserted before the indices, the order of self would be preserved.

Parameters

- value : array_like
  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

- indices : array of ints
  Array of insertion points with the same shape as value.

See also:

numpy.searchsorted

Notes

Binary search is used to find the required insertion points.

Examples

```python
>>> x = pd.Series([1, 2, 3])
>>> x
0  1  
1  2  
2  3  
dtype: int64

>>> x.searchsorted(4)
array([3])

>>> x.searchsorted([0, 4])
array([0, 3])
```
```python
>>> x.searchsorted([1, 3], side='left')
array([0, 2])

>>> x.searchsorted([1, 3], side='right')
array([1, 3])

>>> x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'])
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]

>>> x.searchsorted('bread')
array([1]) # Note: an array, not a scalar

>>> x.searchsorted(['bread'])
array([1])

>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])

>>> x.searchsorted(['bread', 'eggs'], side='right')
array([3, 4]) # eggs before milk
```

### 34.6.1.92 pandas.Index.set_names

Index.set_names(names, level=None, inplace=False)
Set new names on index. Defaults to returning new index.

**Parameters**
- **names**: str or sequence
  - name(s) to set
- **level**: int, level name, or sequence of int/level names (default None)
  - If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels).
  - Otherwise level must be None
- **inplace**: bool
  - if True, mutates in place

**Returns**
- new index (of same type and class...etc) [if inplace, returns None]

**Examples**

```python
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64')

>>> Index([1, 2, 3, 4]).set_names(['foo'])
Int64Index([1, 2, 3, 4], dtype='int64')

>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')],
                               names=['foo', 'bar'])

>>> idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
```
34.6.1.93 pandas.Index.set_value

Index.set_value(arr, key, value)
Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

34.6.1.94 pandas.Index.shift

Index.shift(periods=1, freq=None)
Shift Index containing datetime objects by input number of periods and DateOffset

Returns shifted : Index

34.6.1.95 pandas.Index.slice_indexer

Index.slice_indexer(start=None, end=None, step=None, kind=None)
For an ordered Index, compute the slice indexer for input labels and step

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 : string, default None

Returns indexer : ndarray or slice

Notes
This function assumes that the data is sorted, so use at your own peril

34.6.1.96 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, default None
If None, defaults to 1

kind : {'ix', 'loc', 'getitem'} or None

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

```py
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

34.6.1.97 pandas.Index.sort

Index.sort(*args, **kwargs)

34.6.1.98 pandas.Index.sort_values

Index.sort_values(return_indexer=False, ascending=True)

Return sorted copy of Index

34.6.1.99 pandas.Index.sortlevel

Index.sortlevel(level=None, ascending=True, sort_remaining=None)

For internal compatibility with with the Index API

Sort the Index. This is for compat with MultiIndex

Parameters ascending : boolean, default True

False to sort in descending order

level, sort_remaining are compat parameters

Returns sorted_index : Index

34.6.1.100 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.str.split('_')
>>> s.str.replace('_', '')
```

34.6.1.101 pandas.Index.summary

```python
Index.summary(name=None)
```

34.6.1.102 pandas.Index.symmetric_difference

```python
Index.symmetric_difference(other, result_name=None)
```

Compute the symmetric difference of two Index objects. It’s sorted if sorting is possible.

**Parameters**

- **other**: Index or array-like
- **result_name**: str

**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 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

34.6.1.103 pandas.Index.take

```python
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**: list
  - Indices to be taken
- **axis**: int, optional
  - The axis over which to select values, always 0.
allow_fill : bool, default True
fill_value : bool, default None

If allow_fill=True and fill_value is not None, indices specified by -1 is regarded as NA. If Index doesn’t hold NA, raise ValueError

See also:
numpy.ndarray.take

34.6.1.104 pandas.Index.to_datetime

Index.to_datetime(dayfirst=False)
DEPRECATED: use pandas.to_datetime() instead.

For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex

34.6.1.105 pandas.Index.to_frame

Index.to_frame(index=True)
Create a DataFrame with a column containing the Index.
New in version 0.21.0.

Parameters index : boolean, default True
Set the index of the returned DataFrame as the original Index.

Returns DataFrame : a DataFrame containing the original Index data.

34.6.1.106 pandas.Index.to_native_types

Index.to_native_types(slicer=None, **kwargs)
Format specified values of self and return them.

Parameters slicer : int, array-like
An indexer into self that specifies which values are used in the formatting process.

kwags : 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

34.6.1.107 pandas.Index.to_series

Index.to_series(**kwargs)
Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index

Returns Series : dtype will be based on the type of the Index values.
### 34.6.1.108 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)

See also:

numpy.ndarray.tolist

### 34.6.1.109 pandas.Index.transpose

**Index.transpose(*args, **kwargs)**

return the transpose, which is by definition self

### 34.6.1.110 pandas.Index.union

**Index.union(other)**

Form the union of two Index objects and sorts if possible.

**Parameters**

- **other**: Index or array-like

**Returns**

- **union**: Index

**Examples**

```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')
```

### 34.6.1.111 pandas.Index.unique

**Index.unique()**

Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

**Parameters**

- **values**: 1d array-like

**Returns**

- unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

See also:

unique, Index.unique, Series.unique
34.6.1.112 pandas.Index.value_counts

Index.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

Returns object 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**: boolean, default False

If True then the object returned will contain the relative frequencies of the unique
values.

**sort**: boolean, default True

Sort by values

**ascending**: boolean, default False

Sort in ascending order

**bins**: integer, optional

Rather than count values, group them into half-open bins, a convenience for
pd.cut, only works with numeric data

**dropna**: boolean, default True

Don’t include counts of NaN.

Returns

**counts**: Series

34.6.1.113 pandas.Index.view

Index.view(cls=None)

34.6.1.114 pandas.Index.where

Index.where(cond, other=None)

New in version 0.19.0.

Return an Index of same shape as self and whose corresponding entries are from self where cond is True
and otherwise are from other.

Parameters

**cond**: boolean array-like with the same length as self

**other**: scalar, or array-like

34.6.2 Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.values</td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td>Index.is_monotonic</td>
<td>alias for is_monotonic_increasing (deprecated)</td>
</tr>
<tr>
<td>Index.is_monotonic_increasing</td>
<td>return if the index is monotonic increasing (only equal or</td>
</tr>
<tr>
<td>Index.is_monotonic_decreasing</td>
<td>return if the index is monotonic decreasing (only equal or</td>
</tr>
<tr>
<td>Index.is_unique</td>
<td></td>
</tr>
<tr>
<td>Index.has_duplicates</td>
<td></td>
</tr>
<tr>
<td>Index.dtype</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Index.inferred_type
Index.is_all_dates
Index.shape
Index nbytes
Index.ndim
Index.size
Index.empty
Index.strides
Index.itemsize
Index.base
Index.T
Index.memory_usage(deep)

<table>
<thead>
<tr>
<th>Table 34.91 – continued from previous page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index.inferred_type</strong></td>
</tr>
<tr>
<td><strong>Index.is_all_dates</strong></td>
</tr>
<tr>
<td><strong>Index.shape</strong></td>
</tr>
<tr>
<td><strong>Index nbytes</strong></td>
</tr>
<tr>
<td><strong>Index.ndim</strong></td>
</tr>
<tr>
<td><strong>Index.size</strong></td>
</tr>
<tr>
<td><strong>Index.empty</strong></td>
</tr>
<tr>
<td><strong>Index.itemsize</strong></td>
</tr>
<tr>
<td><strong>Index.base</strong></td>
</tr>
<tr>
<td><strong>Index T</strong></td>
</tr>
<tr>
<td><strong>Index.memory_usage(deep)</strong></td>
</tr>
</tbody>
</table>

34.6.3 Modifying and Computations

| **Index. all(**args, **kwargs)** | Return whether all elements are True |
| **Index. any(**args, **kwargs)** | Return whether any element is True |
| **Index.argmin([axis])** | Return an ndarray of the minimum argument indexer |
| **Index.argmax([axis])** | Return an ndarray of the maximum argument indexer |
| **Index. copy([name, deep, dtype])** | 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])** | Return a boolean np.ndarray denoting duplicate values |
| **Index.equals(other)** | Determines if two Index objects contain the same elements |
| **Index.factorize([sort, na_sentinel])** | Encode the object as an enumerated type or categorical variable |
| **Index.identical(other)** | Similar to equals, but check that other comparable attributes are |
| **Index.insert(loc, item)** | Make new Index inserting new item at location |
| **Index. max()** | The maximum value of the object |
| **Index. min()** | The minimum value of the object |
| **Index. reindex(target[, method, level, ...])** | Create index with target’s values (move/add/delete values as necessary) |
| **Index. repeat(repeats, *args, **kwargs)** | Repeat elements of an Index |
| **Index. where(cond[, other])** | New in version 0.19.0 |
| **Index. take(indices[, axis, allow_fill, ...])** | Return a new Index of the values selected by the indices |
| **Index.putmask(mask, value)** | Return a new Index of the values set with the mask |
| **Index.set_names(names[, level, inplace])** | Set new names on index |
| **Index.unique()** | Return unique values in the object |
| **Index. nunique([dropna])** | Return number of unique elements in the object |
| **Index.value_counts([normalize, sort, ...])** | Returns object containing counts of unique values |

34.6.4 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>Inverse of isna</td>
</tr>
</tbody>
</table>

### 34.6.5 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.astype()</code></td>
<td>Create an Index with values cast to dtypes.</td>
</tr>
<tr>
<td><code>Index.tolist()</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>Index.to_datetime()</code></td>
<td>DEPRECATED: use <code>pandas.to_datetime()</code> instead.</td>
</tr>
<tr>
<td><code>Index.to_series(**kwargs)</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])</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
</tbody>
</table>

### 34.6.6 Sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.argsort(*args, **kwargs)</code></td>
<td>Returns the indices that would sort the index and its underlying data.</td>
</tr>
<tr>
<td><code>Index.sort_values([return_indexer, ascending])</code></td>
<td>Return sorted copy of Index</td>
</tr>
</tbody>
</table>

### 34.6.7 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 containing datetime objects by input number of periods and</td>
</tr>
</tbody>
</table>

### 34.6.8 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>this is an internal non-public method</td>
</tr>
<tr>
<td><code>Index.intersection(other)</code></td>
<td>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td><code>Index.union(other)</code></td>
<td>Form the union of two Index objects and sorts if possible.</td>
</tr>
<tr>
<td><code>Index.difference(other)</code></td>
<td>Return a new Index with elements from the index that are not in other.</td>
</tr>
<tr>
<td><code>Index.symmetric_difference(other[, result_name])</code></td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
</tbody>
</table>

### 34.6.9 Selecting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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_non_unique(target)</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
</tbody>
</table>

---

Continued on next page
Table 34.98 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.get_level_values(level)</code></td>
<td>Return an Index of values for requested level, equal to the length of the index.</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_value(series, key)</code></td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>Index.isin(values[, level])</code></td>
<td>Compute boolean array of whether each index value is found in the passed set of values.</td>
</tr>
<tr>
<td><code>Index.slice_indexer([start, end, step, kind])</code></td>
<td>For an ordered Index, compute the slice indexer for input labels and</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>

34.7 Numeric Index

**RangeIndex**

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), or other RangeIndex instance.
  - If int and “stop” is not given, interpreted as “stop” instead.
- `stop` : int (default: 0)
- `step` : int (default: 1)
- `name` : object, optional
  - Name to be stored in the index
- `copy` : bool, default False
  - Unused, accepted for homogeneity with other index types.

**See also:**

- `Index` The base pandas Index type
- `Int64Index` Index of int64 data

34.7.2 pandas.Int64Index

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Int64Index is a special case of Index with purely integer labels.
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

**Notes**

An Index instance can **only** contain hashable objects.

### 34.7.3 pandas.UInt64Index

#### class `pandas.UInt64Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. UInt64Index is a special case of *Index* with purely unsigned integer labels.

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

**Notes**

An Index instance can **only** contain hashable objects.

### 34.7.4 pandas.Float64Index

#### class `pandas.Float64Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Float64Index is a special case of *Index* with purely float labels.

Parameters **data** : array-like (1-dimensional)

- **dtype** : NumPy dtype (default: float64)
- **copy** : bool
  Make a copy of input ndarray
```python
class pandas.CategoricalIndex
   Immutable Index implementing an ordered, sliceable set. CategoricalIndex represents a sparsely populated Index with an underlying Categorical.

   Parameters:
   data : array-like or Categorical, (1-dimensional)
      categories : optional, array-like
         categories for the CategoricalIndex
   ordered : boolean,
      designating if the categories are ordered
   copy : bool
      Make a copy of input ndarray
   name : object
      Name to be stored in the index

See also:
   Categorical, Index
```

### 34.8.2 Categorical Components

- `CategoricalIndex.codes`  
- `CategoricalIndex.categories`  
- `CategoricalIndex.ordered`  
- `CategoricalIndex.rename_categories(*args, ...)`: Renames categories.  
- `CategoricalIndex.reorder_categories(*args, ...)`: Reorders categories as specified in new_categories.  

Continued on next page
Table 34.101 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CategoricalIndex.add_categories(*args, **kwargs)</code></td>
<td>Add new categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.remove_categories(*args, ...)</code></td>
<td>Removes the specified categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.remove_unused_categories(...)</code></td>
<td>Removes categories which are not used.</td>
</tr>
<tr>
<td><code>CategoricalIndex.set_categories(*args, **kwargs)</code></td>
<td>Sets the categories to the specified new categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.as_ordered(*args, **kwargs)</code></td>
<td>Sets the Categorical to be ordered.</td>
</tr>
<tr>
<td><code>CategoricalIndex.as_unordered(*args, **kwargs)</code></td>
<td>Sets the Categorical to be unordered.</td>
</tr>
</tbody>
</table>

34.8.2.1 pandas.CategoricalIndex.codes

CategoricalIndex.codes

34.8.2.2 pandas.CategoricalIndex.categories

CategoricalIndex.categories

34.8.2.3 pandas.CategoricalIndex.ordered

CategoricalIndex.ordered

34.8.2.4 pandas.CategoricalIndex.rename_categories

CategoricalIndex.rename_categories(*args, **kwargs)

Renames categories.

Parameters

- `new_categories`: list-like or dict-like
  - 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. New in version 0.21.0.

Warning: Currently, Series are considered list like. In a future version of pandas they’ll be considered dict-like.

- `inplace`: boolean (default: False)
  Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns

- `cat`: Categorical or None
  With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

Raises

- ValueError

34.8. CategoricalIndex
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, add_categories, remove_categories,
remove_unused_categories, set_categories

Examples

```python
>>> c = 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]
```

34.8.2.5 pandas.CategoricalIndex.reorder_categories

CategoricalIndex.reorder_categories(*args, **kwargs)
Reorders 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 : boolean, optional
Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

inplace : boolean (default: False)
Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns cat : Categorical with reordered categories or None if inplace.

Raises ValueError
If the new categories do not contain all old category items or any new ones

See also:
rename_categories, add_categories, remove_categories,
remove_unused_categories, set_categories

34.8.2.6 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.
- **inplace**: boolean (default: False)
  - Whether or not to add the categories inplace or return a copy of this categorical with added categories.

**Returns**

- **cat**: Categorical with new categories added or None if inplace.

**Raises**

- **ValueError**: If the new categories include old categories or do not validate as categories

See also:

- rename_categories, reorder_categories, remove_categories, remove_unused_categories, set_categories

### 34.8.2.7 pandas.CategoricalIndex.remove_categories

CategoricalIndex.remove_categories(*args, **kwargs)

Removes the specified categories.

- **removals**: category or list of categories
  - The categories which should be removed.
- **inplace**: boolean (default: False)
  - Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

**Returns**

- **cat**: Categorical with removed categories or None if inplace.

**Raises**

- **ValueError**: If the removals are not contained in the categories

See also:

- rename_categories, reorder_categories, add_categories, remove_unused_categories, set_categories

### 34.8.2.8 pandas.CategoricalIndex.remove_unused_categories

CategoricalIndex.remove_unused_categories(*args, **kwargs)

Removes categories which are not used.

- **inplace**: boolean (default: False)
  - Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

**Returns**

- **cat**: Categorical with unused categories dropped or None if inplace.
See also:

rename_categories, reorder_categories, add_categories, remove_categories, set_categories

### 34.8.2.9 pandas.CategoricalIndex.set_categories

CategoricalIndex.set_categories(*args, **kwargs)
Sets 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 on python3, 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 : boolean, (default: False)
    Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

rename : boolean (default: False)
    Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

inplace : boolean (default: False)
    Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

cat : Categorical with reordered categories or None if inplace.

**Raises**

ValueError
    If new_categories does not validate as categories

See also:

rename_categories, reorder_categories, add_categories, remove_categories, remove_unused_categories

### 34.8.2.10 pandas.CategoricalIndex.as_ordered

CategoricalIndex.as_ordered(*args, **kwargs)
Sets the Categorical to be ordered

**Parameters**

inplace : boolean (default: False)
    Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True
34.8.2.11 pandas.CategoricalIndex.as_unordered

CategoricalIndex.as_unordered(*args, **kwargs)

Sets the Categorical to be unordered

Parameters inplace: boolean (default: False)

Whether or not to set the ordered attribute inplace or return a copy of this categorical
with ordered set to False

34.9 IntervalIndex

IntervalIndex

Immutable Index implementing an ordered, sliceable set.

34.9.1 pandas.IntervalIndex

class pandas.IntervalIndex

Immutable Index implementing an ordered, sliceable set. IntervalIndex represents an Index of intervals that are
all closed on the same side.

New in version 0.20.0.

Warning: The indexing behaviors are provisional and may change in a future version of pandas.

See also:

Index The base pandas Index type
Interval A bounded slice-like interval
interval_range Function to create a fixed frequency

IntervalIndex, IntervalIndex.from_arrays, IntervalIndex.from_breaks,
IntervalIndex.from_intervals, IntervalIndex.from_tuples, cut, qcut

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]],
    closed='right', dtype='interval[int64]')
```

It may also be constructed using one of the constructor methods IntervalIndex.from_arrays(),
IntervalIndex.from_breaks(), IntervalIndex.from_intervals() and IntervalIndex.from_tuples().
See further examples in the doc strings of `interval_range` and the mentioned constructor methods.

### 34.9.2 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 an IntervalIndex from a left and right array</td>
</tr>
<tr>
<td><code>IntervalIndex.from_tuples(data[, closed, ...])</code></td>
<td>Construct an IntervalIndex from a list/array 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>
<tr>
<td><code>IntervalIndex.from_intervals(data[, name, copy])</code></td>
<td>Construct an IntervalIndex from a 1d array of Interval objects</td>
</tr>
</tbody>
</table>

#### 34.9.2.1 pandas.IntervalIndex.from_arrays

**classmethod** `IntervalIndex.from_arrays(left, right, closed='right', name=None, copy=False)`

Construct an IntervalIndex from a left and right array

**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'}, optional
  
  Whether the intervals are closed on the left-side, right-side, both or neither. Defaults to 'right'.

- **name**: object, optional
  
  Name to be stored in the index.

- **copy**: boolean, default False
  
  Copy the data

**See also:**

- `interval_range` Function to create a fixed frequency IntervalIndex

- `IntervalIndex.from_breaks` Construct an IntervalIndex from an array of splits

- `IntervalIndex.from_intervals` Construct an IntervalIndex from an array of Interval objects

- `IntervalIndex.from_tuples` Construct an IntervalIndex from a list/array of tuples

**Examples**

```python
>>> pd.IntervalIndex.from_arrays([0, 1, 2], [1, 2, 3])
IntervalIndex([(0, 1], [1, 2], [2, 3]), closed='right', dtype='interval[int64]')
```
34.9.2.2 pandas.IntervalIndex.from_tuples

classmethod IntervalIndex.\texttt{from\_tuples}(data, closed='right', name=None, copy=False)

Construct an IntervalIndex from a list/array of tuples

**Parameters**

\texttt{data} : array-like (1-dimensional)

Array of tuples

\texttt{closed} : \{'left’, ‘right’, ‘both’, ‘neither’}, optional

Whether the intervals are closed on the left-side, right-side, both or neither. Defaults to ‘right’.

\texttt{name} : object, optional

Name to be stored in the index.

\texttt{copy} : boolean, default False

by-default copy the data, this is compat only and ignored

**See also:**

\texttt{interval\_range} Function to create a fixed frequency IntervalIndex

\texttt{IntervalIndex.\texttt{from\_arrays}} Construct an IntervalIndex from a left and right array

\texttt{IntervalIndex.\texttt{from\_breaks}} Construct an IntervalIndex from an array of splits

\texttt{IntervalIndex.\texttt{from\_intervals}} Construct an IntervalIndex from an array of Interval objects

**Examples**

```
>>> pd.IntervalIndex.from_tuples([(0, 1), (1,2)])
IntervalIndex([(0, 1], (1, 2]],
closed='right', dtype='interval[int64]')
```

34.9.2.3 pandas.IntervalIndex.from_breaks

nclassmethod IntervalIndex.\texttt{from\_breaks}(breaks, closed='right', name=None, copy=False)

Construct an IntervalIndex from an array of splits

**Parameters**

\texttt{breaks} : array-like (1-dimensional)

Left and right bounds for each interval.

\texttt{closed} : \{'left’, ‘right’, ‘both’, ‘neither’}, optional

Whether the intervals are closed on the left-side, right-side, both or neither. Defaults to ‘right’.

\texttt{name} : object, optional

Name to be stored in the index.

\texttt{copy} : boolean, default False

copy the data

**See also:**
**interval_range** Function to create a fixed frequency IntervalIndex

**IntervalIndex.from_arrays** Construct an IntervalIndex from a left and right array

**IntervalIndex.from_intervals** Construct an IntervalIndex from an array of Interval objects

**IntervalIndex.from_tuples** Construct an IntervalIndex from a list/array of tuples

**Examples**

```python
>>> pd.IntervalIndex.from_breaks([0, 1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3)], closed='right', dtype='interval[int64]')
```

### 34.9.2.4 pandas.IntervalIndex.from_intervals

classmethod IntervalIndex.from_intervals(data=None, name=None, copy=False)

Construct an IntervalIndex from a 1d array of Interval objects

**Parameters**
- **data**: array-like (1-dimensional)
  
  Array of Interval objects. All intervals must be closed on the same sides.

- **name**: object, optional
  
  Name to be stored in the index.

- **copy**: boolean, default False
  
  by-default copy the data, this is compat only and ignored

**See also:**

**interval_range** Function to create a fixed frequency IntervalIndex

**IntervalIndex.from_arrays** Construct an IntervalIndex from a left and right array

**IntervalIndex.from_breaks** Construct an IntervalIndex from an array of splits

**IntervalIndex.from_tuples** Construct an IntervalIndex from a list/array of tuples

**Examples**

```python
>>> pd.IntervalIndex.from_intervals([pd.Interval(0, 1), pd.Interval(1, 2)])
IntervalIndex([(0, 1], (1, 2)], closed='right', dtype='interval[int64]')
```

The generic Index constructor work identically when it infers an array of all intervals:

```python
>>> pd.Index([pd.Interval(0, 1), pd.Interval(1, 2)])
IntervalIndex([(0, 1], (1, 2)], closed='right', dtype='interval[int64]')
```
34.10 MultiIndex

<table>
<thead>
<tr>
<th>MultiIndex</th>
<th>A multi-level, or hierarchical, index object for pandas objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexSlice</td>
<td>Create an object to more easily perform multi-index slicing</td>
</tr>
</tbody>
</table>

34.10.1 pandas.MultiIndex

class pandas.MultiIndex
   A multi-level, or hierarchical, index object for pandas objects

   Parameters
   levels : sequence of arrays
      The unique labels for each level
   labels : 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 : boolean, default False
      Copy the meta-data
   verify_integrity : boolean, default True
      Check that the levels/labels 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
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 exam-
ple (using .from_arrays):
polandas: powerful Python data analysis toolkit, Release 0.21.0

>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex(levels=[[1, 2], ['blue', 'red']],
          labels=[[0, 0, 1, 1], [1, 0, 1, 0]],
          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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td>asis</td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td>base</td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td>data</td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td>dtype</td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td>dtype_str</td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>has_duplicates</td>
<td></td>
</tr>
<tr>
<td>hasnans</td>
<td></td>
</tr>
<tr>
<td>inferred_type</td>
<td></td>
</tr>
<tr>
<td>is_all_dates</td>
<td></td>
</tr>
<tr>
<td>is_monotonic</td>
<td></td>
</tr>
<tr>
<td>is_monotonic_decreasing</td>
<td></td>
</tr>
<tr>
<td>is_monotonic_increasing</td>
<td></td>
</tr>
<tr>
<td>is_unique</td>
<td></td>
</tr>
<tr>
<td>labels</td>
<td></td>
</tr>
<tr>
<td>levels</td>
<td></td>
</tr>
<tr>
<td>levshape</td>
<td></td>
</tr>
<tr>
<td>lexsort_depth</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>names</td>
<td>Names of levels in MultiIndex</td>
</tr>
<tr>
<td>nbytes</td>
<td>return the number of dimensions of the underlying data,</td>
</tr>
<tr>
<td>ndim</td>
<td>return the number of elements in the underlying data,</td>
</tr>
<tr>
<td>nlevels</td>
<td>return the number of elements in the underlying data,</td>
</tr>
<tr>
<td>shape</td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td>size</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>strides</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>values</td>
<td>return the strides of the underlying data</td>
</tr>
</tbody>
</table>

**34.10.1.1 pandas.MultiIndex.T**

MultiIndex.T

return the transpose, which is by definition self
34.10.1.2 pandas.MultiIndex.asi8

```
MultiIndex.asi8 = None
```

34.10.1.3 pandas.MultiIndex.base

```
MultiIndex.base
    return the base object if the memory of the underlying data is shared
```

34.10.1.4 pandas.MultiIndex.data

```
MultiIndex.data
    return the data pointer of the underlying data
```

34.10.1.5 pandas.MultiIndex.dtype

```
MultiIndex.dtype = None
```

34.10.1.6 pandas.MultiIndex.dtype_str

```
MultiIndex.dtype_str = None
```

34.10.1.7 pandas.MultiIndex.empty

```
MultiIndex.empty
```

34.10.1.8 pandas.MultiIndex.flags

```
MultiIndex.flags
```

34.10.1.9 pandas.MultiIndex.has_duplicates

```
MultiIndex.has_duplicates
```

34.10.1.10 pandas.MultiIndex.hasnans

```
MultiIndex.hasnans = None
```

34.10.1.11 pandas.MultiIndex.inferred_type

```
MultiIndex.inferred_type = None
```

34.10.1.12 pandas.MultiIndex.is_all_dates

```
MultiIndex.is_all_dates
```
34.10.1.13 pandas.MultiIndex.is_monotonic

MultiIndex.is_monotonic = None

34.10.1.14 pandas.MultiIndex.is_monotonic_decreasing

MultiIndex.is_monotonic_decreasing = None

34.10.1.15 pandas.MultiIndex.is_monotonic_increasing

MultiIndex.is_monotonic_increasing = None

34.10.1.16 pandas.MultiIndex.is_unique

MultiIndex.is_unique = None

34.10.1.17 pandas.MultiIndex.itemsize

MultiIndex.itemsize
  return the size of the dtype of the item of the underlying data

34.10.1.18 pandas.MultiIndex.labels

MultiIndex.labels

34.10.1.19 pandas.MultiIndex.levels

MultiIndex.levels

34.10.1.20 pandas.MultiIndex.levshape

MultiIndex.levshape

34.10.1.21 pandas.MultiIndex.lexsort_depth

MultiIndex.lexsort_depth = None

34.10.1.22 pandas.MultiIndex.name

MultiIndex.name = None

34.10.1.23 pandas.MultiIndex.names

MultiIndex.names
  Names of levels in MultiIndex
34.10.1.24 pandas.MultiIndex.nbytes

MultiIndex.\texttt{nbytes} = None

34.10.1.25 pandas.MultiIndex.ndim

MultiIndex.\texttt{ndim}

return the number of dimensions of the underlying data, by definition 1

34.10.1.26 pandas.MultiIndex.nlevels

MultiIndex.\texttt{nlevels}

34.10.1.27 pandas.MultiIndex.shape

MultiIndex.\texttt{shape}

return a tuple of the shape of the underlying data

34.10.1.28 pandas.MultiIndex.size

MultiIndex.\texttt{size}

return the number of elements in the underlying data

34.10.1.29 pandas.MultiIndex.strides

MultiIndex.\texttt{strides}

return the strides of the underlying data

34.10.1.30 pandas.MultiIndex.values

MultiIndex.\texttt{values}

Methods

\begin{Verbatim}
\begin{tabular}{ll}
\texttt{all([other])} & \\
\texttt{any([other])} & \\
\texttt{append([other])} & Append a collection of Index options together \\
\texttt{argmax([axis])} & return a ndarray of the maximum argument indexer \\
\texttt{argmin([axis])} & return a ndarray of the minimum argument indexer \\
\texttt{argsort(*args, **kwargs)} & \\
\texttt{asof(label)} & For a sorted index, return the most recent label up to and including the passed label. \\
\texttt{asof_locs(where, mask)} & where : array of timestamps \\
\texttt{astype(dtype[, copy])} & Create an Index with values cast to dtypes. \\
\texttt{contains(key)} & return a boolean if this key is IN the index \\
\texttt{copy([names, dtype, levels, labels, deep, ...])} & Make a copy of this object.
\end{tabular}
\end{Verbatim}

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete(loc)</td>
<td>Make new index with passed location deleted</td>
</tr>
<tr>
<td>difference(other)</td>
<td>Compute sorted set difference of two MultiIndex objects</td>
</tr>
<tr>
<td>drop(labels[, level, errors])</td>
<td>Make new MultiIndex 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 removed.</td>
</tr>
<tr>
<td>dropna([how])</td>
<td>Return Index without NA/NaN values</td>
</tr>
<tr>
<td>duplicated([keep])</td>
<td>Return boolean np.ndarray denoting duplicate values</td>
</tr>
<tr>
<td>equal_levels(other)</td>
<td>Return True if the levels of both MultiIndex objects are the same</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two MultiIndex objects have the same labeling information</td>
</tr>
<tr>
<td>factorize([sort, na_sentinel])</td>
<td>Encode the object as an enumerated type or categorical variable</td>
</tr>
<tr>
<td>fillna([value, downcast])</td>
<td>Fill NA/NaN values with the specified value</td>
</tr>
<tr>
<td>format([space, sparsify, adjoin, names, ...])</td>
<td>Convert arrays to MultiIndex</td>
</tr>
<tr>
<td>from_arrays(arrays[, sortorder, names])</td>
<td>Make a MultiIndex from the cartesian product of multiple iterables</td>
</tr>
<tr>
<td>from_product(iterables[, sortorder, names])</td>
<td>Make a MultiIndex from the cartesian product of multiple iterables</td>
</tr>
<tr>
<td>from_tuples(tuples[, sortorder, names])</td>
<td>Convert list of tuples to MultiIndex</td>
</tr>
<tr>
<td>get_duplicates()</td>
<td></td>
</tr>
<tr>
<td>get_indexer(target[, method, limit, tolerance])</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>get_level_values(level)</td>
<td>Return vector of label values for requested level, equal to the length of the index.</td>
</tr>
<tr>
<td>get_loc(key[, method])</td>
<td>Get location for a label or a tuple of labels as an integer, slice or boolean mask.</td>
</tr>
<tr>
<td>get_loc_level(key[, level, drop_level])</td>
<td>Get both the location for the requested label(s) and the resulting sliced index.</td>
</tr>
<tr>
<td>get_locs(seq)</td>
<td>Get location for a given label/slice/list/mask or a sequence of such as an array of integers.</td>
</tr>
<tr>
<td>get_major_bounds([start, end, step, kind])</td>
<td>For an ordered MultiIndex, compute the slice locations for input labels.</td>
</tr>
<tr>
<td>get_slice_bound(label, side, kind)</td>
<td></td>
</tr>
<tr>
<td>get_values()</td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td>groupby(values)</td>
<td>Group the index labels by a given array of values.</td>
</tr>
<tr>
<td>holds_integer()</td>
<td></td>
</tr>
<tr>
<td>identical(other)</td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td>insert(loc, item)</td>
<td>Make new MultiIndex inserting new item at location</td>
</tr>
<tr>
<td>intersection(other)</td>
<td>Form the intersection of two MultiIndex objects, sorting if possible</td>
</tr>
<tr>
<td>is_(other)</td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td>is_boolean()</td>
<td></td>
</tr>
<tr>
<td>is_categorical()</td>
<td></td>
</tr>
<tr>
<td>is_floating()</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 34.106 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_integer</td>
<td>Check if all values are integers</td>
</tr>
<tr>
<td>is_interval</td>
<td>Check if all values are intervals</td>
</tr>
<tr>
<td>is_lexsorted</td>
<td>Check if the labels are lexicographically sorted</td>
</tr>
<tr>
<td>is_lexsorted_for_tuple(tup)</td>
<td>Check if the tuple is lexicographically sorted</td>
</tr>
<tr>
<td>is_mixed</td>
<td>Check if the values are a mix of different types</td>
</tr>
<tr>
<td>is_numeric</td>
<td>Check if all values are numeric</td>
</tr>
<tr>
<td>is_object</td>
<td>Check if the values are object</td>
</tr>
<tr>
<td>is_type_compatible(kind)</td>
<td>Check if the values are compatible with the given type</td>
</tr>
<tr>
<td>isin(values[, level])</td>
<td>Check if each index value is found in the passed set of values</td>
</tr>
<tr>
<td>isna()</td>
<td>Check for missing values</td>
</tr>
<tr>
<td>isnull()</td>
<td>Check for missing values</td>
</tr>
<tr>
<td>item()</td>
<td>Return the first element of the underlying data as a python</td>
</tr>
<tr>
<td>join(other[, how, level, return_indexers, sort])</td>
<td>Join two Index objects using specified method</td>
</tr>
<tr>
<td>map(mapper)</td>
<td>Apply mapper function to an index</td>
</tr>
<tr>
<td>max()</td>
<td>Return the maximum value of the object</td>
</tr>
<tr>
<td>memory_usage([deep])</td>
<td>Return the memory usage of the object</td>
</tr>
<tr>
<td>min()</td>
<td>Return the minimum value of the object</td>
</tr>
<tr>
<td>notna()</td>
<td>Inverse of isna</td>
</tr>
<tr>
<td>notnull()</td>
<td>Inverse of isna</td>
</tr>
<tr>
<td>nunique([dropna])</td>
<td>Return number of unique elements in the object</td>
</tr>
<tr>
<td>putmask(mask, value)</td>
<td>Return a new Index with values set with the mask</td>
</tr>
<tr>
<td>ravel([order])</td>
<td>Return an ndarray of the flattened values of the underlying data</td>
</tr>
<tr>
<td>reindex(target[, method, level, limit, ...])</td>
<td>Create index with target’s values (move/add/delete values as necessary)</td>
</tr>
<tr>
<td>remove_unused_levels()</td>
<td>Create a new MultiIndex from the current that removing unused levels</td>
</tr>
<tr>
<td>rename(names[, level, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>reorder_levels(order)</td>
<td>Rearrange levels using input order</td>
</tr>
<tr>
<td>repeat(repeats, *args, **kwargs)</td>
<td>NOT IMPLEMENTED: do not call this method, as重塑 is not supported for Index objects and will raise an error.</td>
</tr>
<tr>
<td>reshape(*args, **kwargs)</td>
<td>NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.</td>
</tr>
<tr>
<td>searchsorted(value[, side, sortter])</td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td>set_labels(labels[, level, inplace, ...])</td>
<td>Set new labels on MultiIndex.</td>
</tr>
<tr>
<td>set_levels(levels[, level, inplace, ...])</td>
<td>Set new levels on MultiIndex.</td>
</tr>
<tr>
<td>set_names(names[, level, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>set_value(arr, key, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray</td>
</tr>
<tr>
<td>shift([periods, freq])</td>
<td>Shift Index containing datetime objects by input number of periods and</td>
</tr>
<tr>
<td>slice_indexer([start, end, step, kind])</td>
<td>For an ordered Index, compute the slice indexer for input labels and</td>
</tr>
<tr>
<td>slice_locs([start, end, step, kind])</td>
<td>For an ordered MultiIndex, compute the slice locations for input labels.</td>
</tr>
<tr>
<td>sort(*args, **kwargs)</td>
<td>Sort the Index values by specified criteria</td>
</tr>
<tr>
<td>sort_values((return_indexer, ascending))</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>sortlevel([level, ascending, sort_remaining])</td>
<td>Sort MultiIndex at the requested level.</td>
</tr>
</tbody>
</table>

**str** alias of *StringMethods*
Table 34.106 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>summary([name])</code></td>
<td>Swap level i with level j.</td>
</tr>
<tr>
<td><code>swaplevel([i, j])</code></td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
<tr>
<td><code>symmetric_difference(other[, result_name])</code></td>
<td>return a new MultiIndex of the values selected by the indices</td>
</tr>
<tr>
<td><code>take(indices[, axis, allow_fill, fill_value])</code></td>
<td>DEPRECATED: use <code>pandas.to_datetime()</code> instead.</td>
</tr>
<tr>
<td><code>to_datetime([dayfirst])</code></td>
<td>Create a DataFrame with the levels of the MultiIndex as columns.</td>
</tr>
<tr>
<td><code>to_frame([index])</code></td>
<td>Return a MultiIndex reshaped to conform to the shapes</td>
</tr>
<tr>
<td></td>
<td>Format specified values of <code>self</code> and return them.</td>
</tr>
<tr>
<td><code>to_native_types([slicer])</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(*args, **kwargs)</code></td>
<td>Slice index between two labels / tuples, return new MultiIndex</td>
</tr>
<tr>
<td><code>union(other)</code></td>
<td>Form the union of two MultiIndex objects, sorting if possible</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td><code>value_counts([normalize, sort, ascending, ...])</code></td>
<td>Returns object containing counts of unique values.</td>
</tr>
<tr>
<td><code>view([cls])</code></td>
<td>this is defined as a copy with the same identity</td>
</tr>
<tr>
<td><code>where(cond[, other])</code></td>
<td></td>
</tr>
</tbody>
</table>

34.10.1.31 pandas.MultiIndex.all

MultiIndex.all(other=None)

34.10.1.32 pandas.MultiIndex.any

MultiIndex.any(other=None)

34.10.1.33 pandas.MultiIndex.append

MultiIndex.append(other)

Append a collection of Index options together

Parameters other : Index or list/tuple of indices

Returns appended : Index

34.10.1.34 pandas.MultiIndex.argmax

MultiIndex.argmax(axis=None)

return a ndarray of the maximum argument indexer

See also:

numpy.ndarray.argmax
34.10.1.35 pandas.MultiIndex.argmin

```
MultiIndex.argmin(axis=None)
```
return an array of the minimum argument indexer

See also:
```
numpy.ndarray.argmin
```

34.10.1.36 pandas.MultiIndex.argsort

```
MultiIndex.argsort(*args, **kwargs)
```

34.10.1.37 pandas.MultiIndex.asof

```
MultiIndex.asof(label)
```
For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.

See also:
```
get_loc asof is a thin wrapper around get_loc with method='pad'
```

34.10.1.38 pandas.MultiIndex.asof_locs

```
MultiIndex.asof_locs(where, mask)
```
where : array of timestamps mask : array of booleans where data is not NA

34.10.1.39 pandas.MultiIndex.astype

```
MultiIndex.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 ValueError exception is raised.

Parameters
dtype : numpy dtype or pandas type

```
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.

New in version 0.19.0.

34.10.1.40 pandas.MultiIndex.contains

```
MultiIndex.contains(key)
```
return a boolean if this key is in the index

Parameters
key : object

Returns
boolean
34.10.1.41 pandas.MultiIndex.copy

MultiIndex.copy(names=None, dtype=None, levels=None, labels=None, deep=False, _set_identity=False, **kwargs)

Make a copy of this object. Names, dtype, levels and labels can be passed and will be set on new copy.

**Parameters**

- **names**: sequence, optional
- **dtype**: numpy dtype or pandas type, optional
- **levels**: sequence, optional
- **labels**: sequence, optional

**Returns**

- **copy**: MultiIndex

**Notes**

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy. This could be potentially expensive on large MultiIndex objects.

34.10.1.42 pandas.MultiIndex.delete

MultiIndex.delete(loc)

Make new index with passed location deleted

**Returns**

- **new_index**: MultiIndex

34.10.1.43 pandas.MultiIndex.difference

MultiIndex.difference(other)

Compute sorted set difference of two MultiIndex objects

**Returns**

- **diff**: MultiIndex

34.10.1.44 pandas.MultiIndex.drop

MultiIndex.drop(labels, level=None, errors='raise')

Make new MultiIndex with passed list of labels deleted

**Parameters**

- **labels**: array-like
  Must be a list of tuples
- **level**: int or level name, default None

**Returns**

- **dropped**: MultiIndex

34.10.1.45 pandas.MultiIndex.drop_duplicates

MultiIndex.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.
34.10.1.46 pandas.MultiIndex.droplevel

`MultiIndex.droplevel(level=0)`
Return Index with requested level removed. If MultiIndex has only 2 levels, the result will be of Index type not MultiIndex.

Parameters: `level` : int/level name or list thereof

Returns: `index` : Index or MultiIndex

Notes

Does not check if result index is unique or not

34.10.1.47 pandas.MultiIndex.dropna

`MultiIndex.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: `valid` : Index

34.10.1.48 pandas.MultiIndex.duplicated

`MultiIndex.duplicated(keep='first')`
Return boolean np.ndarray denoting duplicate values

Parameters: `keep` : {'first', 'last', False}, default ‘first’

• `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: `duplicated` : np.ndarray

34.10.1.49 pandas.MultiIndex.equal_levels

`MultiIndex.equal_levels(other)`
Return True if the levels of both MultiIndex objects are the same
34.10.1.50 pandas.MultiIndex.equals

MultiIndex.equals(other)
Determines if two MultiIndex objects have the same labeling information (the levels themselves do not necessarily have to be the same)

See also:

equal_levels

34.10.1.51 pandas.MultiIndex.factorize

MultiIndex.factorize(sort=False, na_sentinel=-1)
Encode the object as an enumerated type or categorical variable

Parameters sort : boolean, default False
Sort by values

na_sentinel : int, default -1
Value to mark “not found”

Returns labels : the indexer to the original array
uniques : the unique Index

34.10.1.52 pandas.MultiIndexfillna

MultiIndex.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 filled : %(klass)s

34.10.1.53 pandas.MultiIndex.format

MultiIndex.format(space=2, sparsify=None, adjoin=True, names=False, na_rep=None, formatter=None)

34.10.1.54 pandas.MultiIndex.from_arrays

classmethod MultiIndex.from_arrays(arrays, sortorder=None, names=None)
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)

Returns index : MultiIndex

See also:

MultiIndex.from_tuples Convert list of tuples to MultiIndex
MultiIndex.from_product Make a MultiIndex from cartesian product of iterables

Examples

```python
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> MultiIndex.from_arrays(arrays, names=('number', 'color'))
```

34.10.1.55 pandas.MultiIndex.from_product

classmethod MultiIndex.from_product (iterables, sortorder=None, names=None)
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 strings or None
Names for the levels in the index.

Returns index : MultiIndex

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex
MultiIndex.from_tuples Convert list of tuples to MultiIndex

Examples

```python
>>> numbers = [0, 1, 2]
>>> colors = [u'green', u'purple']
>>> MultiIndex.from_product(([numbers, colors],
                           names=['number', 'color']))
```

34.10.1.56 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)

**Returns**

**index**: MultiIndex

**See also:**

- `MultiIndex.from_arrays` Convert list of arrays to MultiIndex
- `MultiIndex.from_product` Make a MultiIndex from cartesian product of iterables

**Examples**

```python
>>> tuples = [(1, u'red'), (1, u'blue'), (2, u'red'), (2, u'blue')]
>>> MultiIndex.from_tuples(tuples, names=('number', 'color'))
```

**34.10.1.57 pandas.MultiIndex.get_duplicates**

`MultiIndex.get_duplicates()`

**34.10.1.58 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**: MultiIndex or list of tuples
- **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 most 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.
New in version 0.17.0.
New in version 0.21.0: (list-like tolerance)

**Returns**

- **indexer**: ndarray of int

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.

### Examples

```python
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

#### 34.10.1.59 pandas.MultiIndex.get_indexer_for

`MultiIndex.get_indexer_for(target, **kwargs)`

Guaranteed return of an indexer even when non-unique. This dispatches to `get_indexer` or `get_indexer_nonunique` as appropriate.

#### 34.10.1.60 pandas.MultiIndex.get_indexer_non_unique

`MultiIndex.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**: MultiIndex or list of tuples

**Returns**

- **indexer**: ndarray of int

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**: ndarray of int

An indexer into the target of the values not found. These correspond to the -1 in the indexer array.

#### 34.10.1.61 pandas.MultiIndex.get_level_values

`MultiIndex.get_level_values(level)`

Return vector of label values for requested level, 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).
Examples

Create a MultiIndex:

```python
>>> mi = pd.MultiIndex.from_arrays((['abc'], ['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')
```

### 34.10.1.62 pandas.MultiIndex.get_loc

**pandas.MultiIndex.get_loc(key, method=None)**

Get location for a label or a tuple of labels 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`  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(['abb', 'def'])
```

```python
>>> mi.get_loc('b')
slice(1, 3, None)
>>> mi.get_loc(('b', 'e'))
1
```
34.10.1.63 pandas.MultiIndex.get_loc_level

MultiIndex.get_loc_level(key, level=0, drop_level=True)

Get both the location for the requested label(s) and the resulting sliced index.

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
>>> mi = 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'))

>>> mi.get_loc_level('e', level='B')
(array([False, True, False], dtype=bool),
 Index(['b'], dtype='object', name='A'))

>>> mi.get_loc_level(['b', 'e'])
(1, None)
```

34.10.1.64 pandas.MultiIndex.get_locs

MultiIndex.get_locs(seq)

Get location for a given label/slice/list/mask or a sequence of such as an array of integers.

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:
- locs: 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')])
>>> mi.get_locs('b')
array([1, 2], dtype=int64)
>>> mi.get_locs([slice(None), ['e', 'f']])
array([1, 2], dtype=int64)
>>> mi.get_locs([[True, False, True], slice('e', 'f')])
array([2], dtype=int64)
```

34.10.1.65 pandas.MultiIndex.get_major_bounds

`MultiIndex.get_major_bounds(start=None, end=None, step=None, kind=None)`

For an ordered MultiIndex, compute the slice locations for input labels.

The input labels can be tuples representing partial levels, e.g. for a MultiIndex with 3 levels, you can pass a single value (corresponding to the first level), or a 1-, 2-, or 3-tuple.

**Parameters**
- `start` : label or tuple, default None
  - If None, defaults to the beginning
- `end` : label or tuple
  - If None, defaults to the end
- `step` : int or None
  - Slice step
- `kind` : string, optional, defaults None

**Returns** `(start, end)` : (int, int)

**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.

**Notes**

This method only works if the MultiIndex is properly lexorted. So, if only the first 2 levels of a 3-level MultiIndex are lexsorted, you can only pass two levels to .slice_locs.

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays([list('abbd'), list('deff')],
                                  names=['A', 'B'])
```

Get the slice locations from the beginning of ‘b’ in the first level until the end of the multiindex:
## 34.10.1.66 pandas.MultiIndex.get_slice_bound

`MultiIndex.get_slice_bound(label, side, kind)`

## 34.10.1.67 pandas.MultiIndex.get_value

`MultiIndex.get_value(series, key)`

## 34.10.1.68 pandas.MultiIndex.get_values

`MultiIndex.get_values()`

Return the underlying data as an ndarray

## 34.10.1.69 pandas.MultiIndex.groupby

`MultiIndex.groupby(values)`

Group the index labels by a given array of values.

**Parameters**

- **values**: array
  Values used to determine the groups.

**Returns**

- **groups**: dict
  `{group name -> group labels}`

## 34.10.1.70 pandas.MultiIndex.holds_integer

`MultiIndex.holds_integer()`

## 34.10.1.71 pandas.MultiIndex.identical

`MultiIndex.identical(other)`

Similar to `equals`, but check that other comparable attributes are also equal

## 34.10.1.72 pandas.MultiIndex.insert

`MultiIndex.insert(loc, item)`

Make new MultiIndex inserting new item at location

**Parameters**

- **loc**: int
- **item**: tuple
Must be same length as number of levels in the MultiIndex

Returns new_index : Index

34.10.1.73 pandas.MultiIndex.intersection

MultiIndex.intersection(other)
Form the intersection of two MultiIndex objects, sorting if possible

Parameters other : MultiIndex or array / Index of tuples

Returns Index

34.10.1.74 pandas.MultiIndex.is_

MultiIndex.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 True if both have same underlying data, False otherwise : bool

34.10.1.75 pandas.MultiIndex.is_boolean

MultiIndex.is_boolean()

34.10.1.76 pandas.MultiIndex.is_categorical

MultiIndex.is_categorical()

34.10.1.77 pandas.MultiIndex.is_floating

MultiIndex.is_floating()

34.10.1.78 pandas.MultiIndex.is_integer

MultiIndex.is_integer()

34.10.1.79 pandas.MultiIndex.is_interval

MultiIndex.is_interval()

34.10.1.80 pandas.MultiIndex.is_lexsorted

MultiIndex.is_lexsorted()
Return True if the labels are lexicographically sorted
34.10.81 pandas.MultiIndex.is_lexsorted_for_tuple

MultiIndex.is_lexsorted_for_tuple(tup)

34.10.82 pandas.MultiIndex.is_mixed

MultiIndex.is_mixed()

34.10.83 pandas.MultiIndex.is_numeric

MultiIndex.is_numeric()

34.10.84 pandas.MultiIndex.is_object

MultiIndex.is_object()

34.10.85 pandas.MultiIndex.is_type_compatible

MultiIndex.is_type_compatible(kind)

34.10.86 pandas.MultiIndex.isin

MultiIndex.isin(values, level=None)
Compute boolean array of whether each index value is found in the passed set of values.

Parameters
values: set or list-like
Sought values.
New in version 0.18.1.
Support for values as a set

level: str or int, optional
Name or position of the index level to use (if the index is a MultiIndex).

Returns
is_contained: ndarray (boolean dtype)

Notes

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.
34.10.1.87 pandas.MultiIndex.isna

MultiIndex.isna()
Detect missing values
New in version 0.20.0.

Returns a boolean array of whether my values are NA

See also:

isnull alias of isna
pandas.isna top-level isna

34.10.1.88 pandas.MultiIndex.isnull

MultiIndex.isnull()
Detect missing values
New in version 0.20.0.

Returns a boolean array of whether my values are NA

See also:

isnull alias of isna
pandas.isna top-level isna

34.10.1.89 pandas.MultiIndex.item

MultiIndex.item()
return the first element of the underlying data as a python scalar

34.10.1.90 pandas.MultiIndex.join

MultiIndex.join(other, how='left', level=None, return_indexers=False, sort=False)
this is an internal non-public method
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 : boolean, default False
sort : boolean, 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)
New in version 0.20.0.

Returns join_index, (left_indexer, right_indexer)
### 34.10.1.91 pandas.MultiIndex.map

**MultiIndex.map(mapper)**

Apply mapper function to an index.

**Parameters** mapper : callable

Function to be applied.

**Returns** applied : Union[Index, MultiIndex], inferred

The output of the mapping function applied to the index. If the function returns a tuple with more than one element a MultiIndex will be returned.

### 34.10.1.92 pandas.MultiIndex.max

**MultiIndex.max()**

The maximum value of the object

### 34.10.1.93 pandas.MultiIndex.memory_usage

**MultiIndex.memory_usage(deep=False)**

Memory usage of my values

**Parameters** deep : bool

Introspect the data deeply, interrogate object dtypes for system-level memory consumption

**Returns** bytes used

**See also:**

numpy.ndarray.nbytes

**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

### 34.10.1.94 pandas.MultiIndex.min

**MultiIndex.min()**

The minimum value of the object

### 34.10.1.95 pandas.MultiIndex.notna

**MultiIndex.notna()**

Inverse of isna

New in version 0.20.0.

**Returns** a boolean array of whether my values are not NA

**See also:**
### pandas: powerful Python data analysis toolkit, Release 0.21.0

#### 34.10.1.96 pandas.MultiIndex.notNull

**pandas.MultiIndex.notNull**

`MultiIndex.notNull()`

Inverse of isna

New in version 0.20.0.

- **Returns**: a boolean array of whether my values are not NA
- **See also**:

  - `notnull` alias of notna
  - `pandas.notna` top-level notna

#### 34.10.1.97 pandas.MultiIndex.nunique

**pandas.MultiIndex.nunique** *(dropna=True)*

Return number of unique elements in the object.

Excludes NA values by default.

- **Parameters** `dropna` : boolean, default True
  - Don’t include NaN in the count.
- **Returns** `nunique` : int

#### 34.10.1.98 pandas.MultiIndex.putmask

**pandas.MultiIndex.putmask** *(mask, value)*

return a new Index of the values set with the mask

- **See also**:

  - `numpy.ndarray.putmask`

#### 34.10.1.99 pandas.MultiIndex.ravel

**pandas.MultiIndex.ravel** *(order='C')*

return an ndarray of the flattened values of the underlying data

- **See also**:

  - `numpy.ndarray.ravel`

#### 34.10.1.100 pandas.MultiIndex.reindex

**pandas.MultiIndex.reindex** *(target, method=None, level=None, limit=None, tolerance=None)*

Create index with target’s values (move/add/delete values as necessary)

- **Returns** `new_index` : pd.MultiIndex
  - Resulting index
indexer: np.ndarray or None
Indices of output values in original index

34.10.1.101 pandas.MultiIndex.remove_unused_levels

MultiIndex.remove_unused_levels()
create a new MultiIndex from the current that removing unused levels, meaning that they 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.
New in version 0.20.0.

Returns MultiIndex

Examples

```python
>>> i = pd.MultiIndex.from_product([range(2), list('ab')])
MultiIndex(levels=[[0, 1], ['a', 'b']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```python
>>> i[2:]
MultiIndex(levels=[[0, 1], ['a', 'b']],
           labels=[[1, 1], [0, 1]])

The 0 from the first level is not represented and can be removed
```

```python
>>> i[2:].remove_unused_levels()
MultiIndex(levels=[[1], ['a', 'b']],
           labels=[[0, 0], [0, 1]])
```

34.10.1.102 pandas.MultiIndex.rename

MultiIndex.rename(names, level=None, inplace=False)
Set new names on index. Defaults to returning new index.

Parameters names: str or sequence
name(s) to set

level: int, level name, or sequence of int/level names (default None)
If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels). Otherwise level must be None

inplace: bool
if True, mutates in place

Returns new index (of same type and class...etc) [if inplace, returns None]
Examples

```python
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64')
>>> Index([1, 2, 3, 4]).set_names(['foo'])
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                 (2, u'one'), (2, u'two')],
                                 names=['foo', 'bar'])
>>> idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'baz', u'quz'])
>>> idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'baz', u'bar'])
```

### 34.10.1.103 pandas.MultiIndex.reorder_levels

`MultiIndex.reorder_levels(order)`

Rearrange levels using input order. May not drop or duplicate levels.

### 34.10.1.104 pandas.MultiIndex.repeat

`MultiIndex.repeat(repeats, *args, **kwargs)`

### 34.10.1.105 pandas.MultiIndex.reshape

`MultiIndex.reshape(*args, **kwargs)`

NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.

Reshape an Index.

### 34.10.1.106 pandas.MultiIndex.searchsorted

`MultiIndex.searchsorted(value, side='left', sorter=None)`

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted IndexOpsMixin `self` such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

**Parameters**

- `value`: array_like
  - 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 indices : array of ints

Array of insertion points with the same shape as value.

See also:

numpy.searchsorted

Notes

Binary search is used to find the required insertion points.

Examples

```python
>>> x = pd.Series([1, 2, 3])
>>> x
0 1
1 2
2 3
dtype: int64

>>> x.searchsorted(4)
array([3])

>>> x.searchsorted([0, 4])
array([0, 3])

>>> x.searchsorted([1, 3], side='left')
array([0, 2])

>>> x.searchsorted([1, 3], side='right')
array([1, 3])

>>> x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'])
>>> x
Categories (4, object): [apple < bread < cheese < milk]

>>> x.searchsorted('bread')
array([1])  # Note: an array, not a scalar

>>> x.searchsorted(['bread'])
array([1])

>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])  # eggs before milk
```

34.10. MultiIndex 1785
34.10.1.107 pandas.MultiIndex.set_labels

MultiIndex.set_labels(labels, level=None, inplace=False, verify_integrity=True)

Set new labels on MultiIndex. Defaults to returning new index.

Parameters labels : sequence or list of sequence

new labels 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

verify_integrity : bool (default True)

if True, checks that levels and labels are compatible

Returns new index (of same type and class...etc)

Examples

```python
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                 (2, u'one'), (2, u'two')],
                                names=['foo', 'bar'])
>>> idx.set_labels([[1,0,1,0], [0,0,1,1]],
                  level=[0,1],
                  level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           labels=[[1, 0, 1, 0], [0, 0, 1, 1]],
           names=['foo', 'bar'])
```

34.10.1.108 pandas.MultiIndex.set_levels

MultiIndex.set_levels(levels, level=None, inplace=False, 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

verify_integrity : bool (default True)

if True, checks that levels and labels are compatible

Returns  new index (of same type and class...etc)

Examples

```python
>>> idx = MultiIndex.from_tuples([(1, 'one'), (1, 'two'),
                                (2, 'one'), (2, 'two')],
                               names=['foo', 'bar'])
>>> idx.set_levels([['a','b'], [1,2]], level=0)
MultiIndex(levels=[['a', 'b'], [u'one', u'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['foo', 'bar'])
>>> idx.set_levels([['a','b'], [1,2]], level='bar')
MultiIndex(levels=[[1, 2], ['a', 'b']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['foo', 'bar'])
>>> idx.set_levels([['a','b'], [1,2]], level=[0,1])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['foo', 'bar'])
```

34.10.1.109 pandas.MultiIndex.set_names

MultiIndex.set_names(names, level=None, inplace=False)

Set new names on index. Defaults to returning new index.

Parameters names : str or sequence

name(s) to set

level : int, level name, or sequence of int/level names (default None)

If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels).
Otherwise level must be None

inplace : bool

if True, mutates in place

Returns  new index (of same type and class...etc) [if inplace, returns None]

Examples

```python
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64')
```

34.10. MultiIndex
34.10.1.110 pandas.MultiIndex.set_value

MultiIndex.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing.

34.10.1.111 pandas.MultiIndex.shift

MultiIndex.shift(periods=1, freq=None)

Shift Index containing datetime objects by input number of periods and DateOffset

Returns shifted : Index

34.10.1.112 pandas.MultiIndex.slice_indexer

MultiIndex.slice_indexer(start=None, end=None, step=None, kind=None)

For an ordered Index, compute the slice indexer for input labels and step

Parameters start : label, default None
    If None, defaults to the beginning
e : label, default None
    If None, defaults to the end
step : int, default None
kind : string, default None

Returns indexer : ndarray or slice

Notes

This function assumes that the data is sorted, so use at your own peril

34.10.1.113 pandas.MultiIndex.slice_locs

MultiIndex.slice_locs(start=None, end=None, step=None, kind=None)

For an ordered MultiIndex, compute the slice locations for input labels.

The input labels can be tuples representing partial levels, e.g. for a MultiIndex with 3 levels, you can pass a single value (corresponding to the first level), or a 1-, 2-, or 3-tuple.

Parameters start : label or tuple, default None
If None, defaults to the beginning

**end**: label or tuple
If None, defaults to the end

**step**: int or None
Slice step

**kind**: string, optional, defaults None

**Returns**: (start, end) : (int, int)

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.

**Notes**

This method only works if the MultiIndex is properly lex-sorted. So, if only the first 2 levels of a 3-level MultiIndex are lexsorted, you can only pass two levels to `slice_locs`.

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays([list('abbd'), list('deff')],
                                names=['A', 'B'])

Get the slice locations from the beginning of ‘b’ in the first level until the end of the multiindex:

```python
>>> mi.slice_locs(start='b')
(1, 4)
```  

Like above, but stop at the end of ‘b’ in the first level and ‘f’ in the second level:

```python
>>> mi.slice_locs(start='b', end=('b', 'f'))
(1, 3)
```  

### 34.10.1.114 pandas.MultiIndex.sort

```
MultiIndex.sort(*args, **kwargs)
```

### 34.10.1.115 pandas.MultiIndex.sort_values

```
MultiIndex.sort_values(return_indexer=False, ascending=True)
```

Return sorted copy of Index

### 34.10.1.116 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: boolean, 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

Indices of output values in original index

34.10.1.117 pandas.MultiIndex.str

MultiIndex.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.str.split('_')
```

```python
>>> s.str.replace('_', '')
```

34.10.1.118 pandas.MultiIndex.summary

MultiIndex.summary(name=None)

34.10.1.119 pandas.MultiIndex.swaplevel

MultiIndex.swaplevel(i=-2, j=-1)

Swap level i with level j. Do not change the ordering of anything.

Parameters i, j: int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns swapped: MultiIndex

Changed in version 0.18.1: The indexes i and j are now optional, and default to the two innermost levels of the index.

34.10.1.120 pandas.MultiIndex.symmetric_difference

MultiIndex.symmetric_difference(other, result_name=None)

Compute the symmetric difference of two Index objects. It’s sorted if sorting is possible.

Parameters other: Index or array-like

result_name: str
Returns \texttt{symmetric\_difference} : Index

Notes

\texttt{symmetric\_difference} contains elements that appear in either \texttt{idx1} or \texttt{idx2} but not both. Equivalent to the Index created by \texttt{idx1.difference(idx2) \cup idx2.difference(idx1)} with duplicates dropped.

Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the \texttt{^} operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

34.10.1.121 pandas.MultiIndex\texttt{.take}

\texttt{MultiIndex.take(indices, axis=0, allow_fill=True, fill_value=None, **kwargs)}

return a new MultiIndex of the values selected by the indices

For internal compatibility with numpy arrays.

Parameters indices : list

Indices to be taken

axis : int, optional

The axis over which to select values, always 0.

allow_fill : bool, default True

fill_value : bool, default None

If allow_fill=True and fill_value is not None, indices specified by -1 is regarded as NA. If Index doesn’t hold NA, raise ValueError

See also:

\texttt{numpy.ndarray.take}

34.10.1.122 pandas.MultiIndex\texttt{.to\_datetime}

\texttt{MultiIndex.to\_datetime(dayfirst=False)}

DEPRECATED: use pandas\texttt{.to\_datetime()} instead.

For an Index containing strings or \texttt{datetime.datetime} objects, attempt conversion to DatetimeIndex
### 34.10.1.123 pandas.MultiIndex.to_frame

`MultiIndex.to_frame(index=True)`

Create a DataFrame with the levels of the MultiIndex as columns.

New in version 0.20.0.

**Parameters**

- `index` : boolean, default True
  
  Set the index of the returned DataFrame as the original MultiIndex.

**Returns**

- `DataFrame` : a DataFrame containing the original MultiIndex data.

### 34.10.1.124 pandas.MultiIndex.to_hierarchical

`MultiIndex.to_hierarchical(n_repeat, n_shuffle=1)`

Return a MultiIndex reshaped to conform to the shapes given by `n_repeat` and `n_shuffle`.

Useful to replicate and rearrange a MultiIndex for combination with another Index with `n_repeat` items.

**Parameters**

- `n_repeat` : int
  
  Number of times to repeat the labels on self

- `n_shuffle` : int
  
  Controls the reordering of the labels. If the result is going to be an inner level in a MultiIndex, `n_shuffle` will need to be greater than one. The size of each label must divisible by `n_shuffle`.

**Returns**

- `MultiIndex`

**Examples**

```python
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                               (2, u'one'), (2, u'two')])

>>> idx.to_hierarchical(3)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           labels=[[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
                   [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1]])
```

### 34.10.1.125 pandas.MultiIndex.to_native_types

`MultiIndex.to_native_types(slicer=None, **kwargs)`

Format specified values of `self` and return them.

**Parameters**

- `slicer` : int, array-like
  
  An indexer into `self` that specifies which values are used in the formatting process.

- `**kwargs` : dict
  Options for specifying how the values should be formatted. These options include the following:

1. `na_rep` : The value that serves as a placeholder for NULL values
2. `quoting` : Whether or not there are quoted values in `self`
3. `date_format` : The format used to represent date-like values
34.10.1.126 pandas.MultiIndex.to_series

MultiIndex.to_series(**kwargs)
Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index

Returns Series : dtype will be based on the type of the Index values.

34.10.1.127 pandas.MultiIndex.tolist

MultiIndex.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)

See also:
numpy.ndarray.tolist

34.10.1.128 pandas.MultiIndex.transpose

MultiIndex.transpose(*args, **kwargs)
return the transpose, which is by definition self

34.10.1.129 pandas.MultiIndex.truncate

MultiIndex.truncate(before=None, after=None)
Slice index between two labels / tuples, return new MultiIndex

Parameters before : label or tuple, can be partial. Default None

None defaults to start

after : label or tuple, can be partial. Default None

None defaults to end

Returns truncated : MultiIndex

34.10.1.130 pandas.MultiIndex.union

MultiIndex.union(other)
Form the union of two MultiIndex objects, sorting if possible

Parameters other : MultiIndex or array / Index of tuples

Returns Index

>>> index.union(index2)


### pandas.MultiIndex.unique

#### Parameters
- **values**: 1d array-like

#### Returns
- unique values.
  - If the input is an Index, the return is an Index
  - If the input is a Categorical dtype, the return is a Categorical
  - If the input is a Series/ndarray, the return will be an ndarray

See also:
- unique, Index.unique, Series.unique

### pandas.MultiIndex.value_counts

#### Parameters
- **normalize**: boolean, default False
- **sort**: boolean, default True
- **ascending**: boolean, default False
- **bins**: integer, optional
- **dropna**: boolean, default True

#### Returns
- counts: Series

### pandas.MultiIndex.view

#### Parameters
- **cls**: None

this is defined as a copy with the same identity

### pandas.MultiIndex.where

#### Parameters
- **cond**, **other**: None

34.10.2 pandas.IndexSlice

pandas.IndexSlice = <pandas.core.indexing._IndexSlice object>
Create an object to more easily perform multi-index slicing

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:
```none```
>>> 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:
```none```
```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
```

34.10.3 MultiIndex Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiIndex.from_arrays(arrays[, sortorder,...])</td>
<td>Convert arrays to MultiIndex</td>
</tr>
<tr>
<td>MultiIndex.from_tuples(tuples[, sortorder,...])</td>
<td>Convert list of tuples to MultiIndex</td>
</tr>
<tr>
<td>MultiIndex.from_product(iterables[, ...])</td>
<td>Make a MultiIndex from the cartesian product of multiple iterables</td>
</tr>
<tr>
<td>MultiIndex.set_levels(levels[, level,...])</td>
<td>Set new levels on MultiIndex.</td>
</tr>
<tr>
<td>MultiIndex.set_labels(labels[, level,...])</td>
<td>Set new labels on MultiIndex.</td>
</tr>
<tr>
<td>MultiIndex.to_hierarchical([n_repeat[, n_shuffle]])</td>
<td>Return a MultiIndex reshaped to conform to the shapes given by n_repeat and n_shuffle.</td>
</tr>
<tr>
<td>MultiIndex.to_frame([index])</td>
<td>Create a DataFrame with the levels of the MultiIndex as columns.</td>
</tr>
<tr>
<td>MultiIndex.is_lexsorted()</td>
<td>Return True if the labels are lexicographically sorted</td>
</tr>
<tr>
<td>MultiIndex.drop_level([level])</td>
<td>Return Index with requested level removed.</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 a new MultiIndex from the current that removing</td>
</tr>
</tbody>
</table>

34.11 DatetimeIndex
Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata such as frequency information.

34.11.1 pandas.DatetimeIndex

class pandas.DatetimeIndex

Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata such as frequency information.

Parameters

- **data**: array-like (1-dimensional), optional
  - Optional datetime-like data to construct index with
- **copy**: bool
  - Make a copy of input ndarray
- **freq**: string or pandas offset object, optional
  - One of pandas date offset strings or corresponding objects
- **start**: starting value, datetime-like, optional
  - If data is None, start is used as the start point in generating regular timestamp data.
- **periods**: int, optional, > 0
  - Number of periods to generate, if generating index. Takes precedence over end argument
- **end**: end time, datetime-like, optional
  - If periods is none, generated index will extend to first conforming time on or just past end argument
- **closed**: string or None, default None
  - Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)
- **tz**: pytz.timezone or dateutil.tz.tzfile
- **ambiguous**: ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’
  - ‘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
- **infer_dst**: boolean, default False
  - Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order
- **name**: object
  - 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

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>T</td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td>asts</td>
<td>return object Index which contains boxed values</td>
</tr>
<tr>
<td>asobject</td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td>base</td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td>data</td>
<td>Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).</td>
</tr>
<tr>
<td>day</td>
<td>The days of the datetime</td>
</tr>
<tr>
<td>dayofweek</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>dayofyear</td>
<td>The ordinal day of the year</td>
</tr>
<tr>
<td>days_in_month</td>
<td>The number of days in the month</td>
</tr>
<tr>
<td>daysinmonth</td>
<td>The number of days in the month</td>
</tr>
<tr>
<td>dtype</td>
<td></td>
</tr>
<tr>
<td>dtype_str</td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>flags</td>
<td>get/set the frequency of the Index</td>
</tr>
<tr>
<td>freq</td>
<td>Return the frequency object as a string if its set, otherwise None</td>
</tr>
<tr>
<td>freqstr</td>
<td></td>
</tr>
<tr>
<td>has_duplicates</td>
<td></td>
</tr>
<tr>
<td>hasnans</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td>The hours of the datetime</td>
</tr>
<tr>
<td>inferred_freq</td>
<td></td>
</tr>
<tr>
<td>inferred_type</td>
<td></td>
</tr>
<tr>
<td>is_all_dates</td>
<td></td>
</tr>
<tr>
<td>is_leap_year</td>
<td>Logical indicating if the date belongs to a leap year</td>
</tr>
<tr>
<td>is_monotonic</td>
<td>alias for is_monotonic_increasing (deprecated)</td>
</tr>
<tr>
<td>is_monotonic_decreasing</td>
<td>return if the index is monotonic decreasing (only equal or</td>
</tr>
<tr>
<td>is_monotonic_increasing</td>
<td>return if the index is monotonic increasing (only equal or</td>
</tr>
<tr>
<td>is_month_end</td>
<td>Logical indicating if last day of month (defined by frequency)</td>
</tr>
<tr>
<td>is_month_start</td>
<td>Logical indicating if first day of month (defined by frequency)</td>
</tr>
<tr>
<td>is_normalized</td>
<td></td>
</tr>
<tr>
<td>Continued on next page</td>
<td></td>
</tr>
</tbody>
</table>
Table 34.109 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_quarter_end</code></td>
<td>Logical indicating if last day of quarter (defined by frequency)</td>
</tr>
<tr>
<td><code>is_quarter_start</code></td>
<td>Logical indicating if first day of quarter (defined by frequency)</td>
</tr>
<tr>
<td><code>is_unique</code></td>
<td></td>
</tr>
<tr>
<td><code>is_year_end</code></td>
<td>Logical indicating if last day of year (defined by frequency)</td>
</tr>
<tr>
<td><code>is_year_start</code></td>
<td>Logical indicating if first day of year (defined by frequency)</td>
</tr>
<tr>
<td><code>itemsize</code></td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td><code>microsecond</code></td>
<td>The microseconds of the datetime</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td><code>month</code></td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td><code>name</code></td>
<td></td>
</tr>
<tr>
<td><code>names</code></td>
<td></td>
</tr>
<tr>
<td><code>nanosecond</code></td>
<td>The nanoseconds of the datetime</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>return the number of dimensions of the underlying data,</td>
</tr>
<tr>
<td><code>nlevels</code></td>
<td></td>
</tr>
<tr>
<td><code>offset</code></td>
<td></td>
</tr>
<tr>
<td><code>quarter</code></td>
<td>The quarter of the date</td>
</tr>
<tr>
<td><code>resolution</code></td>
<td></td>
</tr>
<tr>
<td><code>second</code></td>
<td>The seconds of the datetime</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>strides</code></td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td><code>time</code></td>
<td>Returns numpy array of datetime.time.</td>
</tr>
<tr>
<td><code>tz</code></td>
<td>Alias for tz attribute</td>
</tr>
<tr>
<td><code>values</code></td>
<td>return the underlying data as an ndarray</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>weekday_name</code></td>
<td>The name of day in a week (ex: Friday)</td>
</tr>
<tr>
<td><code>weekofyear</code></td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td><code>year</code></td>
<td>The year of the datetime</td>
</tr>
</tbody>
</table>

**34.11.1 pandas.DatetimeIndex.T**

DatetimeIndex.T

return the transpose, which is by definition self

**34.11.2 pandas.DatetimeIndex.asi8**

DatetimeIndex.asi8

**34.11.3 pandas.DatetimeIndex.asobject**

DatetimeIndex.asobject

return object Index which contains boxed values
this is an internal non-public method

34.11.1.4 pandas.DatetimeIndex.base

DatetimeIndex.base
return the base object if the memory of the underlying data is shared

34.11.1.5 pandas.DatetimeIndex.data

DatetimeIndex.data
return the data pointer of the underlying data

34.11.1.6 pandas.DatetimeIndex.date

DatetimeIndex.date
Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).

34.11.1.7 pandas.DatetimeIndex.day

DatetimeIndex.day
The days of the datetime

34.11.1.8 pandas.DatetimeIndex.dayofweek

DatetimeIndex.dayofweek
The day of the week with Monday=0, Sunday=6

34.11.1.9 pandas.DatetimeIndex.dayofyear

DatetimeIndex.dayofyear
The ordinal day of the year

34.11.1.10 pandas.DatetimeIndex.days_in_month

DatetimeIndex.days_in_month
The number of days in the month

34.11.1.11 pandas.DatetimeIndex.daysinmonth

DatetimeIndex.daysinmonth
The number of days in the month

34.11.1.12 pandas.DatetimeIndex.dtype

DatetimeIndex.dtype = None
34.11.1.13 `pandas.DatetimeIndex.dtype_str`

```
DatetimeIndex.dtype_str = None
```

34.11.1.14 `pandas.DatetimeIndex.empty`

```
DatetimeIndex.empty
```

34.11.1.15 `pandas.DatetimeIndex.flags`

```
DatetimeIndex.flags
```

34.11.1.16 `pandas.DatetimeIndex.freq`

```
DatetimeIndex.freq
  get/set the frequency of the Index
```

34.11.1.17 `pandas.DatetimeIndex.freqstr`

```
DatetimeIndex.freqstr
  Return the frequency object as a string if its set, otherwise None
```

34.11.1.18 `pandas.DatetimeIndex.has_duplicates`

```
DatetimeIndex.has_duplicates
```

34.11.1.19 `pandas.DatetimeIndex.hasnans`

```
DatetimeIndex.hasnans = None
```

34.11.1.20 `pandas.DatetimeIndex.hour`

```
DatetimeIndex.hour
  The hours of the datetime
```

34.11.1.21 `pandas.DatetimeIndex.inferred_freq`

```
DatetimeIndex.inferred_freq = None
```

34.11.1.22 `pandas.DatetimeIndex.inferred_type`

```
DatetimeIndex.inferred_type
```

34.11.1.23 `pandas.DatetimeIndex.is_all_dates`

```
DatetimeIndex.is_all_dates
```
34.11.24 pandas.DatetimeIndex.is_leap_year

DatetimeIndex.is_leap_year
Logical indicating if the date belongs to a leap year

34.11.25 pandas.DatetimeIndex.is_monotonic

DatetimeIndex.is_monotonic
alias for is_monotonic_increasing (deprecated)

34.11.26 pandas.DatetimeIndex.is_monotonic_decreasing

DatetimeIndex.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
```

34.11.27 pandas.DatetimeIndex.is_monotonic_increasing

DatetimeIndex.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
```

34.11.28 pandas.DatetimeIndex.is_month_end

DatetimeIndex.is_month_end
Logical indicating if last day of month (defined by frequency)

34.11.29 pandas.DatetimeIndex.is_month_start

DatetimeIndex.is_month_start
Logical indicating if first day of month (defined by frequency)
34.11.1.30 pandas.DatetimeIndex.is_normalized

`DatetimeIndex.is_normalized = None`

34.11.1.31 pandas.DatetimeIndex.is_quarter_end

`DatetimeIndex.is_quarter_end
Logical indicating if last day of quarter (defined by frequency)`

34.11.1.32 pandas.DatetimeIndex.is_quarter_start

`DatetimeIndex.is_quarter_start
Logical indicating if first day of quarter (defined by frequency)`

34.11.1.33 pandas.DatetimeIndex.is_unique

`DatetimeIndex.is_unique = None`

34.11.1.34 pandas.DatetimeIndex.is_year_end

`DatetimeIndex.is_year_end
Logical indicating if last day of year (defined by frequency)`

34.11.1.35 pandas.DatetimeIndex.is_year_start

`DatetimeIndex.is_year_start
Logical indicating if first day of year (defined by frequency)`

34.11.1.36 pandas.DatetimeIndex.itemsize

`DatetimeIndex.itemsize
return the size of the dtype of the item of the underlying data`

34.11.1.37 pandas.DatetimeIndex.microsecond

`DatetimeIndex.microsecond
The microseconds of the datetime`

34.11.1.38 pandas.DatetimeIndex.minute

`DatetimeIndex.minute
The minutes of the datetime`

34.11.1.39 pandas.DatetimeIndex.month

`DatetimeIndex.month
The month as January=1, December=12`
34.11.1.40 pandas.DatetimeIndex.name

DatetimeIndex.name = None

34.11.1.41 pandas.DatetimeIndex.names

DatetimeIndex.names

34.11.1.42 pandas.DatetimeIndex.nanosecond

DatetimeIndex.nanosecond
   The nanoseconds of the datetime

34.11.1.43 pandas.DatetimeIndex.nbytes

DatetimeIndex.nbytes
   return the number of bytes in the underlying data

34.11.1.44 pandas.DatetimeIndex.ndim

DatetimeIndex.ndim
   return the number of dimensions of the underlying data, by definition 1

34.11.1.45 pandas.DatetimeIndex.nlevels

DatetimeIndex.nlevels

34.11.1.46 pandas.DatetimeIndex.offset

DatetimeIndex.offset = None

34.11.1.47 pandas.DatetimeIndex.quarter

DatetimeIndex.quarter
   The quarter of the date

34.11.1.48 pandas.DatetimeIndex.resolution

DatetimeIndex.resolution = None

34.11.1.49 pandas.DatetimeIndex.second

DatetimeIndex.second
   The seconds of the datetime
**pandas: powerful Python data analysis toolkit, Release 0.21.0**

34.11.1.50 `pandas.DatetimeIndex.shape`

```
DatetimeIndex.shape
```

return a tuple of the shape of the underlying data

34.11.1.51 `pandas.DatetimeIndex.size`

```
DatetimeIndex.size
```

return the number of elements in the underlying data

34.11.1.52 `pandas.DatetimeIndex.strides`

```
DatetimeIndex.strides
```

return the strides of the underlying data

34.11.1.53 `pandas.DatetimeIndex.time`

```
DatetimeIndex.time
```

Returns numpy array of datetime.time. The time part of the Timestamps.

34.11.1.54 `pandas.DatetimeIndex.tz`

```
DatetimeIndex.tz = None
```

34.11.1.55 `pandas.DatetimeIndex.tzinfo`

```
DatetimeIndex.tzinfo
```

Alias for tz attribute

34.11.1.56 `pandas.DatetimeIndex.values`

```
DatetimeIndex.values
```

return the underlying data as an ndarray

34.11.1.57 `pandas.DatetimeIndex.week`

```
DatetimeIndex.week
```

The week ordinal of the year

34.11.1.58 `pandas.DatetimeIndex.weekday`

```
DatetimeIndex.weekday
```

The day of the week with Monday=0, Sunday=6
34.11.1.59 pandas.DatetimeIndex.weekday_name

DatetimeIndex.weekday_name
The name of day in a week (ex: Friday)
New in version 0.18.1.

34.11.1.60 pandas.DatetimeIndex.weekofyear

DatetimeIndex.weekofyear
The week ordinal of the year

34.11.1.61 pandas.DatetimeIndex.year

DatetimeIndex.year
The year of the datetime

Methods

all([other])
any([other])
apend(other) Append a collection of Index options together
argmax([axis]) Returns the indices of the maximum values along an
axis.
argmin([axis]) Returns the indices of the minimum values along an
axis.
argsort(*args, **kwargs) Returns the indices that would sort the index and its un-
derlying data.
asof(label) For a sorted index, return the most recent label up to and
including the passed label.
asof_locs(where, mask) where : array of timestamps
astype(dtype, copy) Create an Index with values cast to dtypes.
ceil(freq) ceil the index to the specified freq
contains(key) return a boolean if this key is IN the index
copy([name, deep, dtype]) Make a copy of this object.
delete(loc) Make a new DatetimeIndex with passed location(s)
deleted.
difference(other) Return a new Index with elements from the index that
are not in other.
drop(labels[, errors]) Make new Index with passed list of labels deleted
drop_duplicates([keep]) Return Index with duplicate values removed
dropna([how]) Return Index without NA/NaN values
duplicated([keep]) Return boolean np.ndarray denoting duplicate values
equals(other) Determines if two Index objects contain the same ele-
ments.
factorize([sort, na_sentinel]) Encode the object as an enumerated type or categorical
variable
fillna([value, downcast]) Fill NA/NaN values with the specified value
floor(freq) floor the index to the specified freq
### Table 34.110 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format([name, formatter])</td>
<td>Render a string representation of the Index</td>
</tr>
<tr>
<td>get_duplicates()</td>
<td></td>
</tr>
<tr>
<td>get_indexer(target[, method, limit, tolerance])</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>get_indexer_for(target, **kwargs)</td>
<td>guaranteed return of an indexer even when non-unique</td>
</tr>
<tr>
<td>get_indexer_non_unique(target)</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>get_level_values(level)</td>
<td>Return an Index of values for requested level, equal to the length of the index.</td>
</tr>
<tr>
<td>get_loc(key[, method, tolerance])</td>
<td>Get integer location for requested label</td>
</tr>
<tr>
<td>get_slice_bound(label, side, kind)</td>
<td>Calculate slice bound that corresponds to given label.</td>
</tr>
<tr>
<td>get_value(series, key)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>get_value_maybe_box(series, key)</td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td>groupby(values)</td>
<td>Group the index labels by a given array of values.</td>
</tr>
<tr>
<td>holds_integer()</td>
<td></td>
</tr>
<tr>
<td>identical(other)</td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td>indexer_at_time(time[, asof])</td>
<td>Select values at particular time of day (e.g.</td>
</tr>
<tr>
<td>indexer_between_time(start_time, end_time[, ...])</td>
<td>Select values between particular times of day (e.g., 9:00-9:30AM).</td>
</tr>
<tr>
<td>insert(loc, item)</td>
<td>Make new Index inserting new item at location</td>
</tr>
<tr>
<td>intersection(other)</td>
<td>Specialized intersection for DatetimeIndex objects.</td>
</tr>
<tr>
<td>is_(other)</td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td>is_boolean()</td>
<td></td>
</tr>
<tr>
<td>is_categorical()</td>
<td></td>
</tr>
<tr>
<td>is_floating()</td>
<td></td>
</tr>
<tr>
<td>is_integer()</td>
<td></td>
</tr>
<tr>
<td>is_interval()</td>
<td></td>
</tr>
<tr>
<td>is_lexsorted_for_tuple(tup)</td>
<td></td>
</tr>
<tr>
<td>is_mixed()</td>
<td></td>
</tr>
<tr>
<td>is_numeric()</td>
<td></td>
</tr>
<tr>
<td>is_object()</td>
<td></td>
</tr>
<tr>
<td>is_type_compatible(typ)</td>
<td></td>
</tr>
<tr>
<td>isin(values)</td>
<td>Compute boolean array of whether each index value is found in the</td>
</tr>
<tr>
<td>isna()</td>
<td>Detect missing values</td>
</tr>
<tr>
<td>isnull()</td>
<td>Detect missing values</td>
</tr>
<tr>
<td>item()</td>
<td>return the first element of the underlying data as a python</td>
</tr>
<tr>
<td>join(other[, how, level, return_indexers, sort])</td>
<td>See Index.join</td>
</tr>
<tr>
<td>map(f)</td>
<td></td>
</tr>
<tr>
<td>max([axis])</td>
<td>Return the maximum value of the Index or maximum along an axis.</td>
</tr>
<tr>
<td>memory_usage([deep])</td>
<td>Memory usage of my values</td>
</tr>
<tr>
<td>min([axis])</td>
<td>Return the minimum value of the Index or minimum along an axis.</td>
</tr>
<tr>
<td>normalize()</td>
<td>Return DatetimeIndex with times to midnight.</td>
</tr>
<tr>
<td>notna()</td>
<td>Inverse of isna</td>
</tr>
</tbody>
</table>

Continued on next page
### pandas: powerful Python data analysis toolkit, Release 0.21.0

#### Table 34.110 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>notnull()</code></td>
<td>Inverse of isna</td>
</tr>
<tr>
<td><code>nuniquet([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>
<tr>
<td><code>reindex([target[, method, level, limit, ...]])</code></td>
<td>Create index with target’s values (move/add/delete values as necessary)</td>
</tr>
<tr>
<td><code>rename(name[, inplace])</code></td>
<td>Set new names on index.</td>
</tr>
<tr>
<td><code>repeat([repeats, **kwargs])</code></td>
<td>Analogous to ndarray.repeat</td>
</tr>
<tr>
<td><code>reshape(**kwargs)</code></td>
<td>NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.</td>
</tr>
<tr>
<td><code>round(freq, **kwargs)</code></td>
<td>round the index to the specified freq</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>set_names(names[, level, inplace])</code></td>
<td>Set new names on index.</td>
</tr>
<tr>
<td><code>set_value(arr, key, value)</code></td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>shift(n[, freq])</code></td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td><code>slice_indexer([start, end, step, kind])</code></td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td><code>slice_locs([start, end, step, kind])</code></td>
<td>Compute slice locations for input labels.</td>
</tr>
<tr>
<td><code>snap([freq])</code></td>
<td>Snap time stamps to nearest occurring frequency</td>
</tr>
<tr>
<td><code>sort(*args, **kwargs)</code></td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td><code>sort_values([return_indexer, ascending])</code></td>
<td>For internal compatibility with with the Index API</td>
</tr>
<tr>
<td><code>str</code></td>
<td>alias of StringMethods</td>
</tr>
<tr>
<td><code>strftime(date_format)</code></td>
<td>Return an array of formatted strings specified by date_format, which supports the same string format as the python standard library.</td>
</tr>
<tr>
<td><code>summary([name])</code></td>
<td>return a summarized representation</td>
</tr>
<tr>
<td><code>symmetric_difference(other[, result_name])</code></td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
<tr>
<td><code>take(indices[, axis, allow_fill, fill_value])</code></td>
<td>return a new Index of the values selected by the indices</td>
</tr>
<tr>
<td><code>to_datetime([dayfirst])</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td><code>to_frame()</code></td>
<td>Convert DatetimeIndex to Float64Index of Julian Dates.</td>
</tr>
<tr>
<td><code>to_period([freq])</code></td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td><code>to_perioddelta(freq)</code></td>
<td>Calculates TimedeltaIndex of difference between index values and index converted to PeriodIndex at specified freq.</td>
</tr>
<tr>
<td><code>to_pydatetime()</code></td>
<td>Return DatetimeIndex as object ndarray of datetime.datetime objects</td>
</tr>
<tr>
<td><code>to_series([keep_tz])</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 underlying data</td>
</tr>
<tr>
<td><code>transpose(*args, **kwargs)</code></td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td><code>tz_convert(tz)</code></td>
<td>Convert tz-aware DatetimeIndex from one time zone to another (using</td>
</tr>
<tr>
<td><code>tz_localize(tz[, ambiguous, errors])</code></td>
<td>Localize tz-naive DatetimeIndex to given time zone (using</td>
</tr>
<tr>
<td><code>union(other)</code></td>
<td>Specialized union for DatetimeIndex objects.</td>
</tr>
</tbody>
</table>

Continued on next page
### pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>union_many()</code></td>
<td>A bit of a hack to accelerate unioning a collection of indexes</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td><code>value_counts()</code></td>
<td>Returns object containing counts of unique values.</td>
</tr>
<tr>
<td><code>view([cls])</code></td>
<td>New in version 0.19.0.</td>
</tr>
</tbody>
</table>

#### 34.11.1.62 pandas.DatetimeIndex.all

```python
DatetimeIndex.all(other=None)
```

#### 34.11.1.63 pandas.DatetimeIndex.any

```python
DatetimeIndex.any(other=None)
```

#### 34.11.1.64 pandas.DatetimeIndex.append

```python
DatetimeIndex.append(other)
```

- **Parameters**
  - `other`: Index or list/tuple of indices
- **Returns**
  - `appended`: Index

#### 34.11.1.65 pandas.DatetimeIndex.argmax

```python
DatetimeIndex.argmax(axis=None, *args, **kwargs)
```

Returns the indices of the maximum values along an axis. See `numpy.ndarray.argmax` for more information on the `axis` parameter.

- **See also**
  - `numpy.ndarray.argmax`

#### 34.11.1.66 pandas.DatetimeIndex.argmin

```python
DatetimeIndex.argmin(axis=None, *args, **kwargs)
```

Returns the indices of the minimum values along an axis. See `numpy.ndarray.argmin` for more information on the `axis` parameter.

- **See also**
  - `numpy.ndarray.argmin`

#### 34.11.1.67 pandas.DatetimeIndex.argsort

```python
DatetimeIndex.argsort(*args, **kwargs)
```

Returns the indices that would sort the index and its underlying data.

- **Returns**
  - `argsorted`: numpy array
See also:

numpy.ndarray.argsort

34.11.1.68 pandas.DatetimeIndex.asof

DatetimeIndex.asof(label)

For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.

See also:

get_loc asof is a thin wrapper around get_loc with method='pad'

34.11.1.69 pandas.DatetimeIndex.asof_locs

DatetimeIndex.asof_locs(where, mask)

where : array of timestamps mask : array of booleans where data is not NA

34.11.1.70 pandas.DatetimeIndex.astype

DatetimeIndex.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 ValueError exception is raised.

Parameters dtype : numpy dtype or pandas type

    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.

New in version 0.19.0.

34.11.1.71 pandas.DatetimeIndex.ceil

DatetimeIndex.ceil(freq)

ceil the index to the specified freq

Parameters freq : freq string/object

Returns index of same type

Raises ValueError if the freq cannot be converted

34.11.1.72 pandas.DatetimeIndex.contains

DatetimeIndex.contains(key)

return a boolean if this key is IN the index

Parameters key : object

Returns boolean
34.11.1.73 pandas.DatetimeIndex.copy

DatetimelIndex.copy(name=None, deep=False, dtype=None, **kwargs)

Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters

name : string, optional

depth : boolean, default False

dtype : numpy dtype or pandas type

Returns

copy : Index

Notes

In most cases, there should be no functional difference from using deep, but if deep is passed it will attempt to deepcopy.

34.11.1.74 pandas.DatetimeIndex.delete

DatetimelIndex.delete(loc)

Make a new DatetimeIndex with passed location(s) deleted.

Parameters

loc : int, slice or array of ints

Indicate which sub-arrays to remove.

Returns

new_index : DatetimeIndex

34.11.1.75 pandas.DatetimeIndex.difference

DatetimelIndex.difference(other)

Return a new Index with elements from the index that are not in other.

This is the set difference of two Index objects. It’s sorted if sorting is possible.

Parameters

other : Index or array-like

Returns

difference : Index

Examples

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
```

34.11.1.76 pandas.DatetimeIndex.drop

DatetimelIndex.drop(labels, errors='raise')

Make new Index with passed list of labels deleted

Parameters

labels : array-like

errors : {‘ignore’, ‘raise’}, default ‘raise’
If ‘ignore’, suppress error and existing labels are dropped.

Returns dropped : Index

34.11.1.77 pandas.DatetimeIndex.drop_duplicates

DatetimeIndex.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

34.11.1.78 pandas.DatetimeIndex.dropna

DatetimeIndex.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 valid : Index

34.11.1.79 pandas.DatetimeIndex.duplicated

DatetimeIndex.duplicated(keep='first')
Return boolean np.ndarray denoting duplicate values

Parameters keep : {'first', 'last', False}, default ‘first’
- 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 duplicated : np.ndarray

34.11.1.80 pandas.DatetimeIndex.equals

DatetimeIndex.equals(other)
Determines if two Index objects contain the same elements.

34.11.1.81 pandas.DatetimeIndex.factorize

DatetimeIndex.factorize(sort=False, na_sentinel=-1)
Encode the object as an enumerated type or categorical variable

Parameters sort : boolean, default False
Sort by values
na_sentinel: int, default -1

Value to mark “not found”

Returns labels: the indexer to the original array
uniques: the unique Index

34.11.1.82 pandas.DatetimeIndex.fillna

DatetimeIndex.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 filled: %(klass)s

34.11.1.83 pandas.DatetimeIndex.floor

DatetimeIndex.floor(freq)
floor the index to the specified freq

Parameters freq: freq string/object

Returns index of same type

Raises ValueError if the freq cannot be converted

34.11.1.84 pandas.DatetimeIndex.format

DatetimeIndex.format(name=False, formatter=None, **kwargs)
Render a string representation of the Index

34.11.1.85 pandas.DatetimeIndex.get_duplicates

DatetimeIndex.get_duplicates()

34.11.1.86 pandas.DatetimeIndex.get_indexer

DatetimeIndex.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.

New in version 0.17.0.

New in version 0.21.0: (list-like tolerance)

**Returns**

**indexer**: ndarray of int

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.

**Examples**

```python
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

### 34.11.1.87 `pandas.DatetimeIndex.get_indexer_for`

`DatetimeIndex.get_indexer_for(target, **kwargs)`

guaranteed return of an indexer even when non-unique This dispatches to `get_indexer` or `get_indexer_nonunique` as appropriate

### 34.11.1.88 `pandas.DatetimeIndex.get_indexer_non_unique`

`DatetimeIndex.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**: ndarray of int

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**: ndarray of int

An indexer into the target of the values not found. These correspond to the -1 in the indexer array
34.11.1.89 pandas.DatetimeIndex.get_level_values

DatetimeIndex.get_level_values(level)
Return an Index of values for requested level, 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
self, as there is only one level in the Index.

See also:
pandas.MultiIndex.get_level_values get values for a level of a MultiIndex

34.11.1.90 pandas.DatetimeIndex.get_loc

DatetimeIndex.get_loc(key, method=None, tolerance=None)
Get integer location for requested label

Returns loc : int

34.11.1.91 pandas.DatetimeIndex.get_slice_bound

DatetimeIndex.get_slice_bound(label, side, kind)
Calculate slice bound that corresponds to given label.
Returns leftmost (one-past-the-rightmost if side=='right') position of given label.

Parameters label : object
side : {'left', 'right'}
kind : {'ix', 'loc', 'getitem'}

34.11.1.92 pandas.DatetimeIndex.get_value

DatetimeIndex.get_value(series, key)
Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

34.11.1.93 pandas.DatetimeIndex.get_value_maybe_box

DatetimeIndex.get_value_maybe_box(series, key)

34.11.1.94 pandas.DatetimeIndex.get_values

DatetimeIndex.get_values()
return the underlying data as an ndarray
34.11.1.95 pandas.DatetimeIndex.groupby

DatetimeIndex.groupby(values)
Group the index labels by a given array of values.

Parameters
values : array
Values used to determine the groups.

Returns
groups : dict
{group name -> group labels}

34.11.1.96 pandas.DatetimeIndex.holds_integer

DatetimeIndex.holds_integer()

34.11.1.97 pandas.DatetimeIndex.identical

DatetimeIndex.identical(other)
Similar to equals, but check that other comparable attributes are also equal

34.11.1.98 pandas.DatetimeIndex.indexer_at_time

DatetimeIndex.indexer_at_time(time, asof=False)
Select values at particular time of day (e.g. 9:30AM)

Parameters
time : datetime.time or string

Returns
values_at_time : TimeSeries

34.11.1.99 pandas.DatetimeIndex.indexer_between_time

DatetimeIndex.indexer_between_time(start_time, end_time, include_start=True, include_end=True)
Select values between particular times of day (e.g., 9:00-9:30AM).

Return values of the index between two times. If start_time or end_time are strings then
the pandas.tools.to_time is used to convert to a time object.

Parameters
start_time, end_time : datetime.time, str
datetime.time or 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 : boolean, default True
include_end : boolean, default True

Returns
values_between_time : TimeSeries
34.11.1.100 pandas.DatetimeIndex.insert

**pandas.DatetimeIndex.insert**(loc, item)

Make new Index inserting new item at location

**Parameters**

- **loc**: int
- **item**: object

  if not either a Python datetime or a numpy integer-like, returned Index dtype will be object rather than datetime.

**Returns**

- **new_index**: Index

34.11.1.101 pandas.DatetimeIndex.intersection

**pandas.DatetimeIndex.intersection**(other)

Specialized intersection for DatetimeIndex objects. May be much faster than Index.intersection

**Parameters**

- **other**: DatetimeIndex or array-like

**Returns**

- **y**: Index or DatetimeIndex

34.11.1.102 pandas.DatetimeIndex.is_

**pandas.DatetimeIndex.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**

- **True if both have same underlying data, False otherwise**: bool

34.11.1.103 pandas.DatetimeIndex.is_boolean

**pandas.DatetimeIndex.is_boolean**()

34.11.1.104 pandas.DatetimeIndex.is_categorical

**pandas.DatetimeIndex.is_categorical**()

34.11.1.105 pandas.DatetimeIndex.is_floating

**pandas.DatetimeIndex.is_floating**()

34.11.1.106 pandas.DatetimeIndex.is_integer

**pandas.DatetimeIndex.is_integer**()
34.11.1.107 pandas.DatetimeIndex.is_interval

Datetimex.is_interval()

34.11.1.108 pandas.DatetimeIndex.is_lexsorted_for_tuple

Datetimex.is_lexsorted_for_tuple(tup)

34.11.1.109 pandas.DatetimeIndex.is_mixed

Datetimex.is_mixed()

34.11.1.110 pandas.DatetimeIndex.is_numeric

Datetimex.is_numeric()

34.11.1.111 pandas.DatetimeIndex.is_object

Datetimex.is_object()

34.11.1.112 pandas.DatetimeIndex.is_type_compatible

Datetimex.is_type_compatible(typ)

34.11.1.113 pandas.DatetimeIndex.isin

Datetimex.isin(values)

Compute boolean array of whether each index value is found in the passed set of values

Parameters values: set or sequence of values

Returns is_contained: ndarray (boolean dtype)

34.11.1.114 pandas.DatetimeIndex.isna

Datetimex.isna()

Detect missing values

New in version 0.20.0.

Returns a boolean array of whether my values are NA

See also:

isnull alias of isna
pandas.isna top-level isna
### pandas.DatetimeIndex.isnull

**datetime_index**.isnull()  
Detect missing values  
New in version 0.20.0.  
**Returns** a boolean array of whether my values are NA  
**See also:**

- `isnull` alias of isna  
- `pandas.isna` top-level isna

### pandas.DatetimeIndex.item

**datetime_index**.item()  
return the first element of the underlying data as a python scalar

### pandas.DatetimeIndex.join

**datetime_index**.join(other, how='left', level=None, return_indexers=False, sort=False)  
See Index.join

### pandas.DatetimeIndex.map

**datetime_index**.map(f)

### pandas.DatetimeIndex.max

**datetime_index**.max(axis=None, *args, **kwargs)  
Return the maximum value of the Index or maximum along an axis.  
**See also:**

- `numpy.ndarray.max`

### pandas.DatetimeIndex.memory_usage

**datetime_index**.memory_usage(deep=False)  
Memory usage of my values  
**Parameters**  
- `deep` : bool  
  Introspect the data deeply, interrogate object dtypes for system-level memory consumption  
**Returns** bytes used  
**See also:**

- `numpy.ndarray.nbytes`
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

34.11.1.121 pandas.DatetimeIndex.min

```python
datetime_index.min(axis=None, *args, **kwargs)
```

Return the minimum value of the Index or minimum along an axis.

See also:

- `numpy.ndarray.min`

34.11.1.122 pandas.DatetimeIndex.normalize

```python
datetime_index.normalize()
```

Return DatetimeIndex with times to midnight. Length is unaltered

Returns

- `normalized`: DatetimeIndex

34.11.1.123 pandas.DatetimeIndex.notna

```python
datetime_index.notna()
```

Inverse of `isna`

New in version 0.20.0.

Returns

- a boolean array of whether my values are not NA

See also:

- `notnull`: alias of `notna`
- `pandas.notna`: top-level `notna`

34.11.1.124 pandas.DatetimeIndex.notnull

```python
datetime_index.notnull()
```

Inverse of `isna`

New in version 0.20.0.

Returns

- a boolean array of whether my values are not NA

See also:

- `notnull`: alias of `notna`
- `pandas.notna`: top-level `notna`
34.11.1.125 pandas.DatetimeIndex.nunique

DatetimeIndex.nunique(dropna=True)
Return number of unique elements in the object.
Excludes NA values by default.

Parameters dropna: boolean, default True
Don’t include NaN in the count.

Returns nunique: int

34.11.1.126 pandas.DatetimeIndex.putmask

DatetimeIndex.putmask(mask, value)
return a new Index of the values set with the mask

See also:
numpy.ndarray.putmask

34.11.1.127 pandas.DatetimeIndex.ravel

DatetimeIndex.ravel(order='C')
return an ndarray of the flattened values of the underlying data

See also:
numpy.ndarray.ravel

34.11.1.128 pandas.DatetimeIndex.reindex

DatetimeIndex.reindex(target, method=None, level=None, limit=None, tolerance=None)
Create index with target’s values (move/add/delete values as necessary)

Parameters target: an iterable

Returns new_index: pd.Index
Resulting index

indexer: np.ndarray or None
Indices of output values in original index

34.11.1.129 pandas.DatetimeIndex.rename

DatetimeIndex.rename(name, inplace=False)
Set new names on index. Defaults to returning new index.

Parameters name: str or list
name to set

inplace: bool
if True, mutates in place

Returns new index (of same type and class...etc) [if inplace, returns None]
34.11.1.130 pandas.DatetimeIndex.repeat

DatetimeIndex.repeat(repeats, *args, **kwargs)
Analogous to ndarray.repeat

34.11.1.131 pandas.DatetimeIndex.reshape

DatetimeIndex.reshape(*args, **kwargs)
NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.
Reshape an Index.

34.11.1.132 pandas.DatetimeIndex.round

DatetimeIndex.round(freq, *args, **kwargs)
round the index to the specified freq

Parameters freq : freq string/object
Returns index of same type
Raises ValueError if the freq cannot be converted

34.11.1.133 pandas.DatetimeIndex.searchsorted

DatetimeIndex.searchsorted(value, side='left', sorter=None)
Find indices where elements should be inserted to maintain order.
Find the indices into a sorted DatetimeIndex self such that, if the corresponding elements in value were inserted before the indices, the order of self would be preserved.

Parameters value : array_like
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 indices : array of ints
Array of insertion points with the same shape as value.

See also:
numpy.searchsorted

Notes

Binary search is used to find the required insertion points.
Examples

```python
>>> x = pd.Series([1, 2, 3])
>>> x
0 1
1 2
2 3
dtype: int64

>>> x.searchsorted(4)
array([3])

>>> x.searchsorted([0, 4])
array([0, 3])

>>> x.searchsorted([1, 3], side='left')
array([0, 2])

>>> x.searchsorted([1, 3], side='right')
array([1, 3])

>>> x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'])
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]

>>> x.searchsorted('bread')
array([1])  # Note: an array, not a scalar

>>> x.searchsorted(['bread'])
array([1])

>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])

>>> x.searchsorted(['bread', 'eggs'], side='right')
array([3, 4])  # eggs before milk
```

34.11.1.134 pandas.DatetimeIndex.set_names

`DatetimeIndex.set_names(names, level=None, inplace=False)`

Set new names on index. Defaults to returning new index.

**Parameters**

- `names`: str or sequence
  - name(s) to set
- `level`: int, level name, or sequence of int/level names (default None)
  - If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels).
  - Otherwise level must be None
- `inplace`: bool
  - if True, mutates in place
Returns new index (of same type and class...etc) [if inplace, returns None]

Examples

```python
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64')
>>> Index([1, 2, 3, 4]).set_names(['foo'])
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')],
                               names=['foo', 'bar'])
>>> idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
          labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
          names=[u'baz', u'quz'])
>>> idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
          labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
          names=[u'baz', u'bar'])
```

34.11.1.135 pandas.DatetimeIndex.set_value

DatetimIndex.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

34.11.1.136 pandas.DatetimeIndex.shift

DatetimIndex.shift(n, freq=None)

Specialized shift which produces a DatetimeIndex

Parameters

- **n**: int
  Periods to shift by

- **freq**: DateOffset or timedelta-like, optional

Returns

- **shifted**: DatetimeIndex

34.11.1.137 pandas.DatetimeIndex.slice_indexer

DatetimIndex.slice_indexer(start=None, end=None, step=None, kind=None)

Return indexer for specified label slice. Index.slice_indexer, customized to handle time slicing.

In addition to functionality provided by Index.slice_indexer, does the following:

- if both `start` and `end` are instances of `datetime.time`, it invokes `indexer_between_time`
- if `start` and `end` are both either string or None perform value-based selection in non-monotonic cases.

34.11.1.138 pandas.DatetimeIndex.slice_locs

DatetimIndex.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 : {'ix', 'loc', 'getitem'} or None
```

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

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

34.11.1.139 pandas.DatetimeIndex.snap

```
DatetimeIndex.snap(freq='S')
Snap time stamps to nearest occurring frequency
```

34.11.1.140 pandas.DatetimeIndex.sort

```
DatetimeIndex.sort(*args, **kwargs)
```

34.11.1.141 pandas.DatetimeIndex.sort_values

```
DatetimeIndex.sort_values(return_indexer=False, ascending=True)
Return sorted copy of Index
```

34.11.1.142 pandas.DatetimeIndex.sortlevel

```
DatetimeIndex.sortlevel(level=None, ascending=True, sort_remaining=None)
For internal compatibility with with the Index API
Sort the Index. This is for compat with MultiIndex
```

Parameters  
```
ascending : boolean, default True
False to sort in descending order
level, sort_remaining are compat parameters
```
Returns `sorted_index` : Index

### 34.11.1.143 pandas.DatetimeIndex.str

**DatetimeIndex.**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.str.split('_')
>>> s.str.replace('_', '')
```

### 34.11.1.144 pandas.DatetimeIndex.strftime

**DatetimeIndex.**strftime(\textit{date\_format})

Return an array 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

New in version 0.17.0.

**Parameters**

\textit{date\_format} : str

date format string (e.g. “%Y-%m-%d”)

**Returns**

ndarray of formatted strings

### 34.11.1.145 pandas.DatetimeIndex.summary

**DatetimeIndex.**summary(\textit{name}=None)

Return a summarized representation

### 34.11.1.146 pandas.DatetimeIndex.symmetric_difference

**DatetimeIndex.**symmetric_difference(\textit{other}, result\_name=None)

Compute the symmetric difference of two Index objects. It’s sorted if sorting is possible.

**Parameters**

\textit{other} : Index or array-like

\textit{result\_name} : str

**Returns**

\textit{symmetric\_difference} : Index

**Notes**

\textit{symmetric\_difference} contains elements that appear in either \textit{idx1} or \textit{idx2} but not both. Equivalent to the Index created by \textit{idx1.difference(idx2)} \texttt{|} \textit{idx2.difference(idx1)} with duplicates dropped.
Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

34.11.1.147 pandas.DatetimeIndex.take

```
DatetimeIndex.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*: list

Indices to be taken

*axis*: int, optional

The axis over which to select values, always 0.

*allow_fill*: bool, default True

*fill_value*: bool, default None

If allow_fill=True and fill_value is not None, indices specified by -1 is regarded as NA. If Index doesn’t hold NA, raise ValueError

**See also:**

numpy.ndarray.take

34.11.1.148 pandas.DatetimeIndex.to_datetime

```
DatetimeIndex.to_datetime(dayfirst=False)
```

34.11.1.149 pandas.DatetimeIndex.to_frame

```
DatetimeIndex.to_frame(index=True)
```

Create a DataFrame with a column containing the Index.

New in version 0.21.0.

**Parameters**

*index*: boolean, default True

Set the index of the returned DataFrame as the original Index.

**Returns**

*DataFrame*: a DataFrame containing the original Index data.
34.11.1.150 pandas.DatetimeIndex.to_julian_date

DatetimeIndex.to_julian_date()
Convert DatetimeIndex to Float64Index of Julian Dates. 0 Julian date is noon January 1, 4713 BC. http://en.wikipedia.org/wiki/Julian_day

34.11.1.151 pandas.DatetimeIndex.to_native_types

DatetimeIndex.to_native_types(slicer=None, **kwargs)
Format specified values of self and return them.

Parameters slicer : int, array-like
An indexer into self that specifies which values are used in the formatting process.

kwargs : 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

34.11.1.152 pandas.DatetimeIndex.to_period

DatetimeIndex.to_period(freq=None)
Cast to PeriodIndex at a particular frequency

34.11.1.153 pandas.DatetimeIndex.to_perioddelta

DatetimeIndex.to_perioddelta(freq)
Calculates TimedeltaIndex of difference between index values and index converted to PeriodIndex at specified freq. Used for vectorized offsets

Parameters freq : Period frequency
Returns y : TimedeltaIndex

34.11.1.154 pandas.DatetimeIndex.to_pydatetime

DatetimeIndex.to_pydatetime()
Return DatetimeIndex as object ndarray of datetime.datetime objects

Returns datetimes : ndarray

34.11.1.155 pandas.DatetimeIndex.to_series

DatetimeIndex.to_series(keep_tz=False)
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 False.

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.

**Returns** Series

### 34.11.1.156 pandas.DatetimeIndex.tolist

DatetimeIndex.tolist()

return a list of the underlying data

### 34.11.1.157 pandas.DatetimeIndex.transpose

DatetimeIndex.transpose(*args, **kwargs)

return the transpose, which is by definition self

### 34.11.1.158 pandas.DatetimeIndex.tz_convert

DatetimeIndex.tz\_convert(tz)

Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)

**Parameters** tz : string, pytz.timezone, dateutil.tzfile or None

Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding UTC time.

**Returns** normalized : DatetimeIndex

**Raises** TypeError

If DatetimeIndex is tz-naive.

### 34.11.1.159 pandas.DatetimeIndex.tz\_localize

DatetimeIndex.tz\_localize(tz, ambiguous='raise', errors='raise')

Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

**Parameters** tz : string, pytz.timezone, dateutil.tzfile or None

Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding local time.

**ambiguos** : ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’

- ‘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

`errors` : ‘raise’, ‘coerce’, default ‘raise’

- ‘raise’ will raise a NonExistentTimeError if a timestamp is not valid in the specified timezone (e.g. due to a transition from or to DST time)

- ‘coerce’ will return NaT if the timestamp can not be converted into the specified timezone

New in version 0.19.0.

`infer_dst` : boolean, default False

Deprecated since version 0.15.0: Attempt to infer fall dst-transition hours based on order

Returns `localized` : DatetimeIndex

Raises `TypeError`

If the DatetimeIndex is tz-aware and tz is not None.

### 34.11.1.160 pandas.DatetimeIndex.union

`DatetimeIndex.union(other)`  
Specialized union for DatetimeIndex objects. If combine overlapping ranges with the same DateOffset, will be much faster than Index.union

**Parameters** `other` : DatetimeIndex or array-like

**Returns** `y` : Index or DatetimeIndex

### 34.11.1.161 pandas.DatetimeIndex.union_many

`DatetimeIndex.union_many(others)`  
A bit of a hack to accelerate unioning a collection of indexes

### 34.11.1.162 pandas.DatetimeIndex.unique

`DatetimeIndex.unique()`  
Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

**Parameters** `values` : 1d array-like

**Returns** unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

See also:

`unique, Index.unique, Series.unique`
34.11.1.163 pandas.DatetimeIndex.value_counts

```
DatetimeIndex.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)
```

Returns object 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**: boolean, default False
  - If True then the object returned will contain the relative frequencies of the unique values.
- **sort**: boolean, default True
  - Sort by values
- **ascending**: boolean, default False
  - Sort in ascending order
- **bins**: integer, optional
  - Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data
- **dropna**: boolean, default True
  - Don’t include counts of NaN.

**Returns**

- **counts**: Series

34.11.1.164 pandas.DatetimeIndex.view

```
DatetimeIndex.view(cls=None)
```

34.11.1.165 pandas.DatetimeIndex.where

```
DatetimeIndex.where(cond, other=None)
```

New in version 0.19.0.

Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

**Parameters**

- **cond**: boolean array-like with the same length as self
- **other**: scalar, or array-like

34.11.2 Time/Date Components

<table>
<thead>
<tr>
<th>DatetimeIndex.year</th>
<th>The year of the datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.month</td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td>DatetimeIndex.day</td>
<td>The days of the datetime</td>
</tr>
<tr>
<td>DatetimeIndex.hour</td>
<td>The hours of the datetime</td>
</tr>
<tr>
<td>DatetimeIndex.minute</td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td>DatetimeIndex.second</td>
<td>The seconds of the datetime</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.111 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.microsecond</td>
<td>The microseconds of the datetime</td>
</tr>
<tr>
<td>DatetimeIndex.nanosecond</td>
<td>The nanoseconds of the datetime</td>
</tr>
<tr>
<td>DatetimeIndex.date</td>
<td>Returns numpy array of python <code>datetime.datetime</code> objects (namely, the date part of Timestamps without timezone information).</td>
</tr>
<tr>
<td>DatetimeIndex.time</td>
<td>Returns numpy array of <code>datetime.time</code>.</td>
</tr>
<tr>
<td>DatetimeIndex.dayofyear</td>
<td>The ordinal day of the year</td>
</tr>
<tr>
<td>DatetimeIndex.weekofyear</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>DatetimeIndex.week</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>DatetimeIndex.dayofweek</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>DatetimeIndex.weekday</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>DatetimeIndex.weekday_name</td>
<td>The name of day in a week (ex: Friday)</td>
</tr>
<tr>
<td>DatetimeIndex.quarter</td>
<td>The quarter of the date</td>
</tr>
<tr>
<td>DatetimeIndex.tz</td>
<td>get/set the frequency of the Index</td>
</tr>
<tr>
<td>DatetimeIndex.freqstr</td>
<td>Return the frequency object as a string if its set, otherwise None</td>
</tr>
<tr>
<td>DatetimeIndex.is_month_start</td>
<td>Logical indicating if first day of month (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_month_end</td>
<td>Logical indicating if last day of month (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_quarter_start</td>
<td>Logical indicating if first day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_quarter_end</td>
<td>Logical indicating if last day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_year_start</td>
<td>Logical indicating if first day of year (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_year_end</td>
<td>Logical indicating if last day of year (defined by frequency)</td>
</tr>
<tr>
<td>DatetimeIndex.is_leap_year</td>
<td>Logical indicating if the date belongs to a leap year</td>
</tr>
<tr>
<td>DatetimeIndex.inferred_freq</td>
<td></td>
</tr>
</tbody>
</table>

34.11.3 Selecting

- **DatetimeIndex.indexer_at_time**(time[, asof]) Select values at particular time of day (e.g., `9:00-9:30AM`).
- **DatetimeIndex.indexer_between_time**(...[, ...]) Select values between particular times of day (e.g., `9:00-9:30AM`).

34.11.4 Time-specific operations

- **DatetimeIndex.normalize()** Return DatetimeIndex with times to midnight.
- **DatetimeIndex.strftime**(date_format) Return an array of formatted strings specified by date_format, which supports the same string format as the python standard library.
- **DatetimeIndex.snap**(freq) Snap time stamps to nearest occurring frequency
- **DatetimeIndex.tz_convert**(tz) Convert tz-aware DatetimeIndex from one time zone to another (using
- **DatetimeIndex.tz_localize**(tz[, ambiguous, ...]) Localize tz-naive DatetimeIndex to given time zone (using
- **DatetimeIndex.round**(freq, *args, **kwargs) round the index to the specified freq

Continued on next page
Table 34.113 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.floor</td>
<td>floor the index to the specified freq</td>
</tr>
<tr>
<td>DatetimeIndex.ceil</td>
<td>ceil the index to the specified freq</td>
</tr>
</tbody>
</table>

### 34.11.5 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.to_datetime([dayfirst])</td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>DatetimeIndex.to_period([freq])</td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>DatetimeIndex.to_perioddelta(freq)</td>
<td>Calculates TimedeltaIndex of difference between index values and index converted to PeriodIndex at specified freq.</td>
</tr>
<tr>
<td>DatetimeIndex.to_pydatetime()</td>
<td>Return DatetimeIndex as object ndarray of datetime.datetime objects</td>
</tr>
<tr>
<td>DatetimeIndex.to_series([keep_tz])</td>
<td>Create a Series with both index and values equal to the index keys</td>
</tr>
<tr>
<td>DatetimeIndex.to_frame([index])</td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
</tbody>
</table>

### 34.12 TimedeltaIndex

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimedeltaIndex</td>
<td>Immutable ndarray of timedelta64 data, represented internally as int64, and</td>
</tr>
</tbody>
</table>

#### 34.12.1 pandas.TimedeltaIndex

**class pandas.TimedeltaIndex**

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**: a frequency for the index, optional

- **copy**: bool
  
  Make a copy of input ndarray

- **start**: starting value, timedelta-like, optional
  
  If data is None, start is used as the start point in generating regular timedelta data.

- **periods**: int, optional, > 0
  
  Number of periods to generate, if generating index. Takes precedence over end argument

- **end**: end time, timedelta-like, optional
  
  If periods is none, generated index will extend to first conforming time on or just past end argument

- **closed**: string or None, default None
Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

```
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

**Notes**

To learn more about the frequency strings, please see this link.

**Attributes**

<table>
<thead>
<tr>
<th>Method</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>as18</code></td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td><code>asobject</code></td>
<td>return object Index which contains boxed values</td>
</tr>
<tr>
<td><code>base</code></td>
<td>return the base object if the memory of the underlying data is</td>
</tr>
<tr>
<td><code>components</code></td>
<td>Return a dataframe of the components (days, hours, minutes, seconds,</td>
</tr>
<tr>
<td></td>
<td>milliseconds, microseconds, nanoseconds) of the Timedeltas.</td>
</tr>
<tr>
<td><code>data</code></td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td><code>days</code></td>
<td>Number of days for each element.</td>
</tr>
<tr>
<td><code>dtype</code></td>
<td></td>
</tr>
<tr>
<td><code>dtype_str</code></td>
<td></td>
</tr>
<tr>
<td><code>empty</code></td>
<td></td>
</tr>
<tr>
<td><code>flags</code></td>
<td></td>
</tr>
<tr>
<td><code>freq</code></td>
<td></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>has_duplicates</code></td>
<td></td>
</tr>
<tr>
<td><code>hash</code></td>
<td></td>
</tr>
<tr>
<td><code>inferred_freq</code></td>
<td></td>
</tr>
<tr>
<td><code>inferred_type</code></td>
<td></td>
</tr>
<tr>
<td><code>is_all_dates</code></td>
<td>alias for is_monotonic_increasing (deprecated)</td>
</tr>
<tr>
<td><code>is_monotonic</code></td>
<td>return if the index is monotonic decreasing (only equal or</td>
</tr>
<tr>
<td><code>is_monotonic_decreasing</code></td>
<td></td>
</tr>
<tr>
<td><code>is_monotonic_increasing</code></td>
<td></td>
</tr>
<tr>
<td><code>is_unique</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.116 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itemsize</code></td>
<td>Return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td><code>microseconds</code></td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td><code>name</code></td>
<td></td>
</tr>
<tr>
<td><code>names</code></td>
<td></td>
</tr>
<tr>
<td><code>nanoseconds</code></td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</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>Return the number of dimensions of the underlying data</td>
</tr>
<tr>
<td><code>nlevels</code></td>
<td></td>
</tr>
<tr>
<td><code>resolution</code></td>
<td></td>
</tr>
<tr>
<td><code>seconds</code></td>
<td>Number of seconds (&gt;= 0 and less than 1 day) for each element.</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>strides</code></td>
<td>Return the strides of the underlying data</td>
</tr>
<tr>
<td><code>values</code></td>
<td>Return the underlying data as an ndarray</td>
</tr>
</tbody>
</table>

### 34.12.1.1 pandas.TimedeltaIndex.T

`TimedeltaIndex.T`

return the transpose, which is by definition self

### 34.12.1.2 pandas.TimedeltaIndex.asi8

`TimedeltaIndex.asi8`

### 34.12.1.3 pandas.TimedeltaIndex.asobject

`TimedeltaIndex.asobject`

return object Index which contains boxed values

this is an internal non-public method

### 34.12.1.4 pandas.TimedeltaIndex.base

`TimedeltaIndex.base`

return the base object if the memory of the underlying data is shared

### 34.12.1.5 pandas.TimedeltaIndex.components

`TimedeltaIndex.components`

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

**Returns** a DataFrame
34.12.1.6 pandas.TimedeltaIndex.data

TimedeltaIndex.data
to return the data pointer of the underlying data.

34.12.1.7 pandas.TimedeltaIndex.days

TimedeltaIndex.days
Number of days for each element.

34.12.1.8 pandas.TimedeltaIndex.dtype

TimedeltaIndex.dtype

34.12.1.9 pandas.TimedeltaIndex.dtype_str

TimedeltaIndex.dtype_str = None

34.12.1.10 pandas.TimedeltaIndex.empty

TimedeltaIndex.empty

34.12.1.11 pandas.TimedeltaIndex.flags

TimedeltaIndex.flags

34.12.1.12 pandas.TimedeltaIndex.freq

TimedeltaIndex.freq = None

34.12.1.13 pandas.TimedeltaIndex.freqstr

TimedeltaIndex.freqstr
Return the frequency object as a string if its set, otherwise None

34.12.1.14 pandas.TimedeltaIndex.has_duplicates

TimedeltaIndex.has_duplicates

34.12.1.15 pandas.TimedeltaIndex.hasnans

TimedeltaIndex.hasnans = None

34.12.1.16 pandas.TimedeltaIndex.inferred_freq

TimedeltaIndex.inferred_freq = None

34.12. TimedeltaIndex
34.12.1.17 pandas.TimedeltaIndex.inferred_type

TimedeltaIndex.inferred_type

34.12.1.18 pandas.TimedeltaIndex.is_all_dates

TimedeltaIndex.is_all_dates

34.12.1.19 pandas.TimedeltaIndex.is_monotonic

TimedeltaIndex.is_monotonic
alias for is_monotonic_increasing (deprecated)

34.12.1.20 pandas.TimedeltaIndex.is_monotonic_decreasing

TimedeltaIndex.is_monotonic_decreasing
return if the index is monotonic decreasing (only equal or decreasing) values.

Examples

```python
g>>> Index([3, 2, 1]).is_monotonic_decreasing
True
g>>> Index([3, 2, 2]).is_monotonic_decreasing
True
g>>> Index([3, 1, 2]).is_monotonic_decreasing
False
```

34.12.1.21 pandas.TimedeltaIndex.is_monotonic_increasing

TimedeltaIndex.is_monotonic_increasing
return if the index is monotonic increasing (only equal or increasing) values.

Examples

```python
g>>> Index([1, 2, 3]).is_monotonic_increasing
True
g>>> Index([1, 2, 2]).is_monotonic_increasing
True
g>>> Index([1, 3, 2]).is_monotonic_increasing
False
```

34.12.1.22 pandas.TimedeltaIndex.is_unique

TimedeltaIndex.is_unique = None
34.12.1.23 pandas.TimedeltaIndex.itemsize

TimedeltaIndex.itemsize
   return the size of the dtype of the item of the underlying data

34.12.1.24 pandas.TimedeltaIndex.microseconds

TimedeltaIndex.microseconds
   Number of microseconds (>= 0 and less than 1 second) for each element.

34.12.1.25 pandas.TimedeltaIndex.name

TimedeltaIndex.name = None

34.12.1.26 pandas.TimedeltaIndex.names

TimedeltaIndex.names

34.12.1.27 pandas.TimedeltaIndex.nanoseconds

TimedeltaIndex.nanoseconds
   Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

34.12.1.28 pandas.TimedeltaIndex.nbytes

TimedeltaIndex.nbytes
   return the number of bytes in the underlying data

34.12.1.29 pandas.TimedeltaIndex.ndim

TimedeltaIndex.ndim
   return the number of dimensions of the underlying data, by definition 1

34.12.1.30 pandas.TimedeltaIndex.nlevels

TimedeltaIndex.nlevels

34.12.1.31 pandas.TimedeltaIndex.resolution

TimedeltaIndex.resolution = None

34.12.1.32 pandas.TimedeltaIndex.seconds

TimedeltaIndex.seconds
   Number of seconds (>= 0 and less than 1 day) for each element.
34.12.1.33 pandas.TimedeltaIndex.shape

TimedeltaIndex.shape

return a tuple of the shape of the underlying data

34.12.1.34 pandas.TimedeltaIndex.size

TimedeltaIndex.size

return the number of elements in the underlying data

34.12.1.35 pandas.TimedeltaIndex.strides

TimedeltaIndex.strides

return the strides of the underlying data

34.12.1.36 pandas.TimedeltaIndex.values

TimedeltaIndex.values

return the underlying data as an ndarray

Methods

all([other])

any([other])

append(other) Append a collection of Index options together

argmax([axis]) Returns the indices of the maximum values along an axis.

argmin([axis]) Returns the indices of the minimum values along an axis.

argsort(*args, **kwargs) Returns the indices that would sort the index and its underlying data.

asof(label) For a sorted index, return the most recent label up to and including the passed label.

asof_locs(where, mask) where : array of timestamps

copy([name, deep, dtype]) Make a copy of this object.

copy([name, deep, dtype]) Make a copy of this object.

difference(other) Return a new Index with elements from the index that are not in other.

drop(labels[, errors]) Make new Index with passed list of labels deleted

drop_duplicates([keep]) Return Index with duplicate values removed

dropna([how]) Return Index without NA/NaN values

duplicated([keep]) Return boolean np.ndarray denoting duplicate values

equals(other) Determines if two Index objects contain the same elements.
Table 34.117 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>factorize([sort, na_sentinel])</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>floor(freq)</code></td>
<td>Floor the index to the specified freq</td>
</tr>
<tr>
<td><code>format([name, formatter])</code></td>
<td>Render a string representation of the Index</td>
</tr>
<tr>
<td><code>get_duplicates()</code></td>
<td></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, **kwargs)</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, equal to the length of the index.</td>
</tr>
<tr>
<td><code>get_loc(key[, method, tolerance])</code></td>
<td>Get integer location 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>get_value_maybe_box(series, key)</code></td>
<td></td>
</tr>
<tr>
<td><code>get_values()</code></td>
<td>Return the underlying data as an 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>hold_integer()</code></td>
<td></td>
</tr>
<tr>
<td><code>identical(other)</code></td>
<td>Similar to equals, but check that other comparable attributes are</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)</code></td>
<td>Specialized intersection for TimedeltaIndex 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></td>
</tr>
<tr>
<td><code>is_categorical()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_floating()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_integer()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_interval()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_lexsorted_for_tuple(tup)</code></td>
<td></td>
</tr>
<tr>
<td><code>is_mixed()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_numeric()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_object()</code></td>
<td></td>
</tr>
<tr>
<td><code>is_type_compatible(typ)</code></td>
<td></td>
</tr>
<tr>
<td><code>isin(values)</code></td>
<td>Compute boolean array of whether each index value is found in the</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</td>
</tr>
<tr>
<td><code>join(other[, how, level, return_indexers, sort])</code></td>
<td>See Index.join</td>
</tr>
<tr>
<td><code>map(f)</code></td>
<td></td>
</tr>
<tr>
<td><code>max([axis])</code></td>
<td>Return the maximum value of the Index or maximum along an axis.</td>
</tr>
<tr>
<td><code>memory_usage([deep])</code></td>
<td>Memory usage of my values</td>
</tr>
<tr>
<td><code>min([axis])</code></td>
<td>Return the minimum value of the Index or minimum along an axis.</td>
</tr>
<tr>
<td><code>notna()</code></td>
<td>Inverse of isna</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34.117 – continued from previous page

- **notnull()**: Inverse of isna
- **nunique([dropna])**: Return number of unique elements in the object.
- **putmask(mask, value)**: return a new Index of the values set with the mask
- **ravel([order])**: return an ndarray of the flattened values of the underlying data
- **reindex(target[, method, level, limit, ...])**: Create index with target’s values (move/add/delete values as necessary)
- **rename(name[, inplace])**: Set new names on index.
- **repeat(repeats, *args, **kwargs)**: Analogous to ndarray.repeat
- **reshape(*args, **kwargs)**: NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.
- **round(freq, *args, **kwargs)**: round the index to the specified freq
- **searchsorted(value[, side, sorter])**: Find indices where elements should be inserted to maintain order.
- **set_names(names[, level, inplace])**: Set new names on index.
- **set_value(arr, key, value)**: Fast lookup of value from 1-dimensional ndarray.
- **shift(n[, freq])**: Specialized shift which produces a DatetimeIndex
- **slice_indexer([start, end, step, kind])**: For an ordered Index, compute the slice indexer for input labels and
- **slice_locs([start, end, step, kind])**: Compute slice locations for input labels.
- **sort(*args, **kwargs)**
- **sort_values([return_indexer, ascending])**: Return sorted copy of Index
- **sortlevel([level, ascending, sort_remaining])**: For internal compatibility with with the Index API
- **str**
- **summary([name])**
- **symmetric_difference(other[, result_name])**: Compute the symmetric difference of two Index objects.
- **take(indices[, axis, allow_fill, fill_value])**: return a new Index of the values selected by the indices
- **to_datetime([dayfirst])**
- **to_frame(index)**: Create a DataFrame with a column containing the Index.
- **to_native_types([slicer])**: Format specified values of self and return them.
- **to_pytimedelta()**: Return TimedeltaIndex as object ndarray of datetime.timedelta objects
- **to_series(**kwargs)**: Create a Series with both index and values equal to the index keys
- **tolist()**: return a list of the underlying data
- **total_seconds()**: Total duration of each element expressed in seconds.
- **transpose(*args, **kwargs)**: return the transpose, which is by definition self
- **union(other)**
- **unique()**: Return unique values in the object.
- **value_counts([normalize, sort, ascending, ...])**: Returns object containing counts of unique values.
- **view([cls])**
- **where(cond[, other])**: New in version 0.19.0.

### 34.12.1.37 pandas.TimedeltaIndex.all

TimedeltaIndex.all(\texttt{other=None})
34.12.1.38 pandas.TimedeltaIndex.any

TimedeltaIndex.any(other=None)

34.12.1.39 pandas.TimedeltaIndex.append

TimedeltaIndex.append(other)
  Append a collection of Index options together
  Parameters other : Index or list/tuple of indices
  Returns appended : Index

34.12.1.40 pandas.TimedeltaIndex.argmax

TimedeltaIndex.argmax(axis=None, *args, **kwargs)
  Returns the indices of the maximum values along an axis. See numpy.ndarray.argmax for more information on the axis parameter.
  See also:
  numpy.ndarray.argmax

34.12.1.41 pandas.TimedeltaIndex.argmin

TimedeltaIndex.argmin(axis=None, *args, **kwargs)
  Returns the indices of the minimum values along an axis. See numpy.ndarray.argmin for more information on the axis parameter.
  See also:
  numpy.ndarray.argmin

34.12.1.42 pandas.TimedeltaIndex.argsort

TimedeltaIndex.argsort(*args, **kwargs)
  Returns the indices that would sort the index and its underlying data.
  Returns argsorted : numpy array
  See also:
  numpy.ndarray.argsort

34.12.1.43 pandas.TimedeltaIndex.asof

TimedeltaIndex.asof(label)
  For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.
  See also:
  get_loc asof is a thin wrapper around get_loc with method='pad'
34.12.1.44 pandas.TimedeltaIndex.asof_locs

TimedeltaIndex.asof_locs(where, mask)
where : array of timestamps
mask : array of booleans where data is not NA

34.12.1.45 pandas.TimedeltaIndex.astype

TimedeltaIndex.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 ValueError exception is raised.

Parameters
dtype : numpy dtype or pandas type
    
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.

New in version 0.19.0.

34.12.1.46 pandas.TimedeltaIndex.ceil

TimedeltaIndex.ceil(freq)
ceil the index to the specified freq

Parameters
freq : freq string/object

Returns
index of same type

Raises
ValueError if the freq cannot be converted

34.12.1.47 pandas.TimedeltaIndex.contains

TimedeltaIndex.contains(key)
return a boolean if this key is IN the index

Parameters
key : object

Returns
boolean

34.12.1.48 pandas.TimedeltaIndex.copy

TimedeltaIndex.copy(name=None, deep=False, dtype=None, **kwargs)
Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters
name : string, optional
    
deep : boolean, default False
    
dtype : numpy dtype or pandas type

Returns
copy : Index
Notes

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy.

34.12.1.49 pandas.TimedeltaIndex.delete

TimedeltaIndex.delete(loc)
Make a new DatetimeIndex with passed location(s) deleted.

Parameters  loc: int, slice or array of ints
Indicate which sub-arrays to remove.

Returns  new_index : TimedeltaIndex

34.12.1.50 pandas.TimedeltaIndex.difference

TimedeltaIndex.difference(other)
Return a new Index with elements from the index that are not in `other`.
This is the set difference of two Index objects. It’s sorted if sorting is possible.

Parameters  other : Index or array-like

Returns  difference : Index

Examples

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
```

34.12.1.51 pandas.TimedeltaIndex.drop

TimedeltaIndex.drop(labels, errors='raise')
Make new Index with passed list of labels deleted

Parameters  labels : array-like
errors : {'ignore', 'raise'}, default 'raise'
If ‘ignore’, suppress error and existing labels are dropped.

Returns  dropped : Index

34.12.1.52 pandas.TimedeltaIndex.drop_duplicates

TimedeltaIndex.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

34.12.1.53 pandas.TimedeltaIndex.dropna
TimedeltaIndex.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 valid: Index

34.12.1.54 pandas.TimedeltaIndex.duplicated
TimedeltaIndex.duplicated(keep='first')
Return boolean np.ndarray denoting duplicate values

Parameters keep: {'first', 'last', False}, default 'first'
• 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 duplicated: np.ndarray

34.12.1.55 pandas.TimedeltaIndex.equals
TimedeltaIndex.equals(other)
Determines if two Index objects contain the same elements.

34.12.1.56 pandas.TimedeltaIndex.factorize
TimedeltaIndex.factorize(sort=False, na_sentinel=-1)
Encode the object as an enumerated type or categorical variable

Parameters sort: boolean, default False
Sort by values

na_sentinel: int, default -1
Value to mark “not found”

Returns labels: the indexer to the original array
uniques: the unique Index
34.12.1.57 pandas.TimedeltaIndex.fillna

TimedeltaIndex.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 filled : %(klass)s

34.12.1.58 pandas.TimedeltaIndex.floor

TimedeltaIndex.floor(freq)
floor the index to the specified freq

Parameters freq : freq string/object

Returns index of same type

Raises ValueError if the freq cannot be converted

34.12.1.59 pandas.TimedeltaIndex.format

TimedeltaIndex.format(name=False, formatter=None, **kwargs)
Render a string representation of the Index

34.12.1.60 pandas.TimedeltaIndex.get_duplicates

TimedeltaIndex.get_duplicates()

34.12.1.61 pandas.TimedeltaIndex.get_indexer

TimedeltaIndex.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 most satisfy the equation

\[
\text{abs(index[indexer]} - \text{target}) \leq \text{tolerance}.
\]

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.

New in version 0.17.0.

New in version 0.21.0: (list-like tolerance)

**Returns**

- **indexer**: ndarray of int

  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.

**Examples**

```python
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

### 34.12.1.62 pandas.TimedeltaIndex.get_indexer_for

`TimedeltaIndex.get_indexer_for(target, **kwargs)`

Guaranteed return of an indexer even when non-unique. This dispatches to `get_indexer` or `get_indexer_non_unique` as appropriate.

### 34.12.1.63 pandas.TimedeltaIndex.get_indexer_non_unique

`TimedeltaIndex.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**: ndarray of int

  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**: ndarray of int

  An indexer into the target of the values not found. These correspond to the -1 in the indexer array.

### 34.12.1.64 pandas.TimedeltaIndex.get_level_values

`TimedeltaIndex.get_level_values(level)`

Return an Index of values for requested level, 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

self, as there is only one level in the Index.

**See also**:

*pandas.MultiIndex.get_level_values* get values for a level of a MultiIndex

### 34.12.1.65 pandas.TimedeltaIndex.get_loc

*TimedeltaIndex.get_loc(key, method=None, tolerance=None)*

Get integer location for requested label

**Returns** loc : int

### 34.12.1.66 pandas.TimedeltaIndex.get_slice_bound

*TimedeltaIndex.get_slice_bound(label, side, kind)*

Calculate slice bound that corresponds to given label.

Returns leftmost (one-past-the-rightmost if side=='right') position of given label.

**Parameters**

- **label**: object
- **side**: {'left', 'right'}
- **kind**: {'ix', 'loc', 'getitem'}

### 34.12.1.67 pandas.TimedeltaIndex.get_value

*TimedeltaIndex.get_value(series, key)*

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you’re doing

### 34.12.1.68 pandas.TimedeltaIndex.get_value_maybe_box

*TimedeltaIndex.get_value_maybe_box(series, key)*

### 34.12.1.69 pandas.TimedeltaIndex.get_values

*TimedeltaIndex.get_values()*

return the underlying data as an ndarray

### 34.12.1.70 pandas.TimedeltaIndex.groupby

*TimedeltaIndex.groupby(values)*

Group the index labels by a given array of values.

**Parameters**

- **values**: array
  - Values used to determine the groups.

**Returns**

- **groups**: dict
34.12.1.71 pandas.TimedeltaIndex.holds_integer

TimedeltaIndex.holds_integer()

34.12.1.72 pandas.TimedeltaIndex.identical

TimedeltaIndex.identical(other)

Similar to equals, but check that other comparable attributes are also equal

34.12.1.73 pandas.TimedeltaIndex.insert

TimedeltaIndex.insert(loc, item)

Make new Index inserting new item at location

Parameters

loc : int

item : object

if not either a Python datetime or a numpy integer-like, returned Index dtype will be object rather than datetime.

Returns new_index : Index

34.12.1.74 pandas.TimedeltaIndex.intersection

TimedeltaIndex.intersection(other)

Specialized intersection for TimedeltaIndex objects. May be much faster than Index.intersection

Parameters

other : TimedeltaIndex or array-like

Returns y : Index or TimedeltaIndex

34.12.1.75 pandas.TimedeltaIndex.is_

TimedeltaIndex.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 True if both have same underlying data, False otherwise : bool

34.12.1.76 pandas.TimedeltaIndex.is_boolean

TimedeltaIndex.is_boolean()

34.12.1.77 pandas.TimedeltaIndex.is_categorical

TimedeltaIndex.is_categorical()
34.12.1.78 pandas.TimedeltaIndex.is_floating

TimedeltaIndex.is_floating()

34.12.1.79 pandas.TimedeltaIndex.is_integer

TimedeltaIndex.is_integer()

34.12.1.80 pandas.TimedeltaIndex.is_interval

TimedeltaIndex.is_interval()

34.12.1.81 pandas.TimedeltaIndex.is_lexsorted_for_tuple

TimedeltaIndex.is_lexsorted_for_tuple(tup)

34.12.1.82 pandas.TimedeltaIndex.is_mixed

TimedeltaIndex.is_mixed()

34.12.1.83 pandas.TimedeltaIndex.is_numeric

TimedeltaIndex.is_numeric()

34.12.1.84 pandas.TimedeltaIndex.is_object

TimedeltaIndex.is_object()

34.12.1.85 pandas.TimedeltaIndex.is_type_compatible

TimedeltaIndex.is_type_compatible(typ)

34.12.1.86 pandas.TimedeltaIndex.isin

TimedeltaIndex.isin(values)
    Compute boolean array of whether each index value is found in the passed set of values
    Parameters values: set or sequence of values
    Returns is_contained: ndarray (boolean dtype)

34.12.1.87 pandas.TimedeltaIndex.isna

TimedeltaIndex.isna()
    Detect missing values
    New in version 0.20.0.
    Returns a boolean array of whether my values are NA
See also:

`isnull` alias of isna
`pandas.isna` top-level isna

34.12.1.88 pandas.TimedeltaIndex.isnull

`TimedeltaIndex.isnull()`
Detect missing values
New in version 0.20.0.

Returns a boolean array of whether my values are NA

See also:

`isnull` alias of isna
`pandas.isna` top-level isna

34.12.1.89 pandas.TimedeltaIndex.item

`TimedeltaIndex.item()`
return the first element of the underlying data as a python scalar

34.12.1.90 pandas.TimedeltaIndex.join

`TimedeltaIndex.join(other, how='left', level=None, return_indexers=False, sort=False)`

See Index.join

34.12.1.91 pandas.TimedeltaIndex.map

`TimedeltaIndex.map(f)`

34.12.1.92 pandas.TimedeltaIndex.max

`TimedeltaIndex.max(axis=None, *args, **kwargs)`
Return the maximum value of the Index or maximum along an axis.

See also:

`numpy.ndarray.max`

34.12.1.93 pandas.TimedeltaIndex.memory_usage

`TimedeltaIndex.memory_usage(deep=False)`
Memory usage of my values

Parameters `deep` : bool

Introspect the data deeply, interrogate `object` dtypes for system-level memory consumption

Returns bytes used
See also:

numpy.ndarray.nbytes

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

34.12.1.94 pandas.TimedeltaIndex.min

TimedeltaIndex.min(axis=None, *args, **kwargs)
Return the minimum value of the Index or minimum along an axis.

See also:

numpy.ndarray.min

34.12.1.95 pandas.TimedeltaIndex.notna

TimedeltaIndex.notna()
Inverse of isna
New in version 0.20.0.

Returns a boolean array of whether my values are not NA

See also:

notnull alias of notna
pandas.notna top-level notna

34.12.1.96 pandas.TimedeltaIndex.notnull

TimedeltaIndex.notnull()
Inverse of isna
New in version 0.20.0.

Returns a boolean array of whether my values are not NA

See also:

notnull alias of notna
pandas.notna top-level notna

34.12.1.97 pandas.TimedeltaIndex.nunique

TimedeltaIndex.nunique(dropna=True)
Return number of unique elements in the object.
Excludes NA values by default.

Parameters dropna: boolean, default True
Don’t include NaN in the count.

**Returns**

nunique : int

### 34.12.1.98 pandas.TimedeltaIndex.putmask

TimedeltaIndex.putmask(mask, value)

return a new Index of the values set with the mask

**See also:**

numpy.ndarray.putmask

### 34.12.1.99 pandas.TimedeltaIndex.ravel

TimedeltaIndex.ravel(order='C')

return an ndarray of the flattened values of the underlying data

**See also:**

numpy.ndarray.ravel

### 34.12.1.100 pandas.TimedeltaIndex.reindex

TimedeltaIndex.reindex(target, method=None, level=None, limit=None, tolerance=None)

Create index with target’s values (move/add/delete values as necessary)

**Parameters**

target : an iterable

**Returns**

new_index : pd.Index

Resulting index

indexer : np.ndarray or None

Indices of output values in original index

### 34.12.1.101 pandas.TimedeltaIndex.rename

TimedeltaIndex.rename(name, inplace=False)

Set new names on index. Defaults to returning new index.

**Parameters**

name : str or list

name to set

inplace : bool

if True, mutates in place

**Returns**

new index (of same type and class...etc) [if inplace, returns None]

### 34.12.1.102 pandas.TimedeltaIndex.repeat

TimedeltaIndex.repeat(repeats, *args, **kwargs)

Analogous to ndarray.repeat
34.12.1.103 pandas.TimedeltaIndex.reshape

TimedeltaIndex.reshape(*args, **kwargs)

NOT IMPLEMENTED: do not call this method, as reshaping is not supported for Index objects and will raise an error.

Reshape an Index.

34.12.1.104 pandas.TimedeltaIndex.round

TimedeltaIndex.round(freq, *args, **kwargs)

round the index to the specified freq

Parameters freq : freq string/object

Returns index of same type

Raises ValueError if the freq cannot be converted

34.12.1.105 pandas.TimedeltaIndex.searchsorted

TimedeltaIndex.searchsorted(value, side='left', sorter=None)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted TimedeltaIndex self such that, if the corresponding elements in value were inserted before the indices, the order of self would be preserved.

Parameters value : array_like

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 indices : array of ints

Array of insertion points with the same shape as value.

See also:

numpy.searchsorted

Notes

Binary search is used to find the required insertion points.

Examples
```python
>>> x = pd.Series([1, 2, 3])
>>> x
0 1
1 2
2 3
dtype: int64

>>> x.searchsorted(4)
array([3])

>>> x.searchsorted([0, 4])
array([0, 3])

>>> x.searchsorted([1, 3], side='left')
array([0, 2])

>>> x.searchsorted([1, 3], side='right')
array([1, 3])

>>> x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'])
['apple', 'bread', 'bread', 'cheese', 'milk']
Categories (4, object): [apple < bread < cheese < milk]

>>> x.searchsorted('bread')
array([1]) # Note: an array, not a scalar

>>> x.searchsorted(['bread'])
array([1])

>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])

>>> x.searchsorted(['bread', 'eggs'], side='right')
array([3, 4]) # eggs before milk
```

### 34.12.1.106 `pandas.TimedeltaIndex.set_names`

`TimedeltaIndex.set_names(names, level=None, inplace=False)`

Set new names on index. Defaults to returning new index.

**Parameters**
- `names` : str or sequence
  name(s) to set
- `level` : int, level name, or sequence of int/level names (default None)
  If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels).
  Otherwise level must be None
- `inplace` : bool
  if True, mutates in place

**Returns**
new index (of same type and class...etc) [if inplace, returns None]
Examples

```python
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64')
>>> Index([1, 2, 3, 4]).set_names(['foo'])
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx = MultiIndex.from_tuples([(1, 'one'), (1, 'two'),
                                (2, 'one'), (2, 'two')],
                               names=['foo', 'bar'])
>>> idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['baz', 'quz'])
>>> idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['baz', 'bar'])
```

34.12.1.107 **pandas.TimedeltaIndex.set_value**

```python
TimedeltaIndex.set_value(arr, key, value)
```

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you’re doing

34.12.1.108 **pandas.TimedeltaIndex.shift**

```python
TimedeltaIndex.shift(n, freq=None)
```

Specialized shift which produces a DatetimeIndex

**Parameters**

- `n` : int
  Periods to shift by

- `freq` : DateOffset or timedelta-like, optional

**Returns**

- `shifted` : DatetimeIndex

34.12.1.109 **pandas.TimedeltaIndex.slice_indexer**

```python
TimedeltaIndex.slice_indexer(start=None, end=None, step=None, kind=None)
```

For an ordered Index, compute the slice indexer for input labels and step

**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` : string, default None

**Returns**

- `indexer` : ndarray or slice
**Notes**

This function assumes that the data is sorted, so use at your own peril.

### 34.12.1.110 pandas.TimedeltaIndex.slice_locs

TimedeltaIndex.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**: {'ix', 'loc', 'getitem'} or None

**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)
```

### 34.12.1.111 pandas.TimedeltaIndex.sort

TimedeltaIndex.sort(*args, **kwargs)

### 34.12.1.112 pandas.TimedeltaIndex.sort_values

TimedeltaIndex.sort_values(return_indexer=False, ascending=True)

Return sorted copy of Index
34.12.1.113 pandas.TimedeltaIndex.sortlevel

TimedeltaIndex.sortlevel (level=None, ascending=True, sort_remaining=None)
For internal compatibility with with the Index API
Sort the Index. This is for compat with MultiIndex

Parameters ascending : boolean, default True
False to sort in descending order
level, sort_remaining are compat parameters

Returns sorted_index : Index

34.12.1.114 pandas.TimedeltaIndex.str

TimedeltaIndex.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.str.split('_')
>>> s.str.replace('_', '')
```

34.12.1.115 pandas.TimedeltaIndex.summary

TimedeltaIndex.summary (name=None)
return a summarized representation

34.12.1.116 pandas.TimedeltaIndex.symmetric_difference

TimedeltaIndex.symmetric_difference (other, result_name=None)
Compute the symmetric difference of two Index objects. It’s sorted if sorting is possible.

Parameters other : Index or array-like
result_name : str

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
You can also use the ^ operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```
**34.12.1.120 pandas.TimedeltaIndex.to_native_types**

TimedeltaIndex.to_native_types(slicer=None, **kwargs)
Format specified values of self and return them.

**Parameters**: 
- **slicer**: int, array-like
  An indexer into self that specifies which values are used in the formatting process.
- **kwargs**: dict
  Options for specifying how the values should be formatted. These options include:
  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

**34.12.1.121 pandas.TimedeltaIndex.to_pytimedelta**

TimedeltaIndex.to_pytimedelta()
Return TimedeltaIndex as object ndarray of datetime.timedelta objects

**Returns**:  
- **datetimes**: ndarray

**34.12.1.122 pandas.TimedeltaIndex.to_series**

TimedeltaIndex.to_series(**kwargs)
Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index

**Returns**: 
- **Series**: dtype will be based on the type of the Index values.

**34.12.1.123 pandas.TimedeltaIndex.tolist**

TimedeltaIndex.tolist()
return a list of the underlying data

**34.12.1.124 pandas.TimedeltaIndex.total_seconds**

TimedeltaIndex.total_seconds()
Total duration of each element expressed in seconds.
New in version 0.17.0.

**34.12.1.125 pandas.TimedeltaIndex.transpose**

TimedeltaIndex.transpose(*args, **kwargs)
return the transpose, which is by definition self
34.12.1.126 pandas.TimedeltaIndex.union

TimedeltaIndex.union(other)

Specialized union for TimedeltaIndex objects. If combine overlapping ranges with the same DateOffset, will be much faster than Index.union

Parameters other : TimedeltaIndex or array-like

Returns y : Index or TimedeltaIndex

34.12.1.127 pandas.TimedeltaIndex.unique

TimedeltaIndex.unique()

Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

Parameters values : 1d array-like

Returns unique values.

• If the input is an Index, the return is an Index
• If the input is a Categorical dtype, the return is a Categorical
• If the input is a Series/ndarray, the return will be an ndarray

See also:
unique, Index.unique, Series.unique

34.12.1.128 pandas.TimedeltaIndex.value_counts

TimedeltaIndex.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

Returns object 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 : boolean, default False
    If True then the object returned will contain the relative frequencies of the unique values.

sort : boolean, default True
    Sort by values

ascending : boolean, default False
    Sort in ascending order

bins : integer, optional
    Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data

dropna : boolean, default True
    Don’t include counts of NaN.

Returns counts : Series
34.12.1.129 pandas.TimedeltaIndex.view

TimedeltaIndex.view(cls=None)

34.12.1.130 pandas.TimedeltaIndex.where

TimedeltaIndex.where(cond, other=None)

New in version 0.19.0.

Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters cond : boolean array-like with the same length as self
other : scalar, or array-like

34.12.2 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 (≥ 0 and less than 1 day) for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.microseconds</td>
<td>Number of microseconds (≥ 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.nanoseconds</td>
<td>Number of nanoseconds (≥ 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></td>
</tr>
</tbody>
</table>

34.12.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimedeltaIndex.to_pytimedelta()</td>
<td>Return TimedeltaIndex as object ndarray of datetime.timedelta objects</td>
</tr>
<tr>
<td>TimedeltaIndex.to_series(**kwargs)</td>
<td>Create a Series with both index and values equal to the index keys</td>
</tr>
<tr>
<td>TimedeltaIndex.round(freq, *args, **kwargs)</td>
<td>round the index to the specified freq</td>
</tr>
<tr>
<td>TimedeltaIndex.floor(freq)</td>
<td>floor the index to the specified freq</td>
</tr>
<tr>
<td>TimedeltaIndex.ceil(freq)</td>
<td>ceil the index to the specified freq</td>
</tr>
<tr>
<td>TimedeltaIndex.to_frame([index])</td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
</tbody>
</table>

34.13 PeriodIndex

PeriodIndex

Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.
34.13.1 pandas.PeriodIndex

class pandas.PeriodIndex

Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.

Index keys are boxed to Period objects which carries the metadata (eg. frequency information).

Parameters:

- **data**: array-like (1-dimensional), optional
  - Optional period-like data to construct index with

- **copy**: bool
  - Make a copy of input ndarray

- **freq**: string or period object, optional
  - One of pandas period strings or corresponding objects

- **start**: starting value, period-like, optional
  - If data is None, used as the start point in generating regular period data.

- **periods**: int, optional, > 0
  - Number of periods to generate, if generating index. Takes precedence over end argument

- **end**: end value, period-like, optional
  - If periods is none, generated index will extend to first conforming period on or just past end argument

- **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

- **tz**: object, default None
  - Timezone for converting datetime64 data to Periods

- **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
Examples

```python
>>> idx = PeriodIndex(year=year_arr, quarter=q_arr)

>>> idx2 = PeriodIndex(start='2000', end='2010', freq='A')
```

### 34.13.2 Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PeriodIndex.day</code></td>
<td>The days of the period</td>
</tr>
<tr>
<td><code>PeriodIndex.dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td><code>PeriodIndex.dayofyear</code></td>
<td>The ordinal day of the year</td>
</tr>
<tr>
<td><code>PeriodIndex.days_in_month</code></td>
<td>The number of days in the month</td>
</tr>
<tr>
<td><code>PeriodIndex.daysinmonth</code></td>
<td>The number of days in the month</td>
</tr>
<tr>
<td><code>PeriodIndex.end_time</code></td>
<td></td>
</tr>
<tr>
<td><code>PeriodIndex.freq</code></td>
<td>Return the frequency object as a string if its set, otherwise None</td>
</tr>
<tr>
<td><code>PeriodIndex.hour</code></td>
<td>The hour of the period</td>
</tr>
<tr>
<td><code>PeriodIndex.is_leap_year</code></td>
<td>Logical indicating if the date belongs to a leap year</td>
</tr>
<tr>
<td><code>PeriodIndex.minute</code></td>
<td>The minute of the period</td>
</tr>
<tr>
<td><code>PeriodIndex.month</code></td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td><code>PeriodIndex.quarter</code></td>
<td>The quarter of the date</td>
</tr>
<tr>
<td><code>PeriodIndex.qyear</code></td>
<td></td>
</tr>
<tr>
<td><code>PeriodIndex.second</code></td>
<td>The second of the period</td>
</tr>
<tr>
<td><code>PeriodIndex.start_time</code></td>
<td></td>
</tr>
<tr>
<td><code>PeriodIndex.week</code></td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td><code>PeriodIndex.weekday</code></td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td><code>PeriodIndex.weekofyear</code></td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td><code>PeriodIndex.year</code></td>
<td>The year of the period</td>
</tr>
</tbody>
</table>

#### 34.13.2.1 pandas.PeriodIndex.day

`PeriodIndex.day`

The days of the period

#### 34.13.2.2 pandas.PeriodIndex.dayofweek

`PeriodIndex.dayofweek`

The day of the week with Monday=0, Sunday=6

#### 34.13.2.3 pandas.PeriodIndex.dayofyear

`PeriodIndex.dayofyear`

The ordinal day of the year

34.13. PeriodIndex
34.13.2.4 pandas.PeriodIndex.days_in_month

`PeriodIndex.days_in_month`
  The number of days in the month

34.13.2.5 pandas.PeriodIndex.daysinmonth

`PeriodIndex.daysinmonth`
  The number of days in the month

34.13.2.6 pandas.PeriodIndex.end_time

`PeriodIndex.end_time`

34.13.2.7 pandas.PeriodIndex.freq

`PeriodIndex.freq = None`

34.13.2.8 pandas.PeriodIndex.freqstr

`PeriodIndex.freqstr`
  Return the frequency object as a string if its set, otherwise None

34.13.2.9 pandas.PeriodIndex.hour

`PeriodIndex.hour`
  The hour of the period

34.13.2.10 pandas.PeriodIndex.is_leap_year

`PeriodIndex.is_leap_year`
  Logical indicating if the date belongs to a leap year

34.13.2.11 pandas.PeriodIndex.minute

`PeriodIndex.minute`
  The minute of the period

34.13.2.12 pandas.PeriodIndex.month

`PeriodIndex.month`
  The month as January=1, December=12

34.13.2.13 pandas.PeriodIndex.quarter

`PeriodIndex.quarter`
  The quarter of the date
34.13.2.14 pandas.PeriodIndex.qyear

PeriodIndex.qyear

34.13.2.15 pandas.PeriodIndex.second

PeriodIndex.second

The second of the period

34.13.2.16 pandas.PeriodIndex.start_time

PeriodIndex.start_time

34.13.2.17 pandas.PeriodIndex.week

PeriodIndex.week

The week ordinal of the year

34.13.2.18 pandas.PeriodIndex.weekday

PeriodIndex.weekday

The day of the week with Monday=0, Sunday=6

34.13.2.19 pandas.PeriodIndex.weekofyear

PeriodIndex.weekofyear

The week ordinal of the year

34.13.2.20 pandas.PeriodIndex.year

PeriodIndex.year

The year of the period

34.13.3 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PeriodIndex.asfreq([freq, how])</td>
<td>Convert the PeriodIndex to the specified frequency freq.</td>
</tr>
<tr>
<td>PeriodIndex.strftime(date_format)</td>
<td>Return an array of formatted strings specified by date_format, which supports the same string format as the python standard library.</td>
</tr>
<tr>
<td>PeriodIndex.to_timestamp([freq, how])</td>
<td>Cast to DatetimeIndex</td>
</tr>
<tr>
<td>PeriodIndex.tz_convert(tz)</td>
<td>Convert tz-aware DatetimeIndex from one time zone to another (using</td>
</tr>
<tr>
<td>PeriodIndex.tz_localize(tz[, infer_dst])</td>
<td>Localize tz-naive DatetimeIndex to given time zone (using</td>
</tr>
</tbody>
</table>
34.13.3.1 pandas.PeriodIndex.asfreq

PeriodIndex.asfreq(freq=None, how='E')
Convert the PeriodIndex to the specified frequency freq.

Parameters

freq : str
    a frequency
how : str {'E', 'S'}
    'E', 'END', or 'FINISH' for end, 'S', 'START', or 'BEGIN' for start. Whether the elements should be aligned to the end or start within a period. January 31st ('END') vs. January 1st ('START') for example.

Returns

new : PeriodIndex with the new frequency

Examples

>>> pidx = pd.period_range('2010-01-01', '2015-01-01', freq='A')
>>> pidx
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010, ..., 2015]
Length: 6, Freq: A-DEC

>>> pidx.asfreq('M')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-12, ..., 2015-12]
Length: 6, Freq: M

>>> pidx.asfreq('M', how='S')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-01, ..., 2015-01]
Length: 6, Freq: M

34.13.3.2 pandas.PeriodIndex.strftime

PeriodIndex.strftime(date_format)
Return an array 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.

New in version 0.17.0.

Parameters
date_format : str
    date format string (e.g. "%Y-%m-%d")

Returns

ndarray of formatted strings

34.13.3.3 pandas.PeriodIndex.to_timestamp

PeriodIndex.to_timestamp(freq=None, how='start')
Cast to DatetimeIndex

Parameters

freq : string or DateOffset, default ‘D’ for week or longer, ‘S’
otherwise

Target frequency

how : {'s', 'e', 'start', 'end'}

Returns DatetimeIndex

### 34.13.3.4 pandas.PeriodIndex.tz_convert

PeriodIndex.tz_convert(tz)

Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)

**Parameters**

- **tz**: string, pytz.timezone, dateutil.tz.tzfile or None

  Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding UTC time.

**Returns**

- **normalized** : DatetimeIndex

### 34.13.3.5 pandas.PeriodIndex.tz_localize

PeriodIndex.tz_localize(tz, infer_dst=False)

Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

**Parameters**

- **tz**: string, pytz.timezone, dateutil.tz.tzfile or None

  Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding local time.

- **infer_dst**: boolean, default False

  Attempt to infer fall dst-transition hours based on order

**Returns**

- **localized** : DatetimeIndex

### 34.14 Scalars

#### 34.14.1 Period

**Period**

Represents a period of time

#### 34.14.1.1 pandas.Period

**class** pandas.Period

Represents a period of time

**Parameters**

- **value**: Period or compat.string_types, default None

  The time period represented (e.g., ‘4Q2005’)

- **freq**: str, default None

  One of pandas period strings or corresponding objects

- **year**: int, default None
month : int, default 1
quarter : int, default None
day : int, default 1
hour : int, default 0
minute : int, default 0
second : int, default 0

Attributes

day
dayofweek
dayofyear
days_in_month
daysinmonth
end_time
freq
dfreqstr
hour
is_leap_year
minute
month
ordinal
quarter
qyear
second
start_time
week
weekday
weekofyear
year

pandas.Period.day

Period.day

pandas.Period.dayofweek

Period.dayofweek

pandas.Period.dayofyear

Period.dayofyear
pandas.Period.days_in_month
Period.days_in_month

pandas.Period.daysinmonth
Period.daysinmonth

pandas.Period.end_time
Period.end_time

pandas.Period.freq
Period.freq

pandas.Period.freqstr
Period.freqstr

pandas.Period.hour
Period.hour

pandas.Period.is_leap_year
Period.is_leap_year

pandas.Period.minute
Period.minute

pandas.Period.month
Period.month

pandas.Period.ordinal
Period.ordinal

pandas.Period.quarter
Period.quarter
pandas: powerful Python data analysis toolkit, Release 0.21.0

pandas.Period.qyear

Period.qyear

pandas.Period.second

Period.second

pandas.Period.start_time

Period.start_time

pandas.Period.week

Period.week

pandas.Period.weekday

Period.weekday

pandas.Period.weekofyear

Period.weekofyear

pandas.Period.year

Period.year

Methods

<table>
<thead>
<tr>
<th>asfreq</th>
<th>Convert Period to desired frequency, either at the start or end of the interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
<td></td>
</tr>
<tr>
<td>strftime</td>
<td>Returns the string representation of the Period, depending on the selected format.</td>
</tr>
<tr>
<td>to_timestamp</td>
<td>Return the Timestamp representation of the Period at the target</td>
</tr>
</tbody>
</table>

pandas.Period.asfreq

Period.asfreq()

Convert Period to desired frequency, either at the start or end of the interval

Parameters freq : string

how : {'E', 'S', 'end', 'start'}, default 'end'
Start or end of the timespan

Returns resampled : Period

**pandas.Period.now**

```python
Period.now()
```

**pandas.Period.strftime**

```python
Period.strftime()
```

Returns the string representation of the `Period`, depending on the selected `format`. `format` 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>%f</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>Weekday as a decimal number [00(Sunday),6].</td>
<td></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 \%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.
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%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 at the target frequency at the specified end (how) of the Period

Parameters freq : string or DateOffset

Target frequency. Default is ‘D’ if self.freq is week or longer and ‘S’ otherwise

how : str, default ‘S’ (start)


Returns Timestamp

34.14.2 Attributes

- Period.day
- Period.dayofweek
- Period.dayofyear
- Period.days_in_month
- Period.daysinmonth

Continued on next page
34.14.3 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period.asfreq</td>
<td>Convert Period to desired frequency, either at the start or end of the Period.</td>
</tr>
<tr>
<td>Period.now</td>
<td>Returns the string representation of the Period, depending on the selected format.</td>
</tr>
<tr>
<td>Period.strftime</td>
<td>Return the Timestamp representation of the Period at the target.</td>
</tr>
</tbody>
</table>

34.14.4 Timestamp

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Timestamp</td>
<td>TimeStamp is the pandas equivalent of python’s Datetime and is interchangable with it in most cases.</td>
</tr>
</tbody>
</table>

34.14.4.1 pandas.Timestamp

<table>
<thead>
<tr>
<th>Parameters ts_input :</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime-like, str, int, float</td>
<td>Value to be converted to Timestamp</td>
</tr>
<tr>
<td>freq : str, DateOffset</td>
<td>Offset which Timestamp will have</td>
</tr>
<tr>
<td>tz : string, pytz.timezone, dateutil.tz.tzfile or None</td>
<td>Time zone for time which Timestamp will have.</td>
</tr>
</tbody>
</table>
unit : string
    numpy unit used for conversion, if ts_input is int or float

offset : str, DateOffset
    Deprecated, use freq

The other two forms mimic the parameters from ``datetime.datetime``. They can be passed by either position or keyword, but not both mixed together.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>asm8</td>
</tr>
<tr>
<td>day</td>
</tr>
<tr>
<td>dayofweek</td>
</tr>
<tr>
<td>dayofyear</td>
</tr>
<tr>
<td>days_in_month</td>
</tr>
<tr>
<td>daysinmonth</td>
</tr>
<tr>
<td>fold</td>
</tr>
<tr>
<td>freq</td>
</tr>
<tr>
<td>freqstr</td>
</tr>
<tr>
<td>hour</td>
</tr>
<tr>
<td>is_leap_year</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</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>microsecond</td>
</tr>
<tr>
<td>minute</td>
</tr>
<tr>
<td>month</td>
</tr>
<tr>
<td>nanosecond</td>
</tr>
<tr>
<td>offset</td>
</tr>
<tr>
<td>quarter</td>
</tr>
<tr>
<td>second</td>
</tr>
<tr>
<td>tz</td>
</tr>
<tr>
<td>tzinfo</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>week</td>
</tr>
<tr>
<td>weekday_name</td>
</tr>
<tr>
<td>weekofyear</td>
</tr>
<tr>
<td>year</td>
</tr>
</tbody>
</table>

`pandas.Timestamp.asm8`

`Timestamp.asm8`
pandas.Timestamp.day
Timestamp.day

pandas.Timestamp.dayofweek
Timestamp.dayofweek

pandas.Timestamp.dayofyear
Timestamp.dayofyear

pandas.Timestamp.days_in_month
Timestamp.days_in_month

pandas.Timestamp.daysinmonth
Timestamp.daysinmonth

pandas.Timestamp.fold
Timestamp.fold

pandas.Timestamp.freq
Timestamp.freq

pandas.Timestamp.freqstr
Timestamp.freqstr

pandas.Timestamp.hour
Timestamp.hour

pandas.Timestamp.is_leap_year
Timestamp.is_leap_year

pandas.Timestamp.is_month_end
Timestamp.is_month_end
pandas.Timestamp.is_month_start

Timestamp.is_month_start

pandas.Timestamp.is_quarter_end

Timestamp.is_quarter_end

pandas.Timestamp.is_quarter_start

Timestamp.is_quarter_start

pandas.Timestamp.is_year_end

Timestamp.is_year_end

pandas.Timestamp.is_year_start

Timestamp.is_year_start

pandas.Timestamp.microsecond

Timestamp.microsecond

pandas.Timestamp.minute

Timestamp.minute

pandas.Timestamp.month

Timestamp.month

pandas.Timestamp.nanosecond

Timestamp.nanosecond

pandas.Timestamp.offset

Timestamp.offset

pandas.Timestamp.quarter

Timestamp.quarter
pandas.Timestamp.second

Timestamp.second

pandas.Timestamp.tz

Timestamp.tz
Alias for tzinfo

pandas.Timestamp.tzinfo

Timestamp.tzinfo

pandas.Timestamp.value

Timestamp.value

pandas.Timestamp.week

Timestamp.week

pandas.Timestamp.weekday_name

Timestamp.weekday_name

pandas.Timestamp.weekofyear

Timestamp.weekofyear

pandas.Timestamp.year

Timestamp.year

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>astimezone</td>
<td>Convert tz-aware Timestamp to another time zone.</td>
</tr>
<tr>
<td>ceil</td>
<td>return a new Timestamp ceiled to this resolution</td>
</tr>
<tr>
<td>combine</td>
<td>Return ctime() style string.</td>
</tr>
<tr>
<td>date</td>
<td>Return date object with same year, month and day.</td>
</tr>
<tr>
<td>dst</td>
<td>Return self.tzinfo.dst(self).</td>
</tr>
<tr>
<td>floor</td>
<td>return a new Timestamp floored to this resolution</td>
</tr>
<tr>
<td>fromordinal</td>
<td>passed an ordinal, translate and convert to a ts</td>
</tr>
<tr>
<td>fromtimestamp</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 34.130 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isocalendar</code></td>
<td>Return a 3-tuple containing ISO year, week number, and weekday.</td>
</tr>
<tr>
<td><code>isoformat</code></td>
<td></td>
</tr>
<tr>
<td><code>isoweekday</code></td>
<td>Return the day of the week represented by the date.</td>
</tr>
<tr>
<td><code>normalize</code></td>
<td>Normalize Timestamp to midnight, preserving tz information.</td>
</tr>
<tr>
<td><code>now</code></td>
<td>Return the current time in the local timezone.</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>implements datetime.replace, handles nanoseconds</td>
</tr>
<tr>
<td><code>round</code></td>
<td>Round the Timestamp to the specified resolution.</td>
</tr>
<tr>
<td><code>strftime</code></td>
<td>format -&gt; strftime() style string.</td>
</tr>
<tr>
<td><code>strptime</code></td>
<td>string, format -&gt; new datetime parsed from a string (like time.strptime()).</td>
</tr>
<tr>
<td><code>time</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>time</code></td>
<td>Return time object with same time and tzinfo.</td>
</tr>
<tr>
<td><code>to_datetime</code></td>
<td>DEPRECATED: use <code>to_pydatetime()</code> instead.</td>
</tr>
<tr>
<td><code>to_datetime64</code></td>
<td>Returns 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_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</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</code></td>
<td>Convert tz-aware Timestamp to another time zone.</td>
</tr>
<tr>
<td><code>tz_localize</code></td>
<td>Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.</td>
</tr>
<tr>
<td><code>tzname</code></td>
<td>Return self.tzinfo.tzname(self).</td>
</tr>
<tr>
<td><code>utcfromtimestamp</code></td>
<td></td>
</tr>
<tr>
<td><code>utcnow</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()  
Convert tz-aware Timestamp to another time zone.

**Parameters**  
| tz : string, pytz.timezone, dateutil.tz.tzfile or None  |
| Time zone for which Timestamp will be converted to. None will remove timezone holding UTC time.  |

**Returns**  
| converted : Timestamp  |

** Raises **  
| `TypeError`  |
| If Timestamp is tz-naive.  |
pandas.Timestamp.ceil

Timestamp.ceil()
return a new Timestamp ceiled to this resolution

Parameters freq : a freq string indicating the ceiling resolution

pandas.Timestamp.combine

classmethod Timestamp.combine()

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.dst

Timestamp.dst()
Return self.tzinfo.dst(self).

pandas.Timestamp.floor

Timestamp.floor()
return a new Timestamp floored to this resolution

Parameters freq : a freq string indicating the flooring resolution

pandas.Timestamp.fromordinal

classmethod Timestamp.fromordinal()
passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself

Parameters ordinal : int
date corresponding to a proleptic Gregorian ordinal
freq : str, DateOffset
Offset which Timestamp will have
tz : string, pytz.timezone, dateutil.tz.tzfile or None
Time zone for time which Timestamp will have.
offset : str, DateOffset
pandas: powerful Python data analysis toolkit, Release 0.21.0

Deprecated, use freq

`pandas.Timestamp.fromtimestamp`

*classmethod* `Timestamp.fromtimestamp()`

`pandas.Timestamp.isocalendar`

`Timestamp.isocalendar()`

Return a 3-tuple containing ISO year, week number, and weekday.

`pandas.Timestamp.isoformat`

`Timestamp.isoformat()`

`pandas.Timestamp.isoweekday`

`Timestamp.isoweekday()`

Return the day of the week represented by the date. Monday == 1 ... Sunday == 7

`pandas.Timestamp.normalize`

`Timestamp.normalize()`

Normalize Timestamp to midnight, preserving tz information.

`pandas.Timestamp.now`

*classmethod* `Timestamp.now()`

Return the current time in the local timezone. Equivalent to `datetime.now([tz])`

**Parameters**

- `tz` : string / timezone object, default None
  
  Timezone to localize to

`pandas.Timestamp.replace`

`Timestamp.replace()`

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, default is 0
added in 3.6, NotImplemented

Returns  Timestamp with fields replaced

pandas.Timestamp.round

Timestamp.round()
Round the Timestamp to the specified resolution

Parameters freq : a freq string indicating the rounding resolution

Returns a new Timestamp rounded to the given resolution of freq

Raises  ValueError if the freq cannot be converted

pandas.Timestamp.strftime

Timestamp.strftime()
format -> strftime() style string.

pandas.Timestamp.strptime

Timestamp.strptime()
string, format -> new datetime parsed from a string (like time.strptime()).

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.

pandas.Timestamp.timetz

Timestamp.timetz()
Return time object with same time and tzinfo.
pandas.Timestamp.to_datetime

Timestamp.to_datetime()
DEPRECATED: use to_pydatetime() instead.
Convert a Timestamp object to a native Python datetime object.

pandas.Timestamp.to_datetime64

Timestamp.to_datetime64()
Returns a numpy.datetime64 object with ‘ns’ precision

pandas.Timestamp.to_julian_date

Timestamp.to_julian_date()
Convert TimeStamp to a Julian Date. 0 Julian date is noon January 1, 4713 BC.

pandas.Timestamp.to_period

Timestamp.to_period()
Return an period of which this timestamp is an observation.

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.

pandas.Timestamp.today

classmethod Timestamp.today()
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 : string / timezone object, default None
Timezone to localize to

pandas.Timestamp.toordinal

Timestamp.toordinal()
Return proleptic Gregorian ordinal. January 1 of year 1 is day 1.

pandas.Timestamp.tz_convert

Timestamp.tz_convert()
Convert tz-aware Timestamp to another time zone.
**Parameters** tz : string, 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.

### pandas.Timestamp.tz_localize

Timestamp.tz_localize()

Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.

**Parameters** tz : string, 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’

- 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

errors : ‘raise’, ‘coerce’, default ‘raise’

- ‘raise’ will raise a NonExistenTimeError if a timestamp is not valid in the specified timezone (e.g. due to a transition from or to DST time)
- ‘coerce’ will return NaT if the timestamp can not be converted into the specified timezone

New in version 0.19.0.

**Returns** localized : Timestamp

**Raises** TypeError

If the Timestamp is tz-aware and tz is not None.

### pandas.Timestamp.tzname

Timestamp.tzname()

Return self.tzinfo.tzname(self).

### pandas.Timestamp.utcfromtimestamp

classmethod Timestamp.utcfromtimestamp()

### pandas.Timestamp.utcnow

classmethod Timestamp.utcnow()
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

34.14.5 Properties

Timestamp.asm8
Timestamp.day
Timestamp.dayofweek
Timestamp.dayofyear
Timestamp.days_in_month
Timestamp.daysinmonth
Timestamp.hour
Timestamp.is_leap_year
Timestamp.is_month_end
Timestamp.is_month_start
Timestamp.is_quarter_end
Timestamp.is_quarter_start
Timestamp.is_year_end
Timestamp.is_year_start
Timestamp.max
Timestamp.microsecond
Timestamp.min
Timestamp.month
Timestamp.nanosecond
Timestamp.quarter
Timestamp.resolution
Timestamp.second
Timestamp.tz
    Alias for tzinfo
Timestamp.tzinfo
Timestamp.value
Timestamp.weekday_name
Timestamp.weekofyear
Timestamp.year
34.14.5.1 pandas.Timestamp.max

```
Timestamp.max = Timestamp('2262-04-11 23:47:16.854775807')
```

34.14.5.2 pandas.Timestamp.min

```
Timestamp.min = Timestamp('1677-09-21 00:12:43.145225')
```

34.14.5.3 pandas.Timestamp.resolution

```
Timestamp.resolution = datetime.timedelta(0, 0, 1)
```

34.14.6 Methods

- `Timestamp.astimezone` Convert tz-aware Timestamp to another time zone.
- `Timestamp.ceil` return a new Timestamp ceiled to this resolution
- `Timestamp.combine` Return ctime() style string.
- `Timestamp.date` Return date object with same year, month and day.
- `Timestamp.ceil` Return self.tzinfo.dst(self).
- `Timestamp.floor` return a new Timestamp floored to this resolution
- `Timestamp.freq` Timestamp.freqstr
- `Timestamp.fromordinal` passed an ordinal, translate and convert to a ts
- `Timestamp.fromtimestamp` Return a 3-tuple containing ISO year, week number, and weekday.
- `Timestamp.ctime` Return the day of the week represented by the date.
- `Timestamp.normalize` Normalize Timestamp to midnight, preserving tz information.
- `Timestamp.now` Return the current time in the local timezone.
- `Timestamp.replace` implements datetime.replace, handles nanoseconds
- `Timestamp.round` Round the Timestamp to the specified resolution
- `Timestamp.strftime` string, format -> strftime() style string.
- `Timestamp.strptime` string, format -> new datetime parsed from a string (like time.strptime()).
- `Timestamp.time` Return time object with same time but with tzinfo=None.
- `Timestamp.timetz` Return time object with same time and tzinfo.
- `Timestamp.to_datetime64` Returns a numpy.datetime64 object with ‘ns’ precision
- `Timestamp.to_period` Return an period of which this timestamp is an observation.
- `Timestamp.to_pydatetime` Convert a Timestamp object to a native Python datetime object.
- `Timestamp.today` Return the current time in the local timezone.
- `Timestamp.fromordinal` Return proleptic Gregorian ordinal.
- `Timestamp.tz_convert` Convert tz-aware Timestamp to another time zone.
## Table 34.132 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Timestamp.tz_localize</code></td>
<td>Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.</td>
</tr>
<tr>
<td><code>Timestamp.tzname</code></td>
<td>Return self.tzinfo.tzname(self).</td>
</tr>
<tr>
<td><code>Timestamp.utcfromtimestamp</code></td>
<td></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 time.localtime().</td>
</tr>
<tr>
<td><code>Timestamp.weekday</code></td>
<td>Return the day of the week represented by the date.</td>
</tr>
</tbody>
</table>

### 34.14.7 Interval

**Interval**

Immutable object implementing an Interval, a bounded slice-like interval.

#### 34.14.7.1 pandas.Interval

class `pandas.Interval`

Immutable object implementing an Interval, a bounded slice-like interval.

New in version 0.20.0.

**Parameters**

- `left` : value
  
  Left bound for interval.

- `right` : value
  
  Right bound for interval.

- `closed` : {'left', 'right', 'both', 'neither'}
  
  Whether the interval is closed on the left-side, right-side, both or neither. Defaults to `‘right’`.

**See also:**

- `IntervalIndex` an Index of intervals that are all closed on the same side.
- `cut, qcut`

**Examples**

```python
>>> iv = pd.Interval(left=0, right=5)
>>> iv
Interval(0, 5, closed='right')
>>> 2.5 in iv
True

>>> year_2017 = pd.Interval(pd.Timestamp('2017-01-01'),
...                          pd.Timestamp('2017-12-31'), closed='both')
>>> pd.Timestamp('2017-01-01 00:00') in year_2017
True
```
Attributes

- `closed`
- `closed_left`
- `closed_right`
- `left`
- `mid`
- `open_left`
- `open_right`
- `right`

```python
pandas.Interval.closed
```

```python
pandas.Interval.closed_left
```

```python
pandas.Interval.closed_right
```

```python
pandas.Interval.left
```

```python
pandas.Interval.mid
```

```python
pandas.Interval.open_left
```

```python
pandas.Interval.open_right
```

```python
pandas.Interval.right
```
34.14.8 Properties
34.14.9 Timedelta

**Timedelta**

Represents a duration, the difference between two dates or times.

34.14.9.1 pandas.Timedelta

**class pandas.Timedelta**

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, string, or integer
- **unit** : string, [D,h,m,s,ms,us,ns]
  - Denote the unit of the input, if input is an integer. Default ‘ns’.
- **days**, **seconds**, **microseconds**, **milliseconds**, **minutes**, **hours**, **weeks** : numeric, optional
  - Values for construction in compat with `datetime.timedelta`. np ints and floats will be coerced to python ints and floats.

**Notes**

The `.value` attribute is always in ns.

**Attributes**

- **asm8**
  - return a numpy timedelta64 array view of myself
- **components**
  - Return a Components NamedTuple-like
- **days**
  - Number of Days
- **delta**
  - return out delta in ns (for internal compat)
- **freq**
- **is_populated**
- **microseconds**
  - Number of microseconds (>= 0 and less than 1 second).
- **nanoseconds**
  - Number of nanoseconds (>= 0 and less than 1 microsecond).

Continued on next page
Table 34.137 – continued from previous page

| resolution | return a string representing the lowest resolution that we have |
| seconds    | Number of seconds (>= 0 and less than 1 day). |
| value      | |

pandas.Timedelta.asm8

Timedelta.asm8
return a numpy timedelta64 array view of myself

pandas.Timedelta.components

Timedelta.components
Return a Components NamedTuple-like

pandas.Timedelta.days

Timedelta.days
Number of Days
.components will return the shown components

pandas.Timedelta.delta

Timedelta.delta
return out delta in ns (for internal compat)

pandas.Timedelta.freq

Timedelta.freq

pandas.Timedelta.is_population

Timedelta.is_population

pandas.Timedelta.microseconds

Timedelta.microseconds
Number of microseconds (>= 0 and less than 1 second).
.components will return the shown components
pandas.Timedelta.nanoseconds

Timedelta.nanoseconds
Number of nanoseconds (>= 0 and less than 1 microsecond).
.components will return the shown components

pandas.Timedelta.resolution

Timedelta.resolution
return a string representing the lowest resolution that we have

pandas.Timedelta.seconds

Timedelta.seconds
Number of seconds (>= 0 and less than 1 day).
.components will return the shown components

pandas.Timedelta.value

Timedelta.value

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>return a new Timedelta ceiled to this resolution</td>
</tr>
<tr>
<td>floor</td>
<td>return a new Timedelta floored to this resolution</td>
</tr>
<tr>
<td>round</td>
<td>Round the Timedelta to the specified resolution</td>
</tr>
<tr>
<td>to_pytimedelta</td>
<td>return an actual datetime.timedelta object</td>
</tr>
<tr>
<td>to_timedelta64</td>
<td>Returns a numpy.timedelta64 object with ‘ns’ precision</td>
</tr>
<tr>
<td>total_seconds</td>
<td>Total duration of timedelta in seconds (to ns precision)</td>
</tr>
<tr>
<td>view</td>
<td>array view compat</td>
</tr>
</tbody>
</table>

pandas.Timedelta.ceil

Timedelta.ceil() return a new Timedelta ceiled to this resolution

Parameters freq : a freq string indicating the ceiling resolution

pandas.Timedelta.floor

Timedelta.floor() return a new Timedelta floored to this resolution

Parameters freq : a freq string indicating the flooring resolution
pandas.Timedelta.isoformat

Timedelta.isoformat()

New in version 0.20.0.

Returns formatted : str

See also:
Timestamp.isoformat

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 pad components, so it’s ...T5H..., not ...705H...

Examples

```python
>>> td = pd.Timedelta(days=6, minutes=50, seconds=3,
... milliseconds=10, microsseconds=10, nanoseconds=12)
>>> td.isoformat()
'P6DT0H50M3.010010012S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(days=500.5).isoformat()
'P500DT12H0MS'
```

pandas.Timedelta.round

Timedelta.round()
Round the Timedelta to the specified resolution

Parameters freq : a freq 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_pytimedelta

Timedelta.to_pytimedelta()
return an actual datetime.timedelta object note: we lose nanosecond resolution if any
pandas.Timedelta.to_timedelta64

Timedelta.to_timedelta64()
Returns a numpy.timedelta64 object with ‘ns’ precision

pandas.Timedelta.total_seconds

Timedelta.total_seconds()
Total duration of timedelta in seconds (to ns precision)

pandas.Timedelta.view

Timedelta.view()
array view compat

34.14.10 Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timedelta.asm8</td>
<td>return a numpy timedelta64 array view of myself</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.freq</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>Number of nanoseconds (&gt;= 0 and less than 1 microsec-</td>
</tr>
<tr>
<td></td>
<td>ond).</td>
</tr>
<tr>
<td>Timedelta.resolution</td>
<td>return a string representing the lowest resolution th</td>
</tr>
<tr>
<td></td>
<td>at we have</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>
</tbody>
</table>

34.14.11 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timedelta.ceil</td>
<td>return a new Timedelta ceiled to this resolution</td>
</tr>
<tr>
<td>Timedelta.floor</td>
<td>return a new Timedelta floored to this resolution</td>
</tr>
<tr>
<td>Timedelta.isoformat</td>
<td>Format Timedelta as ISO 8601 Duration like P[n]Y[n]M</td>
</tr>
<tr>
<td></td>
<td>[n]DT[n]H[n]M[n]S, where the ‘[n]’s are replaced by</td>
</tr>
<tr>
<td></td>
<td>the values.</td>
</tr>
<tr>
<td>Timedelta.round</td>
<td>Round the Timedelta to the specified resolution</td>
</tr>
<tr>
<td>Timedelta.to_pytimedelta</td>
<td>return an actual datetime.timedelta object</td>
</tr>
<tr>
<td>Timedelta.to_timedelta64</td>
<td>Returns a numpy.timedelta64 object with ‘ns’ preci-</td>
</tr>
<tr>
<td></td>
<td>sion</td>
</tr>
<tr>
<td>Timedelta.total_seconds</td>
<td>Total duration of timedelta in seconds (to ns preci-</td>
</tr>
<tr>
<td></td>
<td>sion)</td>
</tr>
</tbody>
</table>
34.15 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. EWM objects are returned by `.ewm` calls: `pandas.DataFrame.ewm()`, `pandas.Series.ewm()`, etc.

34.15.1 Standard moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Rolling.count()</code></td>
<td>rolling count of number of non-NaN observations inside provided window.</td>
</tr>
<tr>
<td><code>Rolling.sum(*args, **kwars)</code></td>
<td>rolling sum</td>
</tr>
<tr>
<td><code>Rolling.mean(*args, **kwars)</code></td>
<td>rolling mean</td>
</tr>
<tr>
<td><code>Rolling.median(**kwars)</code></td>
<td>rolling median</td>
</tr>
<tr>
<td><code>Rolling.var(ddof)</code></td>
<td>rolling variance</td>
</tr>
<tr>
<td><code>Rolling.std(ddof)</code></td>
<td>rolling standard deviation</td>
</tr>
<tr>
<td><code>Rolling.min(*args, **kwars)</code></td>
<td>rolling minimum</td>
</tr>
<tr>
<td><code>Rolling.max(*args, **kwars)</code></td>
<td>rolling maximum</td>
</tr>
<tr>
<td><code>Rolling.corr(other, pairwise)</code></td>
<td>rolling sample correlation</td>
</tr>
<tr>
<td><code>Rolling.cov(other, pairwise, ddof)</code></td>
<td>rolling sample covariance</td>
</tr>
<tr>
<td><code>Rolling.skew(**kwars)</code></td>
<td>Unbiased rolling skewness</td>
</tr>
<tr>
<td><code>Rolling.kurt(**kwars)</code></td>
<td>Unbiased rolling kurtosis</td>
</tr>
<tr>
<td><code>Rolling.apply(func, args, kwars)</code></td>
<td>rolling function apply</td>
</tr>
<tr>
<td><code>Rolling.quantile(quantile, **kwars)</code></td>
<td>rolling quantile</td>
</tr>
<tr>
<td><code>Window.mean(*args, **kwars)</code></td>
<td>window mean</td>
</tr>
<tr>
<td><code>Window.sum(*args, **kwars)</code></td>
<td>window sum</td>
</tr>
</tbody>
</table>

34.15.1.1 pandas.core.window.Rolling.count

Rolling.count()

rolling count of number of non-NaN observations inside provided window.

Returns same type as input

See also:

`pandas.Series.rolling`, `pandas.DataFrame.rolling`

34.15.1.2 pandas.core.window.Rolling.sum

Rolling.sum(*args, **kwars)

rolling sum

Parameters how : string, default None

Deprecated since version 0.18.0: Method for down- or re-sampling

Returns same type as input

See also:

`pandas.Series.rolling`, `pandas.DataFrame.rolling`
34.15.1.3 pandas.core.window.Rolling.mean

Rolling.mean(*args, **kwargs)

    rolling mean

    Parameters how : string, default None
        Deprecated since version 0.18.0: Method for down- or re-sampling
    
    Returns same type as input

    See also:
        pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.4 pandas.core.window.Rolling.median

Rolling.median(**kwargs)

    rolling median

    Parameters how : string, default ‘median’
        Deprecated since version 0.18.0: Method for down- or re-sampling
    
    Returns same type as input

    See also:
        pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.5 pandas.core.window.Rolling.var

Rolling.var(ddof=1, *args, **kwargs)

    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.
    
    Returns same type as input

    See also:
        pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.6 pandas.core.window.Rolling.std

Rolling.std(ddof=1, *args, **kwargs)

    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.
    
    Returns same type as input

    See also:
        pandas.Series.rolling, pandas.DataFrame.rolling
34.15.1.7 pandas.core.window.Rolling.min

Rolling.min (*args, **kwargs)

rolling minimum

- **Parameters**
  - how : string, default ‘min’
    - Deprecated since version 0.18.0: Method for down- or re-sampling

- **Returns**
  - same type as input

See also:
- pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.8 pandas.core.window.Rolling.max

Rolling.max (*args, **kwargs)

rolling maximum

- **Parameters**
  - how : string, default ‘max’
    - Deprecated since version 0.18.0: Method for down- or re-sampling

- **Returns**
  - same type as input

See also:
- pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.9 pandas.core.window.Rolling.corr

Rolling.corr (other=None, pairwise=None, **kwargs)

rolling sample correlation

- **Parameters**
  - other : Series, DataFrame, or ndarray, 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.

- **Returns**
  - same type as input

See also:
- pandas.Series.rolling, pandas.DataFrame.rolling

34.15.1.10 pandas.core.window.Rolling.cov

Rolling.cov (other=None, pairwise=None, ddof=1, **kwargs)

rolling sample covariance

- **Parameters**
  - other : Series, DataFrame, or ndarray, 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 \( N - \text{ddof} \), where \( N \) represents the number of elements.

**Returns** same type as input

**See also:**

- pandas.Series.rolling
- pandas.DataFrame.rolling

---

### 34.15.1.11 pandas.core.window.Rolling.skew

Rolling.**skew**(**kwargs**)

Unbiased rolling skewness

**Returns** same type as input

**See also:**

- pandas.Series.rolling
- pandas.DataFrame.rolling

---

### 34.15.1.12 pandas.core.window.Rolling.kurt

Rolling.**kurt**(**kwargs**)

Unbiased rolling kurtosis

**Returns** same type as input

**See also:**

- pandas.Series.rolling
- pandas.DataFrame.rolling

---

### 34.15.1.13 pandas.core.window.Rolling.apply

Rolling.**apply**(func, **kwargs=(), **kwargs=[])  

rolling function apply

**Parameters**  

- **func**: function
  
  Must produce a single value from an ndarray input *args and **kwargs are passed to the function

**Returns** same type as input

**See also:**

- pandas.Series.rolling
- pandas.DataFrame.rolling

---

### 34.15.1.14 pandas.core.window.Rolling.quantile

Rolling.**quantile**(quantile, **kwargs)

rolling quantile

**Parameters**  

- **quantile**: float
0 <= quantile <= 1

**Returns** same type as input

**See also:**
- `pandas.Series.rolling`, `pandas.DataFrame.rolling`

### 34.15.1.15 pandas.core.window.Window.mean

**Window.mean(*args, **kwargs)**

window mean

**Parameters**
- `how` : string, default None

Deprecated since version 0.18.0: Method for down- or re-sampling

**Returns** same type as input

**See also:**
- `pandas.Series.window`, `pandas.DataFrame.window`

### 34.15.1.16 pandas.core.window.Window.sum

**Window.sum(*args, **kwargs)**

window sum

**Parameters**
- `how` : string, default None

Deprecated since version 0.18.0: Method for down- or re-sampling

**Returns** same type as input

**See also:**
- `pandas.Series.window`, `pandas.DataFrame.window`

### 34.15.2 Standard expanding window functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expanding.count(**kwargs)</td>
<td>expanding count of number of non-NaN</td>
</tr>
<tr>
<td>Expanding.sum(*args, **kwargs)</td>
<td>expanding sum</td>
</tr>
<tr>
<td>Expanding.mean(*args, **kwargs)</td>
<td>expanding mean</td>
</tr>
<tr>
<td>Expanding.median(**kwargs)</td>
<td>expanding median</td>
</tr>
<tr>
<td>Expanding.var([ddof])</td>
<td>expanding variance</td>
</tr>
<tr>
<td>Expanding.std([ddof])</td>
<td>expanding standard deviation</td>
</tr>
<tr>
<td>Expanding.min(*args, **kwargs)</td>
<td>expanding minimum</td>
</tr>
<tr>
<td>Expanding.max(*args, **kwargs)</td>
<td>expanding maximum</td>
</tr>
<tr>
<td>Expanding.corr([other, pairwise])</td>
<td>expanding sample correlation</td>
</tr>
<tr>
<td>Expanding.cov([other, pairwise, ddof])</td>
<td>expanding sample covariance</td>
</tr>
<tr>
<td>Expanding.skew(**kwargs)</td>
<td>Unbiased expanding skewness</td>
</tr>
<tr>
<td>Expanding.kurt(**kwargs)</td>
<td>Unbiased expanding kurtosis</td>
</tr>
<tr>
<td>Expanding.apply(func[, args, kwargs])</td>
<td>expanding function apply</td>
</tr>
<tr>
<td>Expanding.quantile(quantile, **kwargs)</td>
<td>expanding quantile</td>
</tr>
</tbody>
</table>
34.15.2.1 pandas.core.window.Expanding.count

```python
Expanding.count(**kwargs)
```

expanding count of number of non-NaN observations inside provided window.

Returns same type as input

See also:
```
pandas.Series.expanding, pandas.DataFrame.expanding
```

34.15.2.2 pandas.core.window.Expanding.sum

```python
Expanding.sum(*args, **kwargs)
```

expanding sum

Parameters how : string, default None

Deprecated since version 0.18.0: Method for down- or re-sampling

Returns same type as input

See also:
```
pandas.Series.expanding, pandas.DataFrame.expanding
```

34.15.2.3 pandas.core.window.Expanding.mean

```python
Expanding.mean(*args, **kwargs)
```

expanding mean

Parameters how : string, default None

Deprecated since version 0.18.0: Method for down- or re-sampling

Returns same type as input

See also:
```
pandas.Series.expanding, pandas.DataFrame.expanding
```

34.15.2.4 pandas.core.window.Expanding.median

```python
Expanding.median(**kwargs)
```

expanding median

Parameters how : string, default 'median'

Deprecated since version 0.18.0: Method for down- or re-sampling

Returns same type as input

See also:
```
pandas.Series.expanding, pandas.DataFrame.expanding
```
### pandas.core.window.Expanding.var

Expanding.**var**(*ddof=1, *args, **kwargs*)

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.

**Returns**
same type as input

See also:

- `pandas.Series.expanding`<br>- `pandas.DataFrame.expanding`

### pandas.core.window.Expanding.std

Expanding.**std**(*ddof=1, *args, **kwargs*)

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.

**Returns**
same type as input

See also:

- `pandas.Series.expanding`<br>- `pandas.DataFrame.expanding`

### pandas.core.window.Expanding.min

Expanding.**min**(*args, **kwargs*)

expanding minimum

**Parameters**

- **how**: string, default ‘min’
  Deprecated since version 0.18.0: Method for down-or re-sampling

**Returns**
same type as input

See also:

- `pandas.Series.expanding`<br>- `pandas.DataFrame.expanding`

### pandas.core.window.Expanding.max

Expanding.**max**(*args, **kwargs*)

expanding maximum

**Parameters**

- **how**: string, default ‘max’
  Deprecated since version 0.18.0: Method for down-or re-sampling

**Returns**
same type as input

See also:

- `pandas.Series.expanding`<br>- `pandas.DataFrame.expanding`
34.15.2.9 pandas.core.window.Expanding.corr

Expanding.corr(other=None, pairwise=None, **kwargs)
expanding sample correlation

Parameters other : Series, DataFrame, or ndarray, 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.

Returns same type as input

See also:
    pandas.Series.expanding, pandas.DataFrame.expanding

34.15.2.10 pandas.core.window.Expanding.cov

Expanding.cov(other=None, pairwise=None, ddof=1, **kwargs)
expanding sample covariance

Parameters other : Series, DataFrame, or ndarray, 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.

Returns same type as input

See also:
    pandas.Series.expanding, pandas.DataFrame.expanding

34.15.2.11 pandas.core.window.Expanding.skew

Expanding.skew(**kwargs)
Unbiased expanding skewness

Returns same type as input

See also:
    pandas.Series.expanding, pandas.DataFrame.expanding
### 34.15.2.12 pandas.core.window.Expanding.kurt

**Expanding.kurt(**kwargs)

Unbiased expanding kurtosis

- **Returns**: same type as input

- **See also**:
  - `pandas.Series.expanding`, `pandas.DataFrame.expanding`

### 34.15.2.13 pandas.core.window.Expanding.apply

**Expanding.apply(func, args=(), kwargs={})**

expanding function apply

- **Parameters** `func` : function
  - Must produce a single value from an ndarray input *args and **kwargs are passed to the function
- **Returns**: same type as input

- **See also**:
  - `pandas.Series.expanding`, `pandas.DataFrame.expanding`

### 34.15.2.14 pandas.core.window.Expanding.quantile

**Expanding.quantile(quantile, **kwargs)**

expanding quantile

- **Parameters** `quantile` : float
  - 0 <= quantile <= 1
- **Returns**: same type as input

- **See also**:
  - `pandas.Series.expanding`, `pandas.DataFrame.expanding`

### 34.15.3 Exponentially-weighted moving window functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EWM.mean</code></td>
<td>exponential weighted moving average</td>
</tr>
<tr>
<td><code>EWM.std</code></td>
<td>exponential weighted moving stddev</td>
</tr>
<tr>
<td><code>EWM.var</code></td>
<td>exponential weighted moving variance</td>
</tr>
<tr>
<td><code>EWM.corr</code></td>
<td>exponential weighted sample correlation</td>
</tr>
<tr>
<td><code>EWM.cov</code></td>
<td>exponential weighted sample covariance</td>
</tr>
</tbody>
</table>

### 34.15.3.1 pandas.core.window.EWM.mean

**EWM.mean(**args, **kwargs)**

exponential weighted moving average

- **Returns**: same type as input

- **See also**:

---

Chapter 34. API Reference
34.15.3.2 pandas.core.window.EWM.std

EWM.std(bias=False, *args, **kwargs)

Exponential weighted moving standard deviation

Parameters

bias : boolean, default False
Use a standard estimation bias correction

Returns

same type as input

See also:

pandas.Series.ewm, pandas.DataFrame.ewm

34.15.3.3 pandas.core.window.EWM.var

EWM.var(bias=False, *args, **kwargs)

Exponential weighted moving variance

Parameters

bias : boolean, default False
Use a standard estimation bias correction

Returns

same type as input

See also:

pandas.Series.ewm, pandas.DataFrame.ewm

34.15.3.4 pandas.core.window.EWM.corr

EWM.corr(other=None, pairwise=None, **kwargs)

Exponential weighted sample correlation

Parameters

other : Series, DataFrame, or ndarray, 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 : boolean, default False
Use a standard estimation bias correction

Returns

same type as input

See also:

pandas.Series.ewm, pandas.DataFrame.ewm
34.15.3.5 pandas.core.window.EWM.cov

EWM.cov(*other=None, pairwise=None, bias=False, **kwargs*)

exponential weighted sample covariance

**Parameters**

other : Series, DataFrame, or ndarray, 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 : boolean, default False
    Use a standard estimation bias correction

**Returns**
same type as input

See also:

pandas.Series.ewm, pandas.DataFrame.ewm

34.16 GroupBy

GroupBy objects are returned by groupby calls: pandas.DataFrame.groupby(), pandas.Series.groupby(), etc.

34.16.1 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>Constructs NDFrame from group with provided name</td>
</tr>
</tbody>
</table>

34.16.1.1 pandas.core.groupby.GroupBy.__iter__

GroupBy.__iter__()

Groupby iterator

**Returns**
Generator yielding sequence of (name, subetted object)

for each group

34.16.1.2 pandas.core.groupby.GroupBy.groups

GroupBy.groups
dict {group name -> group labels}
34.16.1.3 pandas.core.groupby.GroupBy.indices

GroupBy.indices
dict {group name -> group indices}

34.16.1.4 pandas.core.groupby.GroupBy.get_group

GroupBy.get_group(name, obj=None)
Constructs NDFrame from group with provided name

Parameters

name : object
the name of the group to get as a DataFrame

obj : NDFrame, default None
the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns

group : type of obj

Grouper([key, level, freq, axis, sort])
A Grouper allows the user to specify a groupby instruction for a target object

34.16.1.5 pandas.Grouper

class pandas.Grouper(key=None, level=None, freq=None, axis=0, sort=False)
A Grouper allows the user to specify a groupby instruction for a target 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.

These are local specifications and will override ‘global’ settings, that is the parameters axis and level which are passed to the groupby itself.

Parameters

key : string, 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 : string / 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 : number/name of the axis, defaults to 0

sort : boolean, default to False
whether to sort the resulting labels

additional kwargs to control time-like groupers (when freq is passed)
closed : closed end of interval; left or right
label : interval boundary to use for labeling; left or right
convention : {'start', 'end', 'e', 's'}

If grouper is PeriodIndex
**Returns**  A specification for a groupby instruction

**Examples**

Syntactic sugar for `df.groupby('A')`

```python
>>> df.groupby(Grouper(key='A'))
```

Specify a resample operation on the column ‘date’

```python
>>> df.groupby(Grouper(key='date', freq='60s'))
```

Specify a resample operation on the level ‘date’ on the columns axis with a frequency of 60s

```python
>>> df.groupby(Grouper(level='date', freq='60s', axis=1))
```

**Attributes**

- `ax`
- `groups`

```python
pandas.Grouper.ax
```

```python
Grouper.ax
```

```python
pandas.Grouper.groups
```

```python
Grouper.groups
```

### 34.16.2 Function application

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GroupBy.apply(func, *args, **kwargs)</code></td>
<td>Apply function and combine results together in an intelligent way.</td>
</tr>
<tr>
<td><code>GroupBy.aggregate(func, *args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>GroupBy.transform(func, *args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>GroupBy.pipe(func, *args, **kwargs)</code></td>
<td>Apply a function with arguments to this <code>GroupBy</code> object,</td>
</tr>
</tbody>
</table>

#### 34.16.2.1 pandas.core.groupby.GroupBy.apply

```python
GroupBy.apply(func, *args, **kwargs)
```

Apply function and combine results together in an intelligent way.

The split-apply-combine combination rules attempt to be as common sense based as possible. For example:

- case 1: group DataFrame apply aggregation function (f(chunk) -> Series) yield DataFrame, with group axis having group labels
- case 2: group DataFrame apply transform function ((f(chunk) -> DataFrame with same indexes) yield DataFrame with resulting chunks glued together
case 3: group Series apply function with f(chunk) -> DataFrame yield DataFrame with result of chunks glued together

Parameters func : function

See also:

pipe Apply function to the full GroupBy object instead of to each group.

aggregate, transform

Notes

See online documentation for full exposition on how to use apply.

In the current implementation apply calls func twice on the first group to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if func has side-effects, as they will take effect twice for the first group.

34.16.2.2 pandas.core.groupby.GroupBy.aggregate

GroupBy.aggregate(func, *args, **kwargs)

34.16.2.3 pandas.core.groupby.GroupBy.transform

GroupBy.transform(func, *args, **kwargs)

34.16.2.4 pandas.core.groupby.GroupBy.pipe

GroupBy.pipe(func, *args, **kwargs)

Apply a function with arguments to this GroupBy object,

New in version 0.21.0.

Parameters func : callable or tuple of (callable, string)

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:

pandas.Series.pipe Apply a function with arguments to a series

pandas.DataFrame.pipe Apply a function with arguments to a dataframe

apply Apply function to each group instead of to the full GroupBy object.
Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> f(g(h(df.groupby('group')), arg1=a), arg2=b, arg3=c)
```

You can write

```python
>>> (df
...    .groupby('group')
...    .pipe(f, arg1)
...    .pipe(g, arg2)
...    .pipe(h, arg3))
```

See more here

### 34.16.3 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.first(**kwargs)</code></td>
<td>Compute first of group values</td>
</tr>
<tr>
<td><code>GroupBy.head(n)</code></td>
<td>Returns first n rows of each group.</td>
</tr>
<tr>
<td><code>GroupBy.last(**kwargs)</code></td>
<td>Compute last of group values</td>
</tr>
<tr>
<td><code>GroupBy.max(**kwargs)</code></td>
<td>Compute max of group values</td>
</tr>
<tr>
<td><code>GroupBy.mean(**args, **kwargs)</code></td>
<td>Compute mean of groups, excluding missing values</td>
</tr>
<tr>
<td><code>GroupBy.median(**kwargs)</code></td>
<td>Compute median of groups, excluding missing values</td>
</tr>
<tr>
<td><code>GroupBy.min(**kwargs)</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, or a subset of rows if n is a list of ints.</td>
</tr>
<tr>
<td><code>GroupBy.ohlc()</code></td>
<td>Compute sum of values, excluding missing values</td>
</tr>
<tr>
<td><code>GroupBy.prod(**kwargs)</code></td>
<td>Compute prod of group values</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.stc(ddof)</code></td>
<td>Compute standard deviation of groups, excluding missing values</td>
</tr>
<tr>
<td><code>GroupBy.sum(**kwargs)</code></td>
<td>Compute sum of group values</td>
</tr>
<tr>
<td><code>GroupBy.var(ddof)</code></td>
<td>Compute variance of groups, excluding missing values</td>
</tr>
<tr>
<td><code>GroupBy.tail(n)</code></td>
<td>Returns last n rows of each group.</td>
</tr>
</tbody>
</table>

#### 34.16.3.1 pandas.core.groupby.GroupBy.count

`GroupBy.count()`

Compute count of group, excluding missing values

See also:

`pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby`
34.16.3.2 pandas.core.groupby.GroupBy.cumcount

GroupBy.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: Series(np.arange(len(x)), x.index))
```

Parameters ascending : bool, default True
If False, number in reverse, from length of group - 1 to 0.

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

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 0
1 1
2 2
3 0
4 1
5 3
dtype: int64
>>> df.groupby('A').cumcount(ascending=False)
0 3
1 2
2 1
3 1
4 0
5 0
dtype: int64
```

34.16.3.3 pandas.core.groupby.GroupBy.first

GroupBy.first(**kwargs)
Compute first of group values

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16. GroupBy
34.16.3.4 pandas.core.groupby.GroupBy.head

GroupBy.head(n=5)
Returns first n rows of each group.

Essentially equivalent to .apply(lambda x: x.head(n)), except ignores as_index flag.

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

Examples

```python
>>> df = DataFrame([[1, 2], [1, 4], [5, 6]],
                 columns=['A', 'B'])
>>> df.groupby('A', as_index=False).head(1)
   A  B
0  1  2
2  5  6
```

34.16.3.5 pandas.core.groupby.GroupBy.last

GroupBy.last(**kwargs)
Compute last of group values

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.6 pandas.core.groupby.GroupBy.max

GroupBy.max(**kwargs)
Compute max of group values

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.7 pandas.core.groupby.GroupBy.mean

GroupBy.mean(*args, **kwargs)
Compute mean of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby
34.16.3.8 pandas.core.groupby.GroupBy.median

GroupBy.median(**kwargs)
Compute median of groups, excluding missing values
For multiple groupings, the result index will be a MultiIndex

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.9 pandas.core.groupby.GroupBy.min

GroupBy.min(**kwargs)
Compute min of group values

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.10 pandas.core.groupby.GroupBy.ngroup

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.

New in version 0.20.2.

Parameters ascending : bool, default True
If False, number in reverse, from number of group - 1 to 0.

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

Examples

```python
>>> df = pd.DataFrame({'A': list('aaabba')})
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').ngroup()
0  0
1  0
2  1
3  1
4  1
5  0
dtype: int64
>>> df.groupby('A').ngroup(ascending=False)
```

34.16. GroupBy
```
0  1
1  1
2  1
3  0
4  0
5  1
dtype: int64
```

```
0  0
1  0
2  1
3  3
4  2
5  0
dtype: int64
```

### 34.16.3.11 pandas.core.groupby.GroupBy.nth

**GroupBy**. **nth** *(n, dropna=None)*

Take the nth row from each group if n is an int, or a subset of rows if n is a list of ints.

If dropna, will take the nth non-null row, dropna is either Truthy (if a Series) or ‘all’, ‘any’ (if a DataFrame); this is equivalent to calling dropna(how=dropna) before the groupby.

**Parameters**

- **n**: int or list of ints
  - a single nth value for the row or a list of nth values
- **dropna**: None or str, optional
  - apply the specified dropna operation before counting which row is the nth row. Needs to be None, ‘any’ or ‘all’

**See also:**

- `pandas.Series.groupby`
- `pandas.DataFrame.groupby`
- `pandas.Panel.groupby`

**Examples**

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5]}, columns=['A', 'B'])
>>> g = df.groupby('A')
>>> g.nth(0)
   B
   A
   1   NaN
   2   3.0
>>> g.nth(1)
   B
   A
   1   2.0
   2   5.0
>>> g.nth(-1)
   B
   A
   1   4.0
   2   5.0
```
>>> g.nth([0, 1])
   B
   A
   1  NaN
   1  2.0
   2  3.0
   2  5.0

Specifying `dropna` allows count ignoring NaN

>>> g.nth(0, dropna='any')
   B
   A
   1  2.0
   2  3.0

NaNs denote group exhausted when using dropna

>>> g.nth(3, dropna='any')
   B
   A
   1  NaN
   2  NaN

Specifying `as_index=False` in `groupby` keeps the original index.

>>> df.groupby('A', as_index=False).nth(1)
   A  B
   1  1  2.0
   4  2  5.0

34.16.3.12 pandas.core.groupby.GroupBy.ohlc

`GroupBy.ohlc()`
Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex

See also:
`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

34.16.3.13 pandas.core.groupby.GroupBy.prod

`GroupBy.prod(**kwargs)`
Compute prod of group values

See also:
`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

34.16.3.14 pandas.core.groupby.GroupBy.size

`GroupBy.size()`
Compute group sizes

See also:
34.16.3.15 pandas.core.groupby.GroupBy.sem

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 : integer, default 1
degrees of freedom

See also:

34.16.3.16 pandas.core.groupby.GroupBy.std

GroupBy.std(ddof=1, *args, **kwargs)
Compute standard deviation of groups, excluding missing values
For multiple groupings, the result index will be a MultiIndex

Parameters
ddof : integer, default 1
degrees of freedom

See also:

34.16.3.17 pandas.core.groupby.GroupBy.sum

GroupBy.sum(**kwargs)
Compute sum of group values

See also:

34.16.3.18 pandas.core.groupby.GroupBy.var

GroupBy.var(ddof=1, *args, **kwargs)
Compute variance of groups, excluding missing values
For multiple groupings, the result index will be a MultiIndex

Parameters
ddof : integer, default 1
degrees of freedom

See also:
34.16.3.19 pandas.core.groupby.GroupBy.tail

**GroupBy.tail**(\(n=5\))

Returns last \(n\) rows of each group

Essentially equivalent to \(\text{.apply(lambda x: x.tail(n))}\), except ignores as_index flag.

*See also:*

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

**Examples**

```python
df = DataFrame([[‘a’, 1], [‘a’, 2], [‘b’, 1], [‘b’, 2]],
               columns=[‘A’, ‘B’])
df.groupby(‘A’).tail(1)
   A B
1 a 2
3 b 2
df.groupby(‘A’).head(1)
   A B
0 a 1
2 b 1
```

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.agg(arg, *args, **kwargs)</td>
<td>Aggregate using callable, string, dict, or list of string/callables</td>
</tr>
<tr>
<td>DataFrameGroupBy.all</td>
<td>Return whether all elements are True over requested axis</td>
</tr>
<tr>
<td>DataFrameGroupBy.any</td>
<td>Return whether any element is True over requested axis</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.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>
<tr>
<td>DataFrameGroupBy.cumsum([axis])</td>
<td>Cumulative sum for each group</td>
</tr>
<tr>
<td>DataFrameGroupBy.describe(**kwargs)</td>
<td>Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.diff</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>DataFrameGroupBy.ffill([limit])</td>
<td>Forward fill the values</td>
</tr>
<tr>
<td>DataFrameGroupBy.fillna</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td>DataFrameGroupBy.filter(func[, dropna])</td>
<td>Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.</td>
</tr>
<tr>
<td>DataFrameGroupBy.hist</td>
<td>Draw histogram of the DataFrame’s series using matplotlib / pylab.</td>
</tr>
</tbody>
</table>

Continued on next page
DataFrameGroupBy.idxmax
Return index of first occurrence of maximum over requested axis.

DataFrameGroupBy.idxmin
Return index of first occurrence of minimum over requested axis.

DataFrameGroupBy.mad
Return the mean absolute deviation of the values for the requested axis.

DataFrameGroupBy.pct_change
Percent change over given number of periods.

DataFrameGroupBy.plot
Class implementing the .plot attribute for groupby objects.

DataFrameGroupBy.quantile
Return values at the given quantile over requested axis, a la numpy.percentile.

DataFrameGroupBy.rank
Compute numerical data ranks (1 through n) along axis.

DataFrameGroupBy.resample
Provide resampling when using a TimeGrouper.

DataFrameGroupBy.shift
Shift each group by periods observations.

DataFrameGroupBy.size
Compute group sizes.

DataFrameGroupBy.skew
Return unbiased skew over requested axis.

DataFrameGroupBy.take
Return the elements in the given positional indices along an axis.

DataFrameGroupBy.tshift
Shift the time index, using the index’s frequency if available.

34.16.3.20 pandas.core.groupby.DataFrameGroupBy.agg

DataFrameGroupBy.agg(arg, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables

Parameters func : callable, string, dictionary, or list of string/callables
Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:
• string function name
• function
• list of functions
• dict of column names -> functions (or list of functions)

Returns aggregated : DataFrame

See also:
pandas.DataFrame.groupby.apply, pandas.DataFrame.groupby.transform, pandas.DataFrame.aggregate

Notes
Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.
Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 2],
...                    'B': [1, 2, 3, 4],
...                    'C': np.random.randn(4))

>>> 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.

>>> df.groupby('A').agg('min')
A
  B   C
1  1  0.227877
2  3 -0.562860

Multiple aggregations

>>> df.groupby('A').agg(['min', 'max'])
   B   C
min  max
A
  1  2  0.227877  0.362838
  3  4 -0.562860  1.267767

Select a column for aggregation

>>> df.groupby('A').B.agg(['min', 'max'])
   B
min  max
A
  1  2
  3  4

Different aggregations per column

>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
   B   C
min  max  sum
A
  1  2  0.590716
  3  4  0.704907
```

34.16.3.21 pandas.core.groupby.DataFrameGroupBy.all

DataFrameGroupBy.all
Return whether all elements are True over requested axis

Parameters
  - axis : {index (0), columns (1)}
  - skipna : boolean, 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
bool_only : boolean, default None
   Include only boolean columns. If None, will attempt to use everything, then use only
   boolean data. Not implemented for Series.

Returns all : Series or DataFrame (if level specified)

34.16.3.22 pandas.core.groupby.DataFrameGroupBy.any

DataFrameGroupBy.any
   Return whether any element is True over requested axis

Parameters axis : {index (0), columns (1)}
   skipna : boolean, 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
bool_only : boolean, default None
      Include only boolean columns. If None, will attempt to use everything, then use only
      boolean data. Not implemented for Series.

Returns any : Series or DataFrame (if level specified)

34.16.3.23 pandas.core.groupby.DataFrameGroupBy.bfill

DataFrameGroupBy.bfill (limit=None)
   Backward fill the values

Parameters limit : integer, optional
   limit of how many values to fill

See also:
   pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.24 pandas.core.groupby.DataFrameGroupBy.corr

DataFrameGroupBy.corr
   Compute pairwise correlation of columns, excluding NA/null values

Parameters method : {'pearson', 'kendall', 'spearman'}
   pearson : standard correlation coefficient
   kendall : Kendall Tau correlation coefficient
   spearman : Spearman rank correlation

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**  
y : DataFrame

### 34.16.3.25 pandas.core.groupby.DataFrameGroupBy.count

DataFrameGroupBy.count()  
Compute count of group, excluding missing values

### 34.16.3.26 pandas.core.groupby.DataFrameGroupBy.cov

DataFrameGroupBy.cov  
Compute pairwise covariance of columns, excluding NA/null values

**Parameters**  
min_periods : int, optional  
Minimum number of observations required per pair of columns to have a valid result.

**Returns**  
y : DataFrame

**Notes**

y contains the covariance matrix of the DataFrame’s time series. The covariance is normalized by N-1 (unbiased estimator).

### 34.16.3.27 pandas.core.groupby.DataFrameGroupBy.cummax

DataFrameGroupBy.cummax(axis=0, **kwargs)  
Cumulative max for each group

**See also:**

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

### 34.16.3.28 pandas.core.groupby.DataFrameGroupBy.cummin

DataFrameGroupBy.cummin(axis=0, **kwargs)  
Cumulative min for each group

**See also:**

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

### 34.16.3.29 pandas.core.groupby.DataFrameGroupBy.cumprod

DataFrameGroupBy.cumprod(axis=0, *args, **kwargs)  
Cumulative product for each group

**See also:**

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby
34.16.3.30 pandas.core.groupby.DataFrameGroupBy.cumsum

DataFrameGroupBy.cumsum(axis=0, *args, **kwargs)
Cumulative sum for each group

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.31 pandas.core.groupby.DataFrameGroupBy.describe

DataFrameGroupBy.describe(**kwargs)
Generates descriptive statistics 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(include=['O'])). To exclude pandas categorical columns, use 'category'

• None (default) : The result will exclude nothing.

Returns summary: Series/DataFrame of summary statistics

See also:
DataFrame.count, DataFrame.max, DataFrame.min, DataFrame.mean, DataFrame.std, DataFrame.select_dtypes
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
```

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()
count    3
unique   2
top      2010-01-01 00:00:00
```
Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                    'numeric': [1, 2, 3],
...                    'categorical': pd.Categorical(['d', 'e', 'f'])
...                })
>>> df.describe()
    numeric
   count    3.0
   mean     2.0
   std      1.0
   25%      1.5
   50%      2.0
   75%      2.5
   max      3.0
```

Describing all columns of a DataFrame regardless of data type.

```python
>>> df.describe(include='all')
       categorical  numeric object
      count      3.0      3.0     3
     unique      3.0     NaN      3
      top        f.0     NaN      c
      freq       1.0     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
   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
```
Including only string columns in a DataFrame description.

```python
>>> df.describe(include=[np.object])
  object
count   3
unique  3
top   c
freq    1
```

Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
  categorical
count   3
unique  3
top    f
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     c
freq        1     1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.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
```

34.16.3.32 pandas.core.groupby.DataFrameGroupBy.diff

DataFrameGroupBy.diff

1st discrete difference of object

**Parameters**

- periods : int, default 1
  - Periods to shift for forming difference
axis : {0 or ‘index’, 1 or ‘columns’}, default 0

Take difference over rows (0) or columns (1).

Returns diffed : DataFrame

34.16.3.33 pandas.core.groupby.DataFrameGroupBy.ffill

DataFrameGroupBy.**ffill**(limit=None)

Forward fill the values

Parameters limit : integer, optional

limit of how many values to fill

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.34 pandas.core.groupby.DataFrameGroupBy.fillna

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.


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

axis : {0 or ‘index’, 1 or ‘columns’}

inplace : boolean, 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 filled : DataFrame

See also:
reindex, asfreq
Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
```

```
A  B  C  D
0  NaN 2.0 0.0 0
1  3.0 4.0 NaN 1
2  NaN NaN NaN 5
3  NaN 3.0 NaN 4
```

Replace all NaN elements with 0s.

```python
>>> df.fillna(0)
```

```
A  B  C  D
0 0.0 2.0 0.0 0
1 3.0 4.0 0.0 1
2 0.0 0.0 0.0 5
3 0.0 3.0 0.0 4
```

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
1 3.0 4.0 NaN 1
2 3.0 4.0 NaN 5
3 3.0 3.0 NaN 4
```

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}
```

```python
>>> df.fillna(value=values)
```

```
A  B  C  D
0 0.0 2.0 2.0 0
1 3.0 4.0 2.0 1
2 0.0 1.0 2.0 5
3 0.0 3.0 2.0 4
```

Only replace the first NaN element.

```python
>>> df.fillna(value=values, limit=1)
```

```
A  B  C  D
0 NaN 2.0 NaN 0
1 3.0 4.0 NaN 1
2 NaN 1.0 NaN 5
3 NaN 3.0 NaN 4
```

34.16.3.35 pandas.core.groupby.DataFrameGroupBy.filter

DataFrameGroupBy.filter(func, dropna=True, *args, **kwargs)

Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.
Parameters `f` : 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.

Examples

```python
>>> import pandas as pd
>>> df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar', ...
... 'foo', 'bar'],
... 'B' : [1, 2, 3, 4, 5, 6],
... 'C' : [2.0, 5., 8., 1., 2., 9.]})
>>> 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
```

34.16.3.36 `pandas.core.groupby.DataFrameGroupBy.hist`

DataFrameGroupBy`.hist`

Draw histogram of the DataFrame's series using matplotlib / pylab.

Parameters `data` : DataFrame

`column` : string or sequence

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` : boolean, 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
rotation of y axis labels

ax : matplotlib axes object, default None

sharex : boolean, 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 subplots in a figure!

sharey : boolean, default False

In case subplots=True, share y axis and set some y axis labels to invisible

figsize : tuple

The size of the figure to create in inches by default

layout : tuple, optional

Tuple of (rows, columns) for the layout of the histograms

bins : integer, default 10

Number of histogram bins to be used

kwds : other plotting keyword arguments

To be passed to hist function

34.16.3.37 pandas.core.groupby.DataFrameGroupBy.idxmax

DataFrameGroupBy.idxmax

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

0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be first index.

Returns idxmax : Series

See also:

Series.idxmax

Notes

This method is the DataFrame version of ndarray.argmax.

34.16.3.38 pandas.core.groupby.DataFrameGroupBy.idxmin

DataFrameGroupBy.idxmin

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

0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise

skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns idxmin : Series

See also:
Series.idxmin

Notes

This method is the DataFrame version of ndarray.argmin.

34.16.3.39 pandas.core.groupby.DataFrameGroupBy.mad

DataFrameGroupBy.mad
Return the mean absolute deviation of the values for the requested axis

Parameters axis : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA or empty, 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

numeric_only : boolean, 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 mad : Series or DataFrame (if level specified)

34.16.3.40 pandas.core.groupby.DataFrameGroupBy.pct_change

DataFrameGroupBy.pct_change
Percent change over given number of periods.

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 offset alias string, optional

Increment to use from time series API (e.g. ‘M’ or BDay())

Returns chg : NDFrame
Notes

By default, the percentage change is calculated along the stat axis: 0, or Index, for DataFrame and 1, or minor for Panel. You can change this with the axis keyword argument.

34.16.3.41 pandas.core.groupby.DataFrameGroupBy.plot

DataFrameGroupBy.plot
Class implementing the .plot attribute for groupby objects

34.16.3.42 pandas.core.groupby.DataFrameGroupBy.quantile

DataFrameGroupBy.quantile
Return values at the given quantile over requested axis, a la numpy.percentile.

Parameters q : float or array-like, default 0.5 (50% quantile)
    0 <= q <= 1, the quantile(s) to compute
axis : {0, 1, ‘index’, ‘columns’} (default 0)
    0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise
    New in version 0.18.0.
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 quantiles : 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.

Examples

```python
>>> df = DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                  columns=['a', 'b'])
>>> df.quantile(.1)
a  1.3
b  3.7
dtype: float64
>>> df.quantile([.1, .5])
a   b
0.1  1.3  3.7
0.5  1.7  3.5
```

34.16. GroupBy
### pandas.core.groupby.DataFrameGroupBy.rank

**DataFrameGroupBy.rank**

Compute numerical data ranks (1 through n) along axis. 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'}
  - 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
- **numeric_only**: boolean, default None
  Include only float, int, boolean data. Valid only for DataFrame or Panel objects
- **na_option**: {'keep', 'top', 'bottom'}
  - keep: leave NA values where they are
  - top: smallest rank if ascending
  - bottom: smallest rank if descending
- **ascending**: boolean, default True
  False for ranks by high (1) to low (N)
- **pct**: boolean, default False
  Computes percentage rank of data

**Returns**

- **ranks**: same type as caller

### pandas.core.groupby.DataFrameGroupBy.resample

**DataFrameGroupBy.resample**

Provide resampling when using a TimeGrouper Return a new grouper with our resampler appended

See also:


### pandas.core.groupby.DataFrameGroupBy.shift

**DataFrameGroupBy.shift**

Shift each group by periods observations

**Parameters**

- **periods**: integer, default 1
number of periods to shift
freq: frequency string
axis: axis to shift, default 0

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.46 pandas.core.groupby.DataFrameGroupBy.size

DataFrameGroupBy.size()
Compute group sizes

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.16.3.47 pandas.core.groupby.DataFrameGroupBy.skew

DataFrameGroupBy.skew
Return unbiased skew over requested axis Normalized by N-1

Parameters
axis: {index (0), columns (1)}
skipna: boolean, default True
Exclude NA/null values. If an entire row/column is NA or empty, 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
numeric_only: boolean, 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
skew: Series or DataFrame (if level specified)

34.16.3.48 pandas.core.groupby.DataFrameGroupBy.take

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: int, default 0
The axis on which to select elements. "0" means that we are selecting rows, "1" means that we are selecting columns, etc.
convert: bool, default True
Deprecation:

```
Deprecated since version 0.21.0: In the future, negative indices will always be converted.
```

Whether to convert negative indices into positive ones. For example, -1 would map to len(axis) - 1. The conversions are similar to the behavior of indexing a regular Python list.

**is_copy**: bool, default True

Whether to return a copy of the original object or not.

**Returns**

```
taken : type of caller
    An array-like containing the elements taken from the object.
```

**See also**

```
numpy.ndarray.take, numpy.take
```

**Examples**

```python
def = 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])
def
    name class max_speed
0  falcon   bird    389.0
2  parrot   bird     24.0
3   lion  mammal     80.5
1  monkey  mammal  NaN
>>> df.take([0, 3])
   name class max_speed
0  falcon   bird    389.0
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
def.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
def.take([1, 2], axis=1)
   class max_speed
0  bird    389.0
1  bird     24.0
2  mammal    80.5
3  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
def.take([-1, -2])
   name class max_speed
1  monkey  mammal  NaN
3   lion  mammal    80.5
```
### 34.16.3.49 pandas.core.groupby.DataFrameGroupBy.tshift

**DataFrameGroupBy.tshift**

Shift the time index, using the index’s frequency if available.

**Parameters**

- **periods** : int
  - Number of periods to move, can be positive or negative
- **freq** : DateOffset, timedelta, or time rule string, default None
  - Increment to use from the tseries module or time rule (e.g. ‘EOM’)
- **axis** : int or basestring
  - Corresponds to the axis that contains the Index

**Returns**

- **shifted** : NDFrame

**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.

The following methods are available only for **SeriesGroupBy** objects.

<table>
<thead>
<tr>
<th><strong>SeriesGroupBy.nlargest</strong></th>
<th>Return the largest ( n ) elements.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SeriesGroupBy.nsmallest</strong></td>
<td>Return the smallest ( n ) elements.</td>
</tr>
<tr>
<td><strong>SeriesGroupBy.nunique([dropna])</strong></td>
<td>Returns number of unique elements in the group</td>
</tr>
<tr>
<td><strong>SeriesGroupBy.unique</strong></td>
<td>Return unique values in the object.</td>
</tr>
<tr>
<td><strong>SeriesGroupBy.value_counts([normalize, ...])</strong></td>
<td></td>
</tr>
</tbody>
</table>

### 34.16.3.50 pandas.core.groupby.SeriesGroupBy.nlargest

**SeriesGroupBy.nlargest**

Return the largest \( n \) elements.

**Parameters**

- **n** : int
  - Return this many descending sorted values
- **keep** : {'first', 'last', False}, default ‘first’
  - Where there are duplicate values: - first: take the first occurrence. - last: take the last occurrence.

**Returns**

- **top_n** : Series
  - The \( n \) largest values in the Series, in sorted order

**See also**

- Series.nsmallest

**Notes**

Faster than `.sort_values(ascending=False).head(n)` for small \( n \) relative to the size of the Series object.
Examples

```python
>>> import pandas as pd
>>> import numpy as np

>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921  4.644710
82124  4.608745
421689  4.564644
425277  4.47014
718691  4.414137
43154  4.403520
283187  4.313922
595519  4.273635
503969  4.250236
121637  4.240952
dtype: float64
```

34.16.3.51 pandas.core.groupby.SeriesGroupBy.nsmallest

SeriesGroupBy.nsmallest
Return the smallest n elements.

Parameters:

- **n**: int
  Return this many ascending sorted values

- **keep**: {'first', 'last', False}, default 'first'
  Where there are duplicate values: - `first`: take the first occurrence. - `last`: take the last occurrence.

Returns:

- **bottom_n**: Series
  The n smallest values in the Series, in sorted order

See also:

Series.nlargest

Notes

Faster than `.sort_values().head(n)` for small n relative to the size of the Series object.

Examples

```python
>>> import pandas as pd
>>> import numpy as np

>>> s = pd.Series(np.random.randn(10**6))
>>> s.nsmallest(10)  # only sorts up to the N requested
288532  -4.954580
732345  -4.835960
64803  -4.812550
446457  -4.609998
501225  -4.483945
669476  -4.472935
```
34.16.3.52 pandas.core.groupby.SeriesGroupBy.nunique

SeriesGroupBy.nunique (dropna=True)
Returns number of unique elements in the group

34.16.3.53 pandas.core.groupby.SeriesGroupBy.unique

SeriesGroupBy.unique
Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

Parameters values : 1d array-like

Returns unique values.

• If the input is an Index, the return is an Index
• If the input is a Categorical dtype, the return is a Categorical
• If the input is a Series/ndarray, the return will be an ndarray

See also: unique, Index.unique, Series.unique

34.16.3.54 pandas.core.groupby.SeriesGroupBy.value_counts

SeriesGroupBy.value_counts (normalize=False, sort=True, ascending=False, bins=None, dropna=True)
The following methods are available only for DataFrameGroupBy objects.

DataFrameGroupBy.corrwith
Compute pairwise correlation between rows or columns of two DataFrame objects.

DataFrameGroupBy.boxplot(grouped[, ...])
Make box plots from DataFrameGroupBy data.

34.16.3.55 pandas.core.groupby.DataFrameGroupBy.corrwith

DataFrameGroupBy.corrwith
Compute pairwise correlation between rows or columns of two DataFrame objects.

Parameters other : DataFrame

axis : {0 or ‘index’, 1 or ‘columns’}, default 0
0 or ‘index’ to compute column-wise, 1 or ‘columns’ for row-wise

drop : boolean, default False
Drop missing indices from result, default returns union of all
Returns correls : Series

34.16.3.56 pandas.core.groupby.DataFrameGroupBy.boxplot

DataFrameGroupBy.boxplot (grouped, subplots=True, column=None, fontsize=None, rot=0, grid=True, ax=None, figsize=None, layout=None, **kwds)

Make box plots from DataFrameGroupBy data.

Parameters grouped : Grouped DataFrame

subplots :
  • 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 string

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)
  (rows, columns) for the layout of the plot

kwds : other plotting keyword arguments to be passed to matplotlib boxplot function

Returns dict of key/value = group key/DataFrame.boxplot return value
  or DataFrame.boxplot return value in case subplots=figures=False

Examples

```python
>>> import pandas
>>> import numpy as np
>>> import itertools

>>> tuples = [t for t in itertools.product(range(1000), range(4))]

>>> index = pandas.MultiIndex.from_tuples(tuples, names=['lvl0', 'lvl1'])

>>> data = np.random.randn(len(index),4)

>>> df = pandas.DataFrame(data, columns=list('ABCD'), index=index)

>>> grouped = df.groupby(level='lvl1')

>>> boxplot_frame_groupby(grouped)

>>> grouped = df.unstack(level='lvl1').groupby(level=0, axis=1)

>>> boxplot_frame_groupby(grouped, subplots=False)
```
34.17 Resampling

Resampler objects are returned by resample calls: pandas.DataFrame.resample(), pandas.Series.resample().

34.17.1 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resampler.<strong>iter</strong>()</td>
<td>Groupby iterator</td>
</tr>
<tr>
<td>Resampler.groups</td>
<td>dict {group name -&gt; group labels}</td>
</tr>
<tr>
<td>Resampler.indices</td>
<td>dict {group name -&gt; group indices}</td>
</tr>
<tr>
<td>Resampler.get_group(name[, obj])</td>
<td>Constructs NDFrame from group with provided name</td>
</tr>
</tbody>
</table>

34.17.1.1 pandas.core.resample.Resampler.__iter__

Resampler.__iter__()

Groupby iterator

Returns Generator yielding sequence of (name, subsetted object)

for each group

34.17.1.2 pandas.core.resample.Resampler.groups

Resampler.groups
dict {group name -> group labels}

34.17.1.3 pandas.core.resample.Resampler.indices

Resampler.indices
dict {group name -> group indices}

34.17.1.4 pandas.core.resample.Resampler.get_group

Resampler.get_group(name, obj=None)

Constructs NDFrame from group with provided name

Parameters name : object

the name of the group to get as a DataFrame

obj : NDFrame, default None

the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns group : type of obj

34.17.2 Function application
34.17.2.1 pandas.core.resample.Resampler.apply

Resampler. **apply** (arg, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables

**Parameters** func : callable, string, dictionary, or list of string/callables
Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:
• string function name
• function
• list of functions
• dict of column names -> functions (or list of functions)

**Returns** aggregated : DataFrame

See also:
pandas.DataFrame.groupby.aggregate, pandas.DataFrame.resample.transform, pandas.DataFrame.aggregate

**Notes**
Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.

**Examples**

```python
>>> s = Series([1,2,3,4,5],
              index=pd.date_range('20130101',
                                 periods=5,freq='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')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left, label=left, convention=start, base=0]

>>> 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'])

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>3</td>
<td>1.5</td>
<td>2</td>
</tr>
<tr>
<td>2013-01-01 00:00:02</td>
<td>7</td>
<td>3.5</td>
<td>4</td>
</tr>
<tr>
<td>2013-01-01 00:00:04</td>
<td>5</td>
<td>5.0</td>
<td>5</td>
</tr>
</tbody>
</table>

>>> r.agg({'result' : lambda x: x.mean() / x.std(), 'total' : np.sum})

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>3</td>
<td>2.121320</td>
</tr>
<tr>
<td>2013-01-01 00:00:02</td>
<td>7</td>
<td>4.949747</td>
</tr>
<tr>
<td>2013-01-01 00:00:04</td>
<td>5</td>
<td>NaN</td>
</tr>
</tbody>
</table>

34.17.2.2 pandas.core.resample.Resampler.aggregate

Resampler.aggregate(arg, *args, **kwargs)
Aggregate using callable, string, dict, or list of string/callables

Parameters

- **func** : callable, string, dictionary, or list of string/callables

  Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

  Accepted Combinations are:
  - string function name
  - function
  - list of functions
  - dict of column names -> functions (or list of functions)

Returns

- **aggregated** : DataFrame

See also:

- pandas.DataFrame.groupby.aggregate
- pandas.DataFrame.resample.transform
- pandas.DataFrame.aggregate

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use the alias.
Examples

```python
>>> s = Series([1, 2, 3, 4, 5],
              index=pd.date_range('20130101',
                                  periods=5, freq='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')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left, label=left, convention=start, base=0]

>>> 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})
total   result
2013-01-01 00:00:00    3    2.121320
2013-01-01 00:00:02    7    4.949747
2013-01-01 00:00:04    5    NaN
```

34.17.2.3 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 func : function

To apply to each group. Should return a Series with the same index

Returns transformed : Series

Examples

```python
>>> resampled.transform(lambda x: (x - x.mean()) / x.std())
```

34.17.3 Upsampling
Resampler.ffill([limit])  Forward fill the values
Resampler.backfill([limit]) Backward fill the values
Resampler.bfill([limit]) Forward fill the values
Resampler.nearest([limit]) Fill values with nearest neighbor starting from center
Resampler.fillna(method[, limit]) Fill missing values
Resampler.asfreq([fill_value]) return the values at the new freq.
Resampler.interpolate([method, axis, limit, ...]) Interpolate values according to different methods.

34.17.3.1 pandas.core.resample.Resampler.ffill

Resampler.**ffill** *(limit=None)*
Forward fill the values

**Parameters**
- **limit**: integer, optional
  - limit of how many values to fill

**Returns**
an upsampled Series

**See also:**
Series.fillna, DataFramefillna

34.17.3.2 pandas.core.resample.Resampler.backfill

Resampler.**backfill** *(limit=None)*
Backward fill the values

**Parameters**
- **limit**: integer, optional
  - limit of how many values to fill

**Returns**
an upsampled Series

**See also:**
Series.fillna, DataFramefillna

34.17.3.3 pandas.core.resample.Resampler.bfill

Resampler.**bfill** *(limit=None)*
Backward fill the values

**Parameters**
- **limit**: integer, optional
  - limit of how many values to fill

**Returns**
an upsampled Series

**See also:**
Series.fillna, DataFramefillna
34.17.3.4 pandas.core.resample.Resampler.pad

Resampler.pad(limit=None)

Forward fill the values

Parameters limit : integer, optional
limit of how many values to fill

Returns an upsampled Series

See also:
Series.fillna, DataFrame.fillna

34.17.3.5 pandas.core.resample.Resampler.nearest

Resampler.nearest(limit=None)

Fill values with nearest neighbor starting from center

Parameters limit : integer, optional
limit of how many values to fill

New in version 0.21.0.

Returns an upsampled Series

See also:
Series.fillna, DataFrame.fillna

34.17.3.6 pandas.core.resample.Resampler.fillna

Resampler.fillna(method, limit=None)

Fill missing values

Parameters method : str, method of resampling (‘ffill’, ‘bfill’)
limit : integer, optional
limit of how many values to fill

See also:
Series.fillna, DataFrame.fillna

34.17.3.7 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 upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

See also:
Series.asfreq, DataFrame.asfreq
**34.17.3.8 pandas.core.resample.Resampler.interpolate**

`Resampler.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', downcast=None, **kwargs)`

Interpolate values according to different methods.

New in version 0.18.1.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

**Parameters**

- `method` : {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}
  - ‘linear’: ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
  - ‘time’: interpolation works on daily and higher resolution data to interpolate given length of interval
  - ‘index’, ‘values’: use the actual numerical values of the index
  - ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘polynomial’ is passed to `scipy.interpolate.interp1d`. Both ‘polynomial’ and ‘spline’ require that you also specify an `order` (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the actual numerical values of the index.
  - ‘krogh’, ‘piecewise_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
  - ‘from_derivatives’ refers to `BPoly.from_derivatives` which replaces ‘piecewise_polynomial’ interpolation method in scipy 0.18

New in version 0.18.1: Added support for the ‘akima’ method. Added interpolate method ‘from_derivatives’ which replaces ‘piecewise_polynomial’ in scipy 0.18; backwards-compatible with scipy < 0.18

- `axis` : {0, 1}, default 0
  - 0: fill column-by-column
  - 1: fill row-by-row

- `limit` : int, default None.
  - Maximum number of consecutive NaNs to fill. Must be greater than 0.

- `limit_direction` : {'forward', 'backward', 'both'}, default ‘forward’
  - If limit is specified, consecutive NaNs will be filled in this direction.
  - New in version 0.17.0.

- `inplace` : bool, default False
  - Update the NDFrame in place if possible.

- `downcast` : optional, ‘infer’ or None, defaults to None
  - Downcast dtypes if possible.
**kwargs** : keyword arguments to pass on to the interpolating function.

**Returns** Series or DataFrame of same shape interpolated at the NaNs

**See also:** reindex, replace, `fillna`

**Examples**

Filling in NaNs

```python
def s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0 0
1 1
2 2
3 3
dtype: float64
```

### 34.17.4 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>Returns 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 sum of values, 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>
</tbody>
</table>

### 34.17.4.1 pandas.core.resample.Resampler.count

**Resampler.count**

Compute count of group, excluding missing values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`
34.17.4.2 pandas.core.resample.Resampler.nunique

Resampler.nunique(_method='nunique')
    Returns number of unique elements in the group

34.17.4.3 pandas.core.resample.Resampler.first

Resampler.first(_method='first', *args, **kwargs)
    Compute first of group values
    See also:
        pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.17.4.4 pandas.core.resample.Resampler.last

Resampler.last(_method='last', *args, **kwargs)
    Compute last of group values
    See also:
        pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.17.4.5 pandas.core.resample.Resampler.max

Resampler.max(_method='max', *args, **kwargs)
    Compute max of group values
    See also:
        pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.17.4.6 pandas.core.resample.Resampler.mean

Resampler.mean(_method='mean', *args, **kwargs)
    Compute mean of groups, excluding missing values
    For multiple groupings, the result index will be a MultiIndex
    See also:
        pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

34.17.4.7 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
    See also:
        pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby
34.17.4.8 pandas.core.resample.Resampler.min

Resampler.min(_method='min', *args, **kwargs)
Compute min of group values

See also:

34.17.4.9 pandas.core.resample.Resampler.ohlc

Resampler.ohlc(_method='ohlc', *args, **kwargs)
Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex

See also:

34.17.4.10 pandas.core.resample.Resampler.prod

Resampler.prod(_method='prod', *args, **kwargs)
Compute prod of group values

See also:

34.17.4.11 pandas.core.resample.Resampler.size

Resampler.size()
Compute group sizes

See also:

34.17.4.12 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
ddf : integer, default 1
degrees of freedom

See also:

34.17.4.13 pandas.core.resample.Resampler.std

Resampler.std(ddof=1, *args, **kwargs)
Compute standard deviation of groups, excluding missing values
Parameters `ddof`: integer, default 1

degrees of freedom

34.17.4.14 pandas.core.resample.Resampler.sum

Resampler.sum(_method='sum', *args, **kwargs)
Compute sum of group values

See also:
```
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby
```

34.17.4.15 pandas.core.resample.Resampler.var

Resampler.var(ddof=1, *args, **kwargs)
Compute variance of groups, excluding missing values

Parameters `ddof`: integer, default 1

degrees of freedom

34.18 Style

Styler objects are returned by `pandas.DataFrame.style`.

34.18.1 Constructor

```
Styler(data[, precision, table_styles, ...])
```
Helps style a DataFrame or Series according to the data with HTML and CSS.

34.18.1.1 pandas.io.formats.style.Styler

```
class pandas.io.formats.style.Styler(data, precision=None, table_styles=None, uuid=None, caption=None, table_attributes=None)
```
Helps style a DataFrame or Series according to the data with HTML and CSS.

New in version 0.17.1.

**Warning:** This is a new feature and is under active development. We’ll be adding features and possibly making breaking changes in future releases.

Parameters `data`: Series or DataFrame

precision: int

precision to round floats to, defaults to `pd.options.display.precision`

table_styles: list-like, default None

list of {selector: (attr, value)} dict; see Notes

uuid: str, default None
a unique identifier to avoid CSS collisions: generated automatically

**caption** : `str`, default `None`

caption to attach to the table

**See also:**

`pandas.DataFrame.style`

**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.render` 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`

**Attributes**

`env`

`template`

`loader`

`pandas.io.formats.style.Styler.env`

`Styler.env = <jinja2.environment.Environment object>`

`pandas.io.formats.style.Styler.template`

`Styler.template = <Template 'html.tpl'>`

`pandas.io.formats.style.Styler.loader`

`Styler.loader = <jinja2.loaders.PackageLoader object>`
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(func[, axis, subset])</code></td>
<td>Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.</td>
</tr>
<tr>
<td><code>applymap(func[, subset])</code></td>
<td>Apply a function elementwise, updating the HTML representation with the result.</td>
</tr>
<tr>
<td><code>background_gradient([cmap, low, high, axis, ...])</code></td>
<td>Color the background in a gradient according to the data in each column (optionally row).</td>
</tr>
<tr>
<td><code>bar([subset, axis, color, width, align])</code></td>
<td>Color the background color proportional to the values in each column.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>&quot;Reset&quot; the styler, removing any previously applied styles.</td>
</tr>
<tr>
<td><code>export()</code></td>
<td>Export the styles to applied to the current Styler.</td>
</tr>
<tr>
<td><code>formatter(formatter[, subset])</code></td>
<td>Format the text display value of cells.</td>
</tr>
<tr>
<td><code>from_custom_template(searchpath, name)</code></td>
<td>Factory function for creating a subclass of Styler with a custom template and Jinja environment.</td>
</tr>
<tr>
<td><code>highlight_max([subset, color, axis])</code></td>
<td>Highlight the maximum by shading the background</td>
</tr>
<tr>
<td><code>highlight_min([subset, color, axis])</code></td>
<td>Highlight the minimum by shading the background</td>
</tr>
<tr>
<td><code>highlight_null([null_color])</code></td>
<td>Shade the background null_color for missing values.</td>
</tr>
<tr>
<td><code>render(**kwargs)</code></td>
<td>Render the built up styles to HTML</td>
</tr>
<tr>
<td><code>set_caption(caption)</code></td>
<td>Set the caption on a Styler</td>
</tr>
<tr>
<td><code>set_precision(precision)</code></td>
<td>Set the precision used to render.</td>
</tr>
<tr>
<td><code>set_properties([subset])</code></td>
<td>Convenience method for setting one or more non-data dependent properties or each cell.</td>
</tr>
<tr>
<td><code>set_table_attributes(attributes)</code></td>
<td>Set the table attributes.</td>
</tr>
<tr>
<td><code>set_table_styles(table_styles)</code></td>
<td>Set the table styles on a Styler.</td>
</tr>
<tr>
<td><code>set_uuid(uuid)</code></td>
<td>Set the uuid for a Styler.</td>
</tr>
<tr>
<td><code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code></td>
<td>Write Styler to an excel sheet</td>
</tr>
<tr>
<td><code>use(styles)</code></td>
<td>Set the styles on the current Styler, possibly using styles from Styler.export.</td>
</tr>
<tr>
<td><code>where(cond, value[, other, subset])</code></td>
<td>Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.</td>
</tr>
</tbody>
</table>

### pandas.io.formats.style.Styler.apply

**Styler.apply** *(func, axis=0, subset=None, **kwargs)*  
Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.

New in version 0.17.1.

**Parameters**  
**func** : function  
*func* should take a Series or DataFrame (depending on *axis*), and return an object with the same shape. Must return a DataFrame with identical index and column labels when *axis*=None

**axis** : int, str or None  
*axis* to apply to each column (axis=0 or 'index') or to each row (axis=1 or 'columns') or to the entire DataFrame at once with *axis*=None
subset : IndexSlice
    a valid indexer to limit data to before applying the function. Consider using a pandas.IndexSlice

kwargs : dict
    pass along to func

Returns self : Styler

Notes

The output shape of func should match the input, i.e. if x is the input row, column, or table (depending on axis), then func(x.shape) == x.shape should be true.

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):
...     return ['background-color: yellow' if v == x.max() else ''
...             for v in x]
... <<<
>>> df = pd.DataFrame(np.random.randn(5, 2))
>>> df.style.apply(highlight_max)
```

pandas.io.formats.style.Styler.applymap

Styler.applymap(func, subset=None,**kwargs)

Apply a function elementwise, updating the HTML representation with the result.

New in version 0.17.1.

Parameters func : function
    func should take a scalar and return a scalar

subset : IndexSlice
    a valid indexer to limit data to before applying the function. Consider using a pandas.IndexSlice

kwargs : dict
    pass along to func

Returns self : Styler

See also:

Styler.where
pandas.io.formats.style.Styler.background_gradient

Styler.background_gradient(cmap='PuBu', low=0, high=0, axis=0, subset=None)

Color the background in a gradient according to the data in each column (optionally row). Requires matplotlib.

New in version 0.17.1.

Parameters

- cmap: str or colormap
  - matplotlib colormap
- low, high: float
  - compress the range by these values.
- axis: int or str
  - 1 or ‘columns’ for columnwise, 0 or ‘index’ for rowwise
- subset: IndexSlice
  - a valid slice for data to limit the style application to

Returns

- self : Styler

Notes

Tune low and high to keep the text legible by not using the entire range of the color map. These extend the range of the data by $\text{low} \times (x_{\text{max}} - x_{\text{min}})$ and $\text{high} \times (x_{\text{max}} - x_{\text{min}})$ before normalizing.

pandas.io.formats.style.Styler.bar

Styler.bar(subset=None, axis=0, color='#d65f5f', width=100, align='left')

Color the background color proportional to the values in each column. Excludes non-numeric data by default.

New in version 0.17.1.

Parameters

- subset: IndexSlice, default None
  - a valid slice for data to limit the style application to
- axis: int
- 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'])
- width: float
  - A number between 0 or 100. The largest value will cover width percent of the cell’s width
- align : {'left', ‘zero’, ‘mid’}, default ‘left’
  - ‘left’ : the min value starts at the left of the cell
  - ‘zero’ : a value of zero is located at the center of the cell
pandas: powerful Python data analysis toolkit, Release 0.21.0

- `mid`: the center of the cell is at \((\text{max}-\text{min})/2\), or if values are all negative (positive) the zero is aligned at the right (left) of the cell

New in version 0.20.0.

Returns `self`: Styler

**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 to applied to the current Styler. Can be applied to a second style with `Styler.use`.

New in version 0.17.1.

Returns `styles`: list

See also:

`Styler.use`

**pandas.io.formats.style.Styler.format**

Styler.format(`formatter`, `subset=None`)

Format the text display value of cells.

New in version 0.18.0.

Parameters

- `formatter`: str, callable, or dict
- `subset`: IndexSlice

An argument to `DataFrame.loc` that restricts which elements `formatter` is applied to.

Returns `self`: Styler

Notes

`formatter` is either an a or a dict `{column name: a}` where a is one of

- `str`: this will be wrapped in: `a.format(x)`
- `callable`: called with the value of an individual cell

The default display value for numeric values is the “general” \((g)\) format with `pd.options.display.precision`.

Examples
```python
>>> df = pd.DataFrame(np.random.randn(4, 2), columns=['a', 'b'])
>>> df.style.format('{:.2%}"
>>> df['c'] = ['a', 'b', 'c', 'd']
>>> df.style.format({'C': str.upper})
```

**pandas.io.formats.style.Styler.from_custom_template**

*classmethod* `Styler.from_custom_template(searchpath, name)`  
Factory function for creating a subclass of *Styler* with a custom template and Jinja environment.

**Parameters**  
`searchpath` : str or list  
Path or paths of directories containing the templates  
`name` : str  
Name of your custom template to use for rendering

**Returns**  
`MyStyler` : subclass of *Styler*  
has the correct *env* and *template* class attributes set.

**pandas.io.formats.style.Styler.highlight_max**

*Styler.highlight_max(subset=None, color='yellow', axis=0)*  
Highlight the maximum by shading the background  
New in version 0.17.1.

**Parameters**  
`subset` : IndexSlice, default None  
a valid slice for *data* to limit the style application to  
`color` : str, default ‘yellow’  
`axis` : int, str, or None; default 0  
0 or ‘index’ for columnwise (default), 1 or ‘columns’ for rowwise, or None for tablewise

**Returns**  
self : Styler

**pandas.io.formats.style.Styler.highlight_min**

*Styler.highlight_min(subset=None, color='yellow', axis=0)*  
Highlight the minimum by shading the background  
New in version 0.17.1.

**Parameters**  
`subset` : IndexSlice, default None  
a valid slice for *data* to limit the style application to  
`color` : str, default ‘yellow’  
`axis` : int, str, or None; default 0  
0 or ‘index’ for columnwise (default), 1 or ‘columns’ for rowwise, or None for tablewise
Returns self : Styler

**pandas.io.formats.style.Styler.highlight_null**

Styler.highlight_null(null_color='red')
Shade the background null_color for missing values.
New in version 0.17.1.

Parameters null_color: str
Returns self : Styler

**pandas.io.formats.style.Styler.render**

Styler.render(**kwargs)
Render the built up styles to HTML.
New in version 0.17.1.

Parameters **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.
New in version 0.20.

Returns rendered: str
the rendered HTML

Notes

Styler objects have defined the _repr_html_ method which automatically calls self.render() 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
* precision
* table_styles
* caption
* table_attributes
pandas.io.formats.style.Styler.set_caption

Styler.set_caption(caption)
Set the caption on a Styler
New in version 0.17.1.
Parameters caption: str
Returns self : Styler

pandas.io.formats.style.Styler.set_precision

Styler.set_precision(precision)
Set the precision used to render.
New in version 0.17.1.
Parameters precision: int
Returns self : Styler

pandas.io.formats.style.Styler.set_properties

Styler.set_properties(subset=None, **kwargs)
Convience method for setting one or more non-data dependent properties or each cell.
New in version 0.17.1.
Parameters subset: IndexSlice
  a valid slice for data to limit the style application to
kwargs: dict
  property: value pairs to be set for each cell
Returns self : Styler

Examples

>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_properties(color="white", align="right")
>>> df.style.set_properties(**{"background-color": "yellow"})

pandas.io.formats.style.Styler.set_table_attributes

Styler.set_table_attributes(attributes)
Set the table attributes. These are the items that show up in the opening <table> tag in addition to to automatic (by default) id.
New in version 0.17.1.
Parameters attributes : string
Returns self : Styler
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)

Set the table styles on a Styler. These are placed in a `<style>` tag before the generated HTML table.

New in version 0.17.1.

**Parameters**

- table_styles: 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).

**Returns**

- self: Styler

Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_styles(
...     [{'selector': 'tr:hover',
...      'props': [('background-color', 'yellow')]}]
...     )
```

**pandas.io.formats.style.Styler.set_uuid**

**Styler.set_uuid**(uuid)

Set the uuid for a Styler.

New in version 0.17.1.

**Parameters**

- uuid: str

**Returns**

- self: Styler

**pandas.io.formats.style.Styler.to_excel**

**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

New in version 0.20.

**Parameters**

- excel_writer: string or ExcelWriter object

  File path or existing ExcelWriter
sheet_name: string, default ‘Sheet1’
   Name of sheet which will contain DataFrame

na_rep: string, default ‘’
   Missing data representation

float_format: string, default None
   Format string for floating point numbers

columns: sequence, optional
   Columns to write

header: boolean or list of string, 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: boolean, default True
   Write row names (index)

index_label: string or sequence, 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 DataFrame uses MultiIndex.

startrow:
   upper left cell row to dump data frame

startcol:
   upper left cell column to dump data frame

engine: string, default None
   write engine to use - you can also set this via the options io.excel.xlsx.writer, io.excel.xls.writer, and io.excel.xlsm.writer.

merge_cells: boolean, default True
   Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None
   encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep: string, default ‘inf’
   Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes: tuple of integer (length 2), default None
   Specifies the one-based bottommost row and rightmost column that is to be frozen
   New in version 0.20.0.

Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:
```python
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with `to_csv`, `to_excel` serializes lists and dicts to strings before writing.

**pandas.io.formats.style.Styler.use**

`Styler.use(styles)`

Set the styles on the current Styler, possibly using styles from `Styler.export`.

New in version 0.17.1.

**Parameters**

- `styles`: list
  - list of style functions

**Returns**

- `self`: Styler

**See also:**

`Styler.export`

**pandas.io.formats.style.Styler.where**

`Styler.where(cond, value=None, other=None, subset=None, **kwargs)`

Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.

New in version 0.21.0.

**Parameters**

- `cond`: callable
  - `cond` should take a scalar and return a boolean
- `value`: str
  - applied when `cond` returns true
- `other`: str
  - applied when `cond` returns false
- `subset`: IndexSlice
  - a valid indexer to limit data to before applying the function. Consider using a pandas.IndexSlice
- `**kwargs`: dict
  - pass along to `cond`

**Returns**

- `self`: Styler

**See also:**

`Styler.applymap`
34.18.2 Style Application
pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Styler.apply</code> (func[, axis, subset])</td>
<td>Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.</td>
</tr>
<tr>
<td><code>Styler.applymap</code> (func[, subset])</td>
<td>Apply a function elementwise, updating the HTML representation with the result.</td>
</tr>
<tr>
<td><code>Styler.where</code> (cond, value[, other, subset])</td>
<td>Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.</td>
</tr>
<tr>
<td><code>Styler.format</code> (formatter[, subset])</td>
<td>Format the text display value of cells.</td>
</tr>
<tr>
<td><code>Styler.set_precision</code> (precision)</td>
<td>Set the precision used to render.</td>
</tr>
<tr>
<td><code>Styler.set_table_styles</code> (table_styles)</td>
<td>Set the table styles on a Styler.</td>
</tr>
<tr>
<td><code>Styler.set_properties</code> ([subset])</td>
<td>Convenience method for setting one or more non-data dependent properties or each cell.</td>
</tr>
<tr>
<td><code>Styler.set_uuid</code> (uuid)</td>
<td>Set the uuid for a Styler.</td>
</tr>
<tr>
<td><code>Styler.clear</code> ()</td>
<td>“Reset” the styler, removing any previously applied styles.</td>
</tr>
</tbody>
</table>

### 34.18.3 Builtin Styles

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Styler.highlight_max</code> ([subset, color, axis])</td>
<td>Highlight the maximum by shading the background</td>
</tr>
<tr>
<td><code>Styler.highlight_min</code> ([subset, color, axis])</td>
<td>Highlight the minimum by shading the background</td>
</tr>
<tr>
<td><code>Styler.highlight_null</code> ([null_color])</td>
<td>Shade the background null_color for missing values.</td>
</tr>
<tr>
<td><code>Styler.background_gradient</code> ([cmap, low, ...])</td>
<td>Color the background in a gradient according to the data in each column (optionally row).</td>
</tr>
<tr>
<td><code>Styler.bar</code> ([subset, axis, color, width, align])</td>
<td>Color the background color proptional to the values in each column.</td>
</tr>
</tbody>
</table>

### 34.18.4 Style Export and Import

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Styler.render</code> (<strong>kwargs</strong>)</td>
<td>Render the built up styles to HTML</td>
</tr>
<tr>
<td><code>Styler.export</code> ()</td>
<td>Export the styles to applied to the current Styler.</td>
</tr>
<tr>
<td><code>Styler.use</code> (styles)</td>
<td>Set the styles on the current Styler, possibly using styles from <code>Styler.export</code>.</td>
</tr>
</tbody>
</table>

### 34.19 General utility functions

#### 34.19.1 Working with options

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>describe_option</code> (pat[, _print_desc])</td>
<td>Prints the description for one or more registered options.</td>
</tr>
<tr>
<td><code>reset_option</code> (pat)</td>
<td>Reset one or more options to their default value.</td>
</tr>
<tr>
<td><code>get_option</code> (pat)</td>
<td>Retrieves the value of the specified option.</td>
</tr>
<tr>
<td><code>set_option</code> (pat, value)</td>
<td>Sets the value of the specified option.</td>
</tr>
<tr>
<td><code>option_context</code> (*args)</td>
<td>Context manager to temporarily set options in the with statement context.</td>
</tr>
</tbody>
</table>
34.19.1.1 pandas.describe_option

```python
pandas.describe_option(pat, _print_desc=False) = <pandas.core.config.CallableDynamicDoc object>
```

Prints the description for one or more registered options.

Call with not arguments to get a listing for all registered options.

Available options:

- `compute.use_bottleneck, use_numexpr`
- `display.chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height`
- `display.html.border, table_schema`
- `display.large_repr`
- `display.latex.escape, longtable, multicolumn, multicolumn_format, multirow, repr`
- `display.line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions`
- `display.unicode.ambiguous_as_wide, east_asian_width`
- `display.width`
- `html.border`
- `io.excel.xls.writer`
- `io.excel.xlsm.writer`
- `io.excel.xlsx.writer`
- `io.hdf.default_format, dropna_table`
- `io.parquet.engine`
- `mode.chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null`

**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**

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_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.height [int] Deprecated. [default: 60] [currently: 60]

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.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: 1] [currently: 1]

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.line_width [int] Deprecated. [default: 80] [currently: 80]

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: 20] [currently: 20]

**display.max_colwidth** [int] 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. [default: 50] [currently: 50]

**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: 15]

**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.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 (number of significant digits). This is only a suggestion [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]
html.border  [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use display.html.border instead.)


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.sim_interactive  [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

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 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.)

### 34.19.1.2 pandas.reset_option

pandas.reset_option(pat) = <pandas.core.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_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height]
- display.html.[border, table_schema]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]

**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**

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_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.height** [int] Deprecated. [default: 60] [currently: 60]

- **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.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.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.line_width  [int] Deprecated. [default: 80] [currently: 80]

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: 20] [currently: 20]

display.max_colwidth  [int] 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. [default: 50] [currently: 50]

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: 15]

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.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 (number of significant digits). This is only a suggestion [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]

**html.border** [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use display.html.border instead.)


**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.sim_interactive** [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

**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 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.)

## 34.19.1.3 pandas.get_option

**pandas.get_option(pat)** = `<pandas.core.config.CallableDynamicDoc object>`

Retrieves the value of the specified option.

**Available options:**

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height]
- display.html.[border, table_schema]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xls.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]

**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**

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_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.height [int] Deprecated. [default: 60] [currently: 60]

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.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: True] [currently: True]

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.line_width [int] Deprecated. [default: 80] [currently: 80]

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: 20] [currently: 20]

display.max_colwidth [int] 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. [default: 50] [currently: 50]

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: 15]

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.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 (number of significant digits). This is only a suggestion [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]

html.border [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr.
[default: 1] [currently: 1] (Deprecated, use display.html.border instead.)


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.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

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 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.)

### 34.19.1.4 pandas.set_option

pandas.set_option(pat, value) = <pandas.core.config.CallableDynamicDoc object>

Sets the value of the specified option.

Available options:

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height]
- display.html.[border, table_schema]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
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`: new value of option.

**Returns** None

**Raises** `OptionError` if no such option exists

**Notes**

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_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.height  [int] Deprecated. [default: 60] [currently: 60]

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.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.line_width  [int] Deprecated. [default: 80] [currently: 80]

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: 20] [currently: 20]

display.max_colwidth  [int] 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. [default: 50] [currently: 50]

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: 15]
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.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 (number of significant digits). This is only a suggestion [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]

html.border [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use display.html.border instead.)


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.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]
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 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.)

34.19.1.5 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

>>> with option_context('display.max_rows', 10, 'display.max_columns', 5):
...  

34.19.2 Testing functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing.assert_frame_equal</td>
<td>Check that left and right DataFrame are equal.</td>
</tr>
<tr>
<td>testing.assert_series_equal</td>
<td>Check that left and right Series are equal.</td>
</tr>
<tr>
<td>testing.assert_index_equal</td>
<td>Check that left and right Index are equal.</td>
</tr>
</tbody>
</table>

34.19.2.1 pandas.testing.assert_frame_equal

pandas.testing.assert_frame_equal(left, right[, ...])  
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left, right</td>
<td>DataFrame</td>
</tr>
<tr>
<td>check_dtypes</td>
<td>bool, default True</td>
</tr>
<tr>
<td>check_index_types</td>
<td>bool / string {'equiv'}, default False</td>
</tr>
<tr>
<td>check_column_types</td>
<td>bool / string {'equiv'}, default False</td>
</tr>
<tr>
<td>check_frame_types</td>
<td>bool, default False</td>
</tr>
</tbody>
</table>

Check that left and right DataFrame are equal.

Parameters left : DataFrame

    right : DataFrame
    check_dtypes : bool, default True
        Whether to check the DataFrame dtype is identical.
    check_index_types : bool / string {'equiv'}, default False
        Whether to check the Index class, dtype and inferred_type are identical.
    check_column_types : bool / string {'equiv'}, default False
        Whether to check the columns class, dtype and inferred_type are identical.
    check_frame_types : bool, default False
        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
check_names : bool, default True
    Whether to check the Index names attribute.
by_blocks : bool, default False
    Specify how to compare internal data. If False, compare by columns. If True, com-
    pare 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 rows & columns
obj : str, default ‘DataFrame’
    Specify object name being compared, internally used to show appropriate assertion
    message

34.19.2.2 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=False,
check_names=True, check_exact=False,
check_datetimelike_compat=False,
check_categorical=True, obj='Series')

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 / string {'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
check_exact : bool, default False
Whether to compare number exactly.

check_names : bool, default True
Whether to check the Series and Index names attribute.

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.

obj : str, default ‘Series’
Specify object name being compared, internally used to show appropriate assertion message

34.19.2.3 pandas.testing.assert_index_equal

pandas.testing.assert_index_equal(left, right, exact='equiv', check_names=True,
check_less_precise=False, check_exact=True, check_categorical=True, obj='Index')

Check that left and right Index are equal.

Parameters left : Index
right : Index
exact : bool / string {‘equiv’}, default False
Whether to check the Index class, dtype and inferred_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.

check_exact : bool, default True
Whether to compare number exactly.

check_categorical : bool, default True
Whether to compare internal Categorical exactly.

obj : str, default ‘Index’
Specify object name being compared, internally used to show appropriate assertion message

Continued on next page
### 34.19.3 Exceptions and warnings

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errors.DtypeWarning</code></td>
<td>Warning that is raised for a dtype incompatibility. This can happen whenever <code>pd.read_csv</code> encounters non-uniform dtypes in a column(s) of a given CSV file.</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.OutOfBoundsDatetime</code></td>
<td>Exception that is raised by an error encountered in <code>pd.read_csv</code>.</td>
</tr>
<tr>
<td><code>errors.ParserError</code></td>
<td>Warning that is raised in <code>pd.read_csv</code> whenever it is necessary to change parsers (generally from ‘c’ to ‘python’) contrary to the one specified by the user due to lack of support or functionality for parsing particular attributes of a CSV file with the requested engine.</td>
</tr>
<tr>
<td><code>errors.ParserWarning</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>

#### 34.19.3.1 pandas.errors.DtypeWarning

**exception pandas.errors.DtypeWarning**

Warning that is raised for a dtype incompatibility. This can happen whenever `pd.read_csv` encounters non-uniform dtypes in a column(s) of a given CSV file.

#### 34.19.3.2 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.

#### 34.19.3.3 pandas.errors.OutOfBoundsDatetime

**exception pandas.errors.OutOfBoundsDatetime**

#### 34.19.3.4 pandas.errors.ParserError

**exception pandas.errors.ParserError**

Exception that is raised by an error encountered in `pd.read_csv`.

#### 34.19.3.5 pandas.errors.ParserWarning

**exception pandas.errors.ParserWarning**

Warning that is raised in `pd.read_csv` whenever it is necessary to change parsers (generally from ‘c’ to ‘python’)
contrary to the one specified by the user due to lack of support or functionality for parsing particular attributes of a CSV file with the requested engine.

### 34.19.3.6 pandas.errors.PerformanceWarning

**exception** `pandas.errors.PerformanceWarning`

Warning raised when there is a possible performance impact.

### 34.19.3.7 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`.

New in version 0.20.0.

### 34.19.3.8 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)`.

### 34.19.4 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>Converts input into a pandas only dtype object or a numpy dtype object.</td>
</tr>
</tbody>
</table>

#### 34.19.4.1 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.

New in version 0.19.0.

**Parameters**

- `to_union` : list-like of Categorical, CategoricalIndex, or Series with dtype='category'
- `sort_categories` : boolean, default False
  
  If true, resulting categories will be lexsorted, otherwise they will be ordered as they appear in the data.
- `ignore_order` : boolean, default False
  
  If true, the ordered attribute of the Categoricals will be ignored. Results in an unordered categorical.

New in version 0.20.0.

**Returns**

- `result` : 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
>>> a = pd.Categorical(['b', 'c'])
>>> b = pd.Categorical(['a', 'b'])
>>> union_categoricals([a, b])
[b, c, a]
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
>>> union_categoricals([a, b], sort_categories=True)
[b, c, a, b]
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
>>> a = pd.Categorical(['a', 'b'], ordered=True)
>>> b = pd.Categorical(['a', 'b', 'a'], ordered=True)
>>> union_categoricals([a, b])
[a, b, a, b]
Categories (2, object): [a < b]
```

**Raises** TypeError because the categories are ordered and not identical.

```python
>>> a = pd.Categorical(['a', 'b'], ordered=True)
>>> b = pd.Categorical(['a', 'b', 'c'], ordered=True)
>>> union_categoricals([a, b])
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.
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
>>> 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]
```

34.19.4.2 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 False

Ignore NaN values when inferring the type. The default of `False` will be deprecated in a later version of pandas.

New in version 0.21.0.

**Returns**

- string describing the common type of the input data.

Results can include:

- string
- unicode
- bytes
- floating
- integer
- mixed-integer
- mixed-integer-float
- decimal
- complex
- categorical
- boolean
- datetime64
- datetime
- date
- timedelta64
- timedelta
• time
• period
• mixed

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

Examples

```python
>>> infer_dtypes(['foo', 'bar'])
'string'

>>> infer_dtypes(['a', np.nan, 'b'], skipna=True)
'string'

>>> infer_dtypes(['a', np.nan, 'b'], skipna=False)
'mixed'

>>> infer_dtypes([b'foo', b'bar'])
'bytes'

>>> infer_dtypes([1, 2, 3])
'integer'

>>> infer_dtypes([1, 2, 3.5])
'mixed-integer-float'

>>> infer_dtypes([1.0, 2.0, 3.5])
'floating'

>>> infer_dtypes(['a', 1])
'mixed-integer'

>>> infer_dtypes([Decimal(1), Decimal(2.0)])
'decimal'

>>> infer_dtypes([True, False])
'boolean'

>>> infer_dtypes([True, False, np.nan])
'mixed'
```
>>> infer_dtypes([pd.Timestamp('20130101')])
'datetime'

>>> infer_dtypes([datetime.date(2013, 1, 1)])
'date'

>>> infer_dtypes([np.datetime64('2013-01-01')])
'datetime64'

>>> infer_dtypes([datetime.timedelta(0, 1, 1)])
'timedelta'

>>> infer_dtypes(pd.Series(list('aabc')).astype('category'))
'categorical'

34.19.4.3 pandas.api.types.pandas_dtypes

pandas.api.types.pandas_dtypes(dtype)

Converts 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

Dtype introspection

api.types.is_bool_dtype(arr_or_dtype) Check whether the provided array or dtype is of a boolean dtype.
api.types.is_categorical_dtype(arr_or_dtype) Check whether an array-like or dtype is of the Categorical dtype.
api.types.is_complex_dtype(arr_or_dtype) Check whether the provided array or dtype is of a complex dtype.
api.types.is_datetime64_any_dtype(arr_or_dtype) Check whether the provided array or dtype is of the datetime64 dtype.
api.types.is_datetime64_dtype(arr_or_dtype) Check whether an array-like or dtype is of the datetime64 dtype.
api.types.is_datetime64_ns_dtype(arr_or_dtype) Check whether the provided array or dtype is of the datetime64[ns] dtype.
api.types.is_datetime64tz_dtype(arr_or_dtype) Check whether an array-like or dtype is of a DatetimeTZD-type dtype.
api.types.is_extension_type(arr) Check whether an array-like is of a pandas extension class instance.
api.types.is_float_dtype(arr_or_dtype) Check whether the provided array or dtype is of a float dtype.
api.types.is_int64_dtype(arr_or_dtype) Check whether the provided array or dtype is of the int64 dtype.
api.types.is_integer_dtype(arr_or_dtype) Check whether the provided array or dtype is of an integer dtype.
api.types.is_interval_dtype(arr_or_dtype) Check whether an array-like or dtype is of the Interval dtype.
api.types.is_numeric_dtype(arr_or_dtype) Check whether the provided array or dtype is of a numeric dtype.

Continued on next page
Table 34.166 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api.types.is_object_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</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</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</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</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</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</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</code></td>
<td>Check whether an array-like is a pandas sparse array.</td>
</tr>
</tbody>
</table>

34.19.4.4 pandas.api.types.is_bool_dtype

```python
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

The array or dtype to check.

**Returns**

`boolean` : Whether or not the array or dtype is of a boolean dtype.

**Examples**

```python
golden >> is_bool_dtype(str)
False
>>> is_bool_dtype(int)
False
>>> is_bool_dtype(bool)
True
>>> is_bool_dtype(np.bool)
True
>>> is_bool_dtype(np.array(['a', 'b']))
False
>>> is_bool_dtype(pd.Series([1, 2]))
False
>>> is_bool_dtype(np.array([True, False]))
True
```

34.19.4.5 pandas.api.types.is_categorical_dtype

```python
pandas.api.types.is_categorical_dtype(arr_or_dtyple)
```

Check whether an array-like or dtype is of the Categorical dtype.

**Parameters**

`arr_or_dtyple` : array-like

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
```

34.19.4.6 pandas.api.types.is_complex_dtype

```python
def is_complex_dtype(arr_or_dtype):
    
    Parameters
    ----------
    arr_or_dtype : array-like
        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
```

34.19.4.7 pandas.api.types.is_datetime64_any_dtype

```python
def is_datetime64_any_dtype(arr_or_dtype):
    
    Parameters
    ----------
    arr_or_dtype : array-like
        The array or dtype to check.

    Returns
    -------
    boolean : Whether or not the array or dtype is of a datetime64 dtype.
```

Examples

```python
```
34.19.4.8 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
  The array-like or dtype to check.

Returns boolean : Whether or not the array-like or dtype is of the datetime64 dtype.

Examples

>>> 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

34.19.4.9 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
  The array or dtype to check.
Returns boolean: Whether or not the array or dtype is of the datetime64[ns] dtype.

Examples

```python
>>> 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=np.datetime64))  # no unit
False
>>> is_datetime64_ns_dtype(np.array([], dtype="datetime64[ps]"))  # wrong unit
False
>>> is_datetime64_ns_dtype(pd.DatetimeIndex([1, 2, 3],
                   dtype=np.datetime64))  # has 'ns' unit
True
```

34.19.4.10 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

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)
is_datetime64tz_dtype(dtype)
True
>>> is_datetime64tz_dtype(s)
True
```
### 34.19.4.11 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.

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
  
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.SparseArray([1, 2, 3]))
True
>>> is_extension_type(pd.SparseSeries([1, 2, 3]))
True
>>> from scipy.sparse import bsr_matrix
>>> is_extension_type(bsr_matrix([1, 2, 3]))
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
```

### 34.19.4.12 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.

**Parameters**

- **arr_or_dtype**: array-like
  
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
```

34.19.4.13 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
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(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
```
34.19.4.14 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 in_any_int_dtype, timedelta64 instances will return False.

Parameters arr_or_dtype : array-like
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)
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
```

34.19.4.15 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
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())
False
```
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

34.19.4.16 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
    The array or dtype to check.

Returns boolean : Whether or not the array or dtype is of a numeric dtype.

Examples

>>> 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

34.19.4.17 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
    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
```

34.19.4.18 pandas.api.types.is_period_dtype

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
    The array-like or dtype to check.

Returns
boolean : Whether or not the array-like or dtype is of the Period dtype.

Examples

```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
```

34.19.4.19 pandas.api.types.is_signed_integer_dtype

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 `in_any_int_dtype`, timedelta64 instances will return False.

Parameters
arr_or_dtype : array-like
    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.
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(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
```

### 34.19.4.20 pandas.api.types.is_string_dtypes

**pandas.api.types.is_string_dtypes**(*arr_or_dtype*)

Check whether the provided array or dtype is of the string dtype.

**Parameters**

- **arr_or_dtype** : array-like

  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
```

### 34.19.4.21 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

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
```

### 34.19.4.22 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

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
```

### 34.19.4.23 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 **arr_or_dtype** : array-like

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(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
```

### 34.19.4.24 pandas.api.types.is_sparse

pandas.api.types.is_sparse(arr)

Check whether an array-like is a pandas sparse array.

**Parameters**

arr : array-like

The array-like to check.

**Returns** boolean : Whether or not the array-like is a pandas sparse array.

**Examples**

```python
>>> is_sparse(np.array([1, 2, 3]))
False
>>> is_sparse(pd.SparseArray([1, 2, 3]))
True
>>> is_sparse(pd.SparseSeries([1, 2, 3]))
True
```

This function checks only for pandas sparse array instances, so sparse arrays from other libraries will return False.

```python
>>> from scipy.sparse import bsr_matrix
>>> is_sparse(bsr_matrix([1, 2, 3]))
False
```

Iterable introspection

**api.types.is_dict_like**(obj) Check if the object is dict-like.

Continued on next page
pandas: powerful Python data analysis toolkit, Release 0.21.0

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_file_like(obj)</code></td>
<td>Check if the object is a file-like object.</td>
</tr>
<tr>
<td><code>is_list_like(obj)</code></td>
<td>Check if the object is list-like.</td>
</tr>
<tr>
<td><code>is_named_tuple(obj)</code></td>
<td>Check if the object is a named tuple.</td>
</tr>
<tr>
<td><code>is_iterator(obj)</code></td>
<td>Check if the object is an iterator.</td>
</tr>
</tbody>
</table>

### 34.19.4.25 pandas.api.types.is_dict_like

```python
def is_dict_like(obj):
    Parameters
    ----------
    obj : The object to check.

    Returns
    -------
    is_dict_like : bool
    Whether `obj` has dict-like properties.
```

**Examples**

```python
globals().update(locals())
>>> is_dict_like({1: 2})
True
>>> is_dict_like([1, 2, 3])
False
```

### 34.19.4.26 pandas.api.types.is_file_like

```python
def is_file_like(obj):
    Parameters
    ----------
    obj : The object to check.

    Returns
    -------
    is_file_like : bool
    Whether `obj` has file-like properties.
```

**Examples**

```python
globals().update(locals())
>>> buffer(StringIO("data"))
>>> is_file_like(buffer)
True
>>> is_file_like([1, 2, 3])
False
```
34.19.4.27 pandas.api.types.is_list_like

pandas.api.types.is_list_like(obj)
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 : The object to check.

Returns  is_list_like : bool
    Whether obj has list-like properties.

Examples

    >>> is_list_like([1, 2, 3])
    True
    >>> is_list_like((1, 2, 3))
    True
    >>> is_list_like(datetime(2017, 1, 1))
    False
    >>> is_list_like("foo")
    False
    >>> is_list_like(1)
    False

34.19.4.28 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

    >>> Point = namedtuple("Point", ["x", "y"])  
    >>> p = Point(1, 2)
    >>> is_named_tuple(p)
    True
    >>> is_named_tuple((1, 2))
    False

34.19.4.29 pandas.api.types.is_iterator

pandas.api.types.is_iterator(obj)
Check if the object is an iterator.

    For example, lists are considered iterators but not strings or datetime objects.
Parameters `obj` : The object to check.

Returns `is_iter` : bool

Whether `obj` is an iterator.

Examples

```python
>>> is_iterator([1, 2, 3])
True
>>> is_iterator(datetime(2017, 1, 1))
False
>>> is_iterator("foo")
False
>>> is_iterator(1)
False
```

Scalar introspection

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api.types.is_bool</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_categorical</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_complex</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_datetime</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_float</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_hashable</code></td>
<td>Return True if hash(obj) will succeed, False otherwise.</td>
</tr>
<tr>
<td><code>api.types.is_integer</code></td>
<td>Check if the object is a number.</td>
</tr>
<tr>
<td><code>api.types.is_interval</code></td>
<td>Check if the object is a periodical index.</td>
</tr>
<tr>
<td><code>api.types.is_period</code></td>
<td>Check if the object is a periodical index.</td>
</tr>
<tr>
<td><code>api.types.is_re</code></td>
<td>Check if the object is a regex pattern instance.</td>
</tr>
<tr>
<td><code>api.types.is_re_compilable</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 value is scalar.</td>
</tr>
</tbody>
</table>

### 34.19.4.30 pandas.api.types.is_bool

pandas.api.types.is_bool()

### 34.19.4.31 pandas.api.types.is_categorical

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
```

34.19.4.32 pandas.api.types.is_complex

pandas.api.types.is_complex()

34.19.4.33 pandas.api.types.is_datetimetz

pandas.api.types.is_datetimetz(arr)

Check whether an array-like is a datetime array-like with a timezone component in its dtype.

Parameters

arr : array-like

The array-like to check.

Returns

boolean : Whether or not the array-like is a datetime array-like with

a timezone component in its dtype.

Examples

```python
>>> is_datetimetz([1, 2, 3])
False
```

Although the following examples are both DatetimeIndex objects, the first one returns False because it has no
timezone component unlike the second one, which returns True.

```python
>>> is_datetimetz(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_datetimetz(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

The object need not be a DatetimeIndex object. It just needs to have a dtype which has a timezone component.

```python
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_datetimetz(s)
True
```
34.19.4.34 pandas.api.types.is_float

pandas.api.types.is_float()

34.19.4.35 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.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.

Examples

```python
>>> a = ([],)
>>> isinstance(a, collections.Hashable)
True
>>> is_hashable(a)
False
```

34.19.4.36 pandas.api.types.is_integer

pandas.api.types.is_integer()

34.19.4.37 pandas.api.types.is_interval

pandas.api.types.is_interval()

34.19.4.38 pandas.api.types.is_number

pandas.api.types.is_number(obj)

Check if the object is a number.

Parameters

- obj : The object to check.

Returns

- is_number : bool

Whether obj is a number or not.

Examples

```python
>>> is_number(1)
True
>>> is_number("foo")
False
```
34.19.4.39 pandas.api.types.is_period

```
pandas.api.types.is_period(arr)
Check whether an array-like is a periodical index.

Parameters:
arr : array-like
    The array-like to check.

Returns:
    boolean : Whether or not the array-like is a periodical index.
```

Examples

```python
>>> is_period([1, 2, 3])
False
>>> is_period(pd.Index([1, 2, 3]))
False
>>> is_period(pd.PeriodIndex(['2017-01-01'], freq="D"))
True
```

34.19.4.40 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_regex : bool
        Whether obj is a regex pattern.
```

Examples

```python
>>> is_re(re.compile(".*"))
True
>>> is_re("foo")
False
```

34.19.4.41 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
```

34.19.4.42 pandas.api.types.is_scalar

pandas.api.types.is_scalar()

Return True if given value is scalar.

This includes:
- numpy array scalar (e.g. np.int64)
- Python builtin numerics
- Python builtin byte arrays and strings
- None
- instances of datetime.datetime
- instances of datetime.timedelta
- Period
- instances of decimal.Decimal
- Interval
This section will focus on downstream applications of pandas.

### 35.1 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:

```plaintext
5: optional list<KeyValue> key_value_metadata
```

where `KeyValue` is

```plaintext
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:

```plaintext
{'index_columns': ['__index_level_0__', '__index_level_1__', ...],
 'column_indexes': [<ci0>, <ci1>, ..., <ciN>],
 'columns': [<c0>, <c1>, ...],
 'pandas_version': $VERSION}
```

Here, `<c0>`/<ci0> and so forth are dictionaries containing the metadata for each column. This has JSON form:

```json
{"name": column_name,
 'pandas_type': pandas_type,
 'numpy_type': numpy_type,
 'metadata': 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'

---

2003
- **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'
  - 'msgpack'
  - '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,
   'pandas_type': 'string',
   'numpy_type': 'object',
   'metadata': None}
 ],
'columns': [ 
 { 'name': 'c0',
   'pandas_type': 'int8',
   'numpy_type': 'int8',
   'metadata': None},
 { 'name': 'c1',
   'pandas_type': 'bytes',
   'numpy_type': 'object',
   'metadata': None},
 { 'name': 'c2',
   'pandas_type': 'categorical',
   'numpy_type': 'int16',
   'metadata': { 'num_categories': 1000, 'ordered': False}},
 { 'name': 'c3',
   'pandas_type': 'datetimetz',
   'numpy_type': 'datetime64[ns]',
   'metadata': { 'timezone': 'America/Los_Angeles'}}
}
35.1. Storing pandas DataFrame objects in Apache Parquet format
This section will provide a look into some of pandas internals.

### 36.1 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 datetime 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

## 36.1.1 MultiIndex

Internally, the MultiIndex consists of a few things: the **levels**, the integer **labels**, and the level **names**:

```python
In [1]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first', 'second'])
In [2]: index
Out[2]: MultiIndex(levels=[[0, 1, 2], ['one', 'two']], labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]], names=['first', 'second'])

In [3]: index.levels
Out[3]: FrozenList([[0, 1, 2], ['one', 'two']])

In [4]: index.labels
Out[4]: FrozenList([[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])

In [5]: index.names
Out[5]: FrozenList(['first', 'second'])
```

You can probably guess that the labels 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 labels 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 labels yourself, please be careful.

## 36.2 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.

This section describes how to subclass pandas data structures to meet more specific needs. There are 2 points which need attention:

1. Override constructor properties.
2. Define original properties
36.2.1 Override Constructor Properties

Each data structure has constructor properties to specifying data constructors. By overriding these properties, you can retain defined-classes through pandas data manipulations.

There are 3 constructors to be defined:

- `__constructor`: Used when a manipulation result has the same dimensions as the original.
- `__constructor_sliced`: Used when a manipulation result has one lower dimension(s) as the original, such as DataFrame single columns slicing.
- `__constructor_expanddim`: Used when a manipulation result has one higher dimension as the original, such as Series.to_frame() and DataFrame.to_panel().

Following table shows how pandas data structures define constructor properties by default.

<table>
<thead>
<tr>
<th>Property Attributes</th>
<th>Series</th>
<th>DataFrame</th>
<th>Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>__constructor</td>
<td>Series</td>
<td>DataFrame</td>
<td>Panel</td>
</tr>
<tr>
<td>__constructor_sliced</td>
<td>NotImplemented</td>
<td>Series</td>
<td>DataFrame</td>
</tr>
<tr>
<td>__constructor_expanddim</td>
<td>DataFrame</td>
<td>Panel</td>
<td>NotImplemented</td>
</tr>
</tbody>
</table>

Below example shows how to define SubclassedSeries and SubclassedDataFrame overriding constructor properties.

```python
class SubclassedSeries(Series):
    @property
def __constructor(self):
        return SubclassedSeries

    @property
def __constructor_expanddim(self):
        return SubclassedDataFrame

class SubclassedDataFrame(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
```

36.2. Subclassing pandas Data Structures
36.2.2 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 __getattr__. 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 2 original properties, “internal_cache” as a temporary property and “added_property” as a normal property

```python
class SubclassedDataFrame2(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
```

```python
>>> df = SubclassedDataFrame2({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
```

```plaintext
   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
RELEASE NOTES

This is the list of changes to pandas between each release. For full details, see the commit logs at http://github.com/pandas-dev/pandas

What is it

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.

Where to get it

- Source code: http://github.com/pandas-dev/pandas
- Binary installers on PyPI: http://pypi.python.org/pypi/pandas
- Documentation: http://pandas.pydata.org

37.1 pandas 0.21.0

Release date: 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, 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.

See the v0.21.0 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.21.0
37.1.1 Thanks

A total of 206 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

37.1.1.1 Contributors

- 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
• 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 Battistoni
• 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 +
• abarber4gh +
• aermlund +
• agustín ménendez +
• andymaheshw +
• ante328 +
• aviolov +
• bpraggastis
• cbertinato +
• cclauss +
• chernick
• chris-b1
• dkamm +
• dwkenefick
• economy
• faic +
• fding253 +
• gfyoun
• guygoldberg +
• hhuuggoo +
• huashuai +
• ian
• iulia +
• jaredsnyder
• jbrockmendel +
• jdeschenes
• jebob +
• jschendel +
- keitakurita
- kerne +
- kiwirob +
- kjford
- linebp
- lloydkirk
- louispotok +
- majiang +
- manikbhandari +
- matthiashuschle +
- mattip
- maxwasserman +
- mjlove12 +
- nmartensen +
- pandas-docs-bot +
- parchd-l +
- philipphanemann +
- rdk1024 +
- reidy-p +
- ri938
- ruiann +
- rvernica +
- s-weigand +
- scotthavard92 +
- skwbc +
- step4me +
- tobycheese +
- topper-123 +
- tsdlovell
- ysau +
- zzgao +

37.2 pandas 0.20.0 / 0.20.1

Release date: 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

See the v0.20.1 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.20.1.

**Note:** This is a combined release for 0.20.0 and and 0.20.1. Version 0.20.1 contains one additional change for backwards-compatibility with downstream projects using pandas’ `utils` routines. (GH16250)

### 37.2.1 Thanks

- abaldenko
- Adam J. Stewart
- Adrian
- adrian-stepien
- Ajay Saxena
- Akash Tandon
- Albert Villanueva del Moral
- Aleksey Bilogur
- alexandercbooth
- Alexis Mignon
- Amol Kahat
- Andreas Winkler
- Andrew Kittredge
• Anthonios Partheniou
• Arco Bast
• Ashish Singal
• atbd
• bastewart
• Baurzhan Muftakhidinov
• Ben Kandel
• Ben Thayer
• Ben Welsh
• Bill Chambers
• bmagnusson
• Brandon M. Burroughs
• Brian
• Brian McFee
• carlosdanielcsantos
• Carlos Souza
• chaimdemulder
• Chris
• chris-b1
• Chris Ham
• Christopher C. Aycock
• Christoph Gohlke
• Christoph Paulik
• Chris Warth
• Clemens Brunner
• DaanVanHauwermeiren
• Daniel Himmelstein
• Dave Willmer
• David Cook
• David Gwynne
• David Hoffman
• David Krych
• dickreuter
• Diego Fernandez
• Dimitris Spathis
• discort
[Names of contributors]
• manuels
• Mark Mandel
• Matthew Brett
• Matthew Roeschke
• mattip
• Matti Picus
• Matt Roeschke
• maxalbert
• Maximilian Roos
• mcocdawc
• Michael Charlton
• Michael Felt
• Michael Lamparski
• Michiel Stock
• Mikolaj Chwalisz
• Min RK
• Miroslav Šedivý
• Mykola Golubyev
• Nate Yoder
• Nathalie Rud
• Nicholas Ver Halen
• Nick Chmura
• Nolan Nichols
• nuffe
• Pankaj Pandey
• paul-mannino
• Pawel Kordek
• pbreach
• Pete Huang
• Peter
• Peter Csizsek
• Petio Petrov
• Phil Ruffwind
• Pietro Battiston
• Piotr Chromiec
• Prasanjit Prakash
• Robert Bradshaw
• Rob Forgione
• Robin
• Rodolfo Fernandez
• Roger Thomas
• Rouz Azari
• Sahil Dua
• sakkemo
• Sam Foo
• Sami Salonen
• Sarah Bird
• Sarma Tangirala
• scls19fr
• Scott Sanderson
• Sebastian Bank
• Sebastian Gsänger
• Sébastien de Menten
• Shawn Heide
• Shyam Saladi
• sinhcrk
• Sinhrks
• Stephen Rauch
• stijnvanhoeoy
• Tara Adiseshan
• themrmax
• the-nose-knows
• Thiago Serafim
• Thoralf Gutierrez
• Thrasibule
• Tobias Gustafsson
• Tom Augspurger
• tomrod
• Tong Shen
• Tong SHEN
• TrigonaMinima
• tzinckgraf
37.3 pandas 0.19.2

Release date: 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.

Highlights include:
- Compatibility with Python 3.6
- Added a Pandas Cheat Sheet. (GH13202).

See the v0.19.2 Whatsnew page for an overview of all bugs that have been fixed in 0.19.2.

37.3.1 Thanks

- Ajay Saxena
- Ben Kandel
- Chris
- Chris Ham
- Christopher C. Aycock
- Daniel Himmelstein
- Dave Willmer
- Dr-Irv
- gfyoung
- hesham shabana
- Jeff Carey
- Jeff Reback
- Joe Jevnik
• Joris Van den Bossche
• Julian Santander
• Kerby Shedden
• Keshav Ramaswamy
• Kevin Sheppard
• Luca Scarabello
• Matti Picus
• Matt Roeschke
• Maximilian Roos
• Mykola Golubyev
• Nate Yoder
• Nicholas Ver Halen
• Pawel Kordek
• Pietro Battiston
• Rodolfo Fernandez
• sinhkrks
• Tara Adiseshan
• Tom Augspurger
• wandersoncferreira
• Yaroslav Halchenko

37.4 pandas 0.19.1

Release date: 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.

See the v0.19.1 Whatsnew page for an overview of all bugs that have been fixed in 0.19.1.

37.4.1 Thanks

• Adam Chainz
• Anthonios Partheniou
• Arash Rouhani
• Ben Kandel
• Brandon M. Burroughs
• Chris
• chris-b1
Release date: 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.

See the v0.19.0 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.19.0.

37.5.1 Thanks

• adneu
• Adrien Emery
• agraboso
• Alex Alekseyev
• Alex Vig
• Allen Riddell
• Amol
• Amol Agrawal
• Andy R. Terrel
• Anthonios Partheniou
• babakkeyvani
• Ben Kandel
• Bob Baxley
• Brett Rosen
• c123w
• Camilo Cota
• Chris
• chris-b1
• Chris Grinolds
• Christian Hudon
• Christopher C. Aycock
• Chris Warth
• cmazzullo
• conquistador1492
• cr3
• Daniel Siladjii
• Douglas McNeil
• Drewrey Lupton
• dsm054
• Eduardo Blancas Reyes
• Elliot Marsden
• Evan Wright
• Felix Marczinowski
• Francis T. O’Donovan
• Gábor Lipták
• Geraint Duck
• gyoung
• Giacomo Ferroni
• Grant Roch
• Haleemur Ali
• harshul1610
• Hassan Shamim
• iamsimha
• Iulius Curt
• Ivan Nazarov
• jackieleng
• Jeff Reback
• Jeffrey Gerard
• Jenn Olsen
• Jim Crist
• Joe Jevnik
• John Evans
• John Freeman
• John Liekezer
• Johnny Gill
• John W. O’Brien
• John Zwinck
• Jordan Erenrich
• Joris Van den Bossche
• Josh Howes
• Jozef Brandys
• Kamil Sindi
• Ka Wo Chen
• Kerby Shedden
• Kerne
• Kevin Sheppard
• Matthieu Brucher
• Maximilian Roos
• Michael Scherer
• Mike Graham
• Mortada Mehyar
• mpuels
• Muhammad Haseeb Tariq
• Nate George
• Neil Parley
• Nicolas Bonnotte
• OXPHOS
• Pan Deng / Zora
• Paul
• Pauli Virtanen
• Paul Mestemaker
• Pawel Kordek
• Pietro Battiston
• pijucha
• Piotr Jucha
• priyankjain
• Ravi Kumar Nimmi
• Robert Gieseke
• Robert Kern
• Roger Thomas
• Roy Keyes
• Russell Smith
• Sahil Dua
• Sanjiv Lobo
• Sašo Stanovnik
• Shawn Heide
• sinhrks
• Sinhrks
• Stephen Kappel
• Steve Choi
• Stewart Henderson
• Sudarshan Konge
• Thomas A Caswell
• Tom Augspurger
• Tom Bird
• Uwe Hoffmann
• wcwagner
• WillAyd
• Xiang Zhang
• Yadunandan
• Yaroslav Halchenko
• YG-Riku
• Yuichiro Kaneko
• yui-knk
• zhangjinjie
• znmean
• Yan Facai

37.6 pandas 0.18.1

Release date: (May 3, 2016)
This is a minor release from 0.18.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.

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).

See the [v0.18.1 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.18.1.

37.6.1 Thanks

• 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
• Peter Waller
• Pietro Battistoni
• Prabhjot Singh
• Robin Wilson
• Roger Thomas
• Sebastian Bank
• Stephen Hoover
• Tim Hopper
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.

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 `.extractall()` method, and API changes to the `.extract()` method and `.cat()` method.
- `pd.test()` top-level nose test runner is available (GH4327).

See the v0.18.0 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.18.0.
37.7.1 Thanks

- ARF
- Alex Alekseyev
- Andrew McPherson
- Andrew Rosenfeld
- 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
pandas: powerful Python data analysis toolkit, Release 0.21.0

• 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
• dgram0
• gfyoung
• hack-c
• hcontrast
• jfoo
• kaustuv deolal
• lllllll
• ranarag
• rockg
• scls19fr
• seales
• sinhrks
• srib
• surveymedia.ca
• tworec

37.8  pandas 0.17.1

Release date:  (November 21, 2015)
This is a minor release from 0.17.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.
Highlights include:

• Support for Conditional HTML Formatting, see here
• Releasing the GIL on the csv reader & other ops, see here
• Regression in DataFrame.drop_duplicates from 0.16.2, causing incorrect results on integer values (GH11376)

See the v0.17.1 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.17.1.

37.8.1 Thanks

• 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
• Thierry Moisan
• Tom Augspurger
• Tux1
• Varun
• Wieland Hoffmann
• Winterflower
• Yoav Ram
• Younggun Kim
• Zeke
• ajcr
• azuranski
• behzad nouri
• cel4
• emilydolson
• hironow
• lexical
• llllllllllll
• rockg
• silentquasar
• sinhhrs
• taeold

37.9 pandas 0.17.0

Release date: (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.

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 (GH8316)
- 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)

See the v0.17.0 Whatsnew overview for an extensive list of all enhancements and bugs that have been fixed in 0.17.0.

### 37.9.1 Thanks

- Alex Rothberg
- Andrea Bedini
- Andrew Rosenfeld
- Andy Li
- Anthonios Partheniou
- Artemy Kolchinsky
- Bernard Willers
- Charlie Clark
- Chris
- Chris Whelan
- Christoph Gohlke
- Christopher Whelan
• Clark Fitzgerald
• Clearfield Christopher
• Dan Ringwalt
• 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 Carruccioni
• Stephan Hoyer
• Stephen Hoover
• Stephen Pascoe
• Terry Santegoeds
• Thomas Grainger
This is a minor release from 0.16.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.

Highlights include:

- A new `pipe` method, see here
- Documentation on how to use `numba` with `pandas`, see here

See the `v0.16.2` `Whatsnew` overview for an extensive list of all enhancements and bugs that have been fixed in 0.16.2.
37.10.1 Thanks

- 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
- Thomas Grainger
- Tom Ajamian
- Tom Augspurger
- Yoshiki Vázquez Baeza
- Younggun Kim
- austinc
- behzad nouri
- jreback
- lexical
- rekcahpassyla
- scls19fr
- sinhhrs
37.11 pandas 0.16.1

Release date: (May 11, 2015)

This is a minor release from 0.16.0 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.

See the v0.16.1 Whatsnew overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.16.1.

37.11.1 Thanks

- Alfonso MHC
- Andy Hayden
- Artemy Kolchinsky
- Chris Gilmer
- Chris Grinolds
- Dan Birken
- David BROCHART
- David Hirschfeld
- David Stephens
- Dr. Leo
- Evan Wright
- Frans van Dunné
- 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
37.12 pandas 0.16.0

**Release date:** (March 22, 2015)

This is a major release from 0.15.2 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

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
• 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

See the v0.16.0 Whatsnew overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.16.0.

37.12.1 Thanks

• Aaron Toth
• Alan Du
• Alessandro Amici
• Artemy 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
- Kodi Arfer
- Matti Airas
- Min RK
- Mortada Mehyar
- Robert
- Scott E Lasley
- Scott Lasley
- Sergio Pascual
- Skipper Seabold
- Stephan Hoyer
- Thomas Grainger
- Tom Augspurger
- TomAugspurger
- Vladimir Filimonov
- Vyomkesh Tripathi
- Will Holmgren
- Yulong Yang
- behzad nouri
- bertrandhaut
- bjonen
- cel4
- clham
- hsperr
- ischwabacher
- jnmclarty
- josham
- jreback
- omtinez
- roch
- sinhrks
- unutbu
37.13 pandas 0.15.2

Release date: (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.

See the v0.15.2 Whatsnew overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.2.

37.13.1 Thanks

- 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 Suggit
- Matthew Brett
- Phillip Cloud
- Rupert Thompson
- Scott E Lasley
- Stephan Hoyer
- Stephen Simmons
- Sylvain Corlay
- Thomas Grainger
- Tiago Antao
- Trent Hauck
This is a minor 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.

See the v0.15.1 Whatsnew overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.1.

37.14 Thanks

• Aaron Staple
• Andrew Rosenfeld
• Anton I. Sipos
• Artemy Kolchinsky
• Bill Letson
• Dave Hughes
• David Stephens
• Guillaume Horel
• Jeff Reback
• Joris Van den Bossche
• Kevin Sheppard
• Nick Stahl
• Sanghee Kim
• Stephan Hoyer
• TomAugspurger
• WANG Aiyong
• behzad nouri
• immerrr
• jnmclarty
• jreback
• pallav-fdsi
• unutbu

37.15 pandas 0.15.0

Release date: (October 18, 2014)
This is a major release from 0.14.1 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

• Drop support for numpy < 1.7.0 (GH7711)
• 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
• New DataFrame default display for df.info() to include memory usage, see Memory Usage
• New datetimelike properties accessor .dt for Series, see Datetimelike Properties
• Split indexing documentation into Indexing and Selecting Data and MultiIndex / Advanced Indexing
• Split out string methods documentation into Working with Text Data
• read_csv will now by default ignore blank lines when parsing, see here
• API change in using Indexes in set operations, 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)

See the v0.15.0 Whatsnew overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.0.
37.15.1 Thanks

- Aaron Schumacher
- Adam Greenhall
- Andy Hayden
- Anthony O’Brien
- Artemy Kolchinsky
- behzad nouri
- Benedikt Sauer
- benjamin
- Benjamin Thyreau
- Ben Schiller
- bjonen
- BorisVerk
- Chris Reynolds
- Chris Stoafer
- Dav Clark
- dlovell
- DSM
- dsm054
- FragLegs
- German Gomez-Herrero
- Hsiaoming Yang
- Huan Li
- hunterowens
- Hyungtae Kim
- immerrr
- Isaac Slavitt
- ischwabacher
- Jacob Schaer
- Jacob Wasserman
- Jan Schulz
- Jeff Tratner
- Jesse Farnham
- jmorris0x0
- jmclarty
- Joe Bradish
• Joerg Rittinger
• John W. O’Brien
• Joris Van den Bossche
• jreback
• Kevin Sheppard
• klonuo
• Kyle Meyer
• lexual
• Max Chang
• mcjcode
• Michael Mueller
• Michael W Schatzow
• Mike Kelly
• Mortada Mehyar
• mtrbean
• Nathan Sanders
• Nathan Typanski
• onesandzeroes
• Paul Masurel
• Phillip Cloud
• Pietro Battiston
• RenzoBertocchi
• rockg
• Ross Petchler
• seth-p
• Shahul Hameed
• Shashank Agarwal
• sinhrks
• someben
• stahlous
• stas-sl
• Stephan Hoyer
• thatneat
• tom-alcorn
• TomAugspurger
• Tom Augspurger
37.16 pandas 0.14.1

**Release date:** (July 11, 2014)

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.

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.

See the v0.14.1 `Whatsnew` overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.14.1.

37.16.1 Thanks

- Andrew Rosenfeld
- Andy Hayden
- Benjamin Adams
- Benjamin M. Gross
- Brian Quistorff
- Brian Wignall
- bwignall
- clham
- Daniel Waebert
- David Bew
- David Stephens
- DSM
• dsm054
• helger
• immerrr
• Jacob Schaer
• jaimefrio
• Jan Schulz
• John David Reaver
• John W. O’Brien
• Joris Van den Bossche
• jreback
• Julien Danjou
• Kevin Sheppard
• K.-Michael Aye
• Kyle Meyer
• lexual
• Matthew Brett
• Matt Wittmann
• Michael Mueller
• Mortada Mehyar
• onesandzeroes
• Phillip Cloud
• Rob Levy
• rockg
• sanguineturtle
• Schaer, Jacob C
• seth-p
• sinhrks
• Stephan Hoyer
• Thomas Kluyver
• Todd Jennings
• TomAugspurger
• unknown
• yelite
37.17 pandas 0.14.0

Release date: (May 31, 2014)

This is a major release from 0.13.1 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

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 multi-indexed 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.

See the v0.14.0 Whatsnew overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.14.0.

37.17.1 Thanks

- Acanthostega
- Adam Marcus
- agijsberts
- akittredge
- Alex Gaudio
- Alex Rothberg
- AllenDowney
- Andrew Rosenfeld
- Andy Hayden
- ankostis
- anomrake
- Antoine Mazières
- anton-d
- bashtage
- Benedikt Sauer
- benjamin
- Brad Buran
- bwignall
• cgohlke
• chebee7i
• Christopher Whelan
• Clark Fitzgerald
• clham
• Dale Jung
• Dan Allan
• Dan Birken
• danielballan
• Daniel Waeger
• David Jung
• David Stephens
• Douglas McNeil
• DSM
• Garrett Drapala
• Gouthaman Balaraman
• Guillaume Poulin
• hshimizu77
• hugo
• immerr
• ischwabacher
• Jacob Howard
• Jacob Schaer
• jaimefrio
• Jason Sexauer
• Jeff Reback
• Jeffrey Starr
• Jeff Tratner
• John David Reaver
• John McNamara
• John W. O’Brien
• Jonathan Chambers
• Joris Van den Bossche
• jreback
• jsexauer
• Julia Evans
• Júlio
• Katie Atkinson
• kdiether
• Kelsey Jordahl
• Kevin Sheppard
• K.-Michael Aye
• Matthias Kuhn
• Matt Wittmann
• Max Grender-Jones
• Michael E. Gruen
• michaelws
• mikebailey
• Mike Kelly
• Nipun Batra
• Noah Spies
• ojdo
• onesandzeroes
• Patrick O’Keefe
• phaebz
• Phillip Cloud
• Pietro Battiston
• PKEuS
• Randy Carnevale
• ribonouos
• Robert Gibboni
• rockg
• sinhrks
• Skipper Seabold
• SplashDance
• Stephan Hoyer
• Tim Cera
• Tobias Brandt
• Todd Jennings
• TomAugspurger
• Tom Augspurger
• unutbu
37.18 pandas 0.13.1

Release date: (February 3, 2014)

37.18.1 New Features

- Added `date_format` and `datetime_format` attribute to `ExcelWriter`. (GH4133)

37.18.2 API Changes

- `Series.sort` will raise a `ValueError` (rather than a `TypeError`) on sorting an object that is a view of another (GH5856, GH5853)
- `Raise/Warn SettingWithCopyError` (according to the option `chained_assignment` in more cases, when detecting chained assignment, related (GH5938, GH6025)
- `DataFrame.head(0)` returns self instead of empty frame (GH5846)
- `autocorrelation_plot` now accepts `**kwargs`. (GH5623)
- `convert_objects` now accepts a `convert_timedeltas='coerce'` argument to allow forced dtype conversion of timedeltas (GH5458; issue: 5689)
- Add `-NaN` and `-nan` to the default set of NA values (GH5952). See NA Values.
- `NDFrame` now has an `equals` method. (GH5283)
- `DataFrame.apply` will use the `reduce` argument to determine whether a `Series` or a `DataFrame` should be returned when the `DataFrame` is empty (GH6007).

37.18.3 Experimental Features

37.18.4 Improvements to existing features

- perf improvements in Series datetime/timedelta binary operations (GH5801)
- `option_context` context manager now available as top-level API (GH5752)
- `df.info()` view now display dtype info per column (GH5682)
- `df.info()` now honors option `max_info_rows`, disable null counts for large frames (GH5974)
- perf improvements in 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)
- support `dtypes` property on `Series/Panel/Panel4D`
• **extend `Panel.apply`** to allow arbitrary functions (rather than only ufuncs) (GH1148) allow multiple axes to be used to operate on slabs of a `Panel`.

• **The `ArrayFormatter` for datetime and timedelta64** now intelligently limit precision based on the values in the array (GH3401)

• `pd.show_versions()` is now available for convenience when reporting issues.

• **perf improvements to Series.str.extract** (GH5944)

• **perf improvements in dtypes/ftypes methods** (GH5968)

• **perf improvements in indexing with object dtypes** (GH5968)

• improved dtype inference for timedelta like passed to constructors (GH5458, GH5689)

• escape special characters when writing to latex (:issue: 5374)

• **perf improvements in `DataFrame.apply`** (GH6013)

• `pd.read_csv` and `pd.to_datetime` learned a new `infer_datetime_format` keyword which greatly improves parsing perf in many cases. Thanks to @lexual for suggesting and @danbirken for rapidly implementing. (GH5490, :issue: 6021)

• add ability to recognize ‘%p’ format code (am/pm) to date parsers when the specific format is supplied (GH5361)

• **Fix performance regression in JSON IO** (GH5765)

• performance regression in Index construction from Series (GH6150)

### 37.18.5 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)
• Bug in plotting `scatter_matrix` function. Wrong alignment among diagonal and off-diagonal plots, see (GH5497).
• Regression in Series with a multi-index via `ix` (GH6018)
• Bug in `Series.xs` with a multi-index (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).
pandas: powerful Python data analysis toolkit, Release 0.21.0

- Bug in HDFStore on appending a dataframe with multi-indexed columns to an existing table (GH6167)
- Consistency with dtypes in setting an empty DataFrame (GH6171)
- Bug in selecting on a multi-index 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)

37.19 pandas 0.13.0

Release date: January 3, 2014

37.19.1 New Features

- `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)
- Added `isin` method to DataFrame (GH4211)
- `df.to_clipboard()` learned a new `excel` keyword that let’s you paste df data directly into excel (enabled by default). (GH5070).
- Clipboard functionality now works with PySide (GH4282)
- New `extract` string method returns regex matches more conveniently (GH4685)
- Auto-detect field widths in read_fwf when unspecified (GH4488)
- `to_csv()` now outputs datetime objects according to a specified format string via the `date_format` keyword (GH4313)
- Added `LastWeekOfMonth` DateOffset (GH4637)
- Added `cumcount` groupby method (GH4646)
- Added FY5253, and FY5253Quarter DateOffsets (GH4511)
- Added `mode()` method to Series and DataFrame to get the statistical mode(s) of a column/series. (GH5367)

37.19.2 Experimental Features

- The new `eval()` function implements expression evaluation using `numexpr` behind the scenes. This results in large speedups for complicated expressions involving large DataFrames/Series.
- `DataFrame` has a new `eval()` that evaluates an expression in the context of the DataFrame; allows inline expression assignment
- A `query()` method has been added that allows you to select elements of a DataFrame using a natural query syntax nearly identical to Python syntax.
- `pd.eval` and friends now evaluate operations involving `datetime64` objects in Python space because `numexpr` cannot handle NaT values (GH4897).
• Add msgpack support via `pd.read_msgpack()` and `pd.to_msgpack()` / `df.to_msgpack()` for serialization of arbitrary pandas (and python objects) in a lightweight portable binary format (GH686, GH5506)
• Added PySide support for the qtpandas DataFrameModel and DataFrameWidget.
• Added `pandas.io.gbq` for reading from (and writing to) Google BigQuery into a DataFrame. (GH4140)

37.19.3 Improvements to existing features

• `read_html` now raises a `URLError` instead of catching and raising a `ValueError` (GH4303, GH4305)
• `read_excel` now supports an integer in its `sheetname` argument giving the index of the sheet to read in (GH4301).
• `get_dummies` works with NaN (GH4446)
• Added a test for `read_clipboard()` and `to_clipboard()` (GH4282)
• Added bins argument to `value_counts` (GH3945), also sort and ascending, now available in Series method as well as top-level function.
• Text parser now treats anything that reads like inf (“inf”, “Inf”, “-Inf”, “iNf”, etc.) to infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
• Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)
• Significant table writing performance improvements in HDFStore
• JSON date serialization now performed in low-level C code.
• JSON support for encoding datetime.time
• Expanded JSON docs, more info about orient options and the use of the numpy param when decoding.
• Add `drop_level` argument to `xs` (GH4180)
• Can now resample a DataFrame with ohlc (GH2320)
• `Index.copy()` and `MultiIndex.copy()` now accept keyword arguments to change attributes (i.e., `names`, `levels`, `labels`) (GH4039)
• Add `rename` and `set_names` methods to `Index` as well as `set_names`, `set_levels`, `set_labels` to `MultiIndex`. (GH4039) with improved validation for all (GH4039, GH4794)
• A Series of `dtype timedelta64[ns]` can now be divided/multiplied by an integer series (GH4521)
• A Series of `dtype timedelta64[ns]` can now be divided by another `timedelta64[ns]` object to yield a `float64` dtyped Series. This is frequency conversion; astyping is also supported.
• `Timedelta64` support `fillna/ffill/bfill` with an integer interpreted as seconds, or a `timedelta` (GH3371)
• Box numeric ops on `timedelta Series` (GH4984)
• Datetime64 support `ffill/bfill`
• Performance improvements with `__getitem__` on `DataFrames` with when the key is a column
• Support for using a `DatetimeIndex/PeriodsIndex` directly in a datelike calculation e.g. `s-s.index` (GH4629)
• Better/cleaned up exceptions in core/common, io/excel and core/format (GH4721, GH3954), as well as cleaned up test cases in tests/test_frame, tests/test_multilevel (GH4732).
• Performance improvement of timeseries plotting with PeriodIndex and added test to vbench (GH4705 and GH4722)

• Add `axis` and `level` keywords to `where`, so that the `other` argument can now be an alignable pandas object.

• `to_datetime` with a format of ‘%Y%m%d’ now parses much faster

• It’s now easier to hook new Excel writers into pandas (just subclass `ExcelWriter` and register your engine). You can specify an engine in `to_excel` or in `ExcelWriter`. You can also specify which writers you want to use by default with config options `io.excel.xlsx.writer` and `io.excel.xls.writer`. (GH4745, GH4750)

• `Panel.to_excel()` now accepts keyword arguments that will be passed to its `DataFrame.to_excel()` methods. (GH4750)

• Added XlsxWriter as an optional `ExcelWriter` engine. This is about 5x faster than the default openpyxl xlsx writer and is equivalent in speed to the xlwt xlsx writer module. (GH4542)

• allow `DataFrame` constructor to accept more list-like objects, e.g. list of `collections.Sequence` and `array.Array` objects (GH3783, GH4297, GH4851), thanks @lgautier

• `DataFrame` constructor now accepts a numpy masked record array (GH3478), thanks @jnothman

• `_getitem__ with tuple key (e.g., [:, 2]) on Series without MultiIndex raises ValueError` (GH4759, GH4837)

• `read_json` now raises a (more informative) `ValueError` when the dict contains a bad key and `orient='split'` (GH4730, GH4838)

• `read_stata` now accepts Stata 13 format (GH4291)

• `ExcelWriter` and `ExcelFile` can be used as contextmanagers. (GH3441, GH4933)

• `pandas` is now tested with two different versions of `statsmodels` (0.4.3 and 0.5.0) (GH4981).

• Better string representations of `MultiIndex` (including ability to roundtrip via `repr`). (GH3347, GH4935)

• Both `ExcelFile` and `read_excel` to accept an `xlrd.Book` for the `io` (formerly `path_or_buf`) argument; this requires engine to be set. (GH4961).

• `concat` now gives a more informative error message when passed objects that cannot be concatenated (GH4608).

• Add `halflife` option to exponentially weighted moving functions (PR GH4998)

• `to_dict` now takes `records` as a possible outtype. Returns an array of column-keyed dictionaries. (GH4936)

• `tz_localize` can infer a fall daylight savings transition based on the structure of unlocalized data (GH4230)

• `DatetimeIndex` is now in the API documentation

• Improve support for converting R datasets to pandas objects (more informative index for timeseries and numeric, support for factors, dist, and high-dimensional arrays).

• `read_html()` now supports the `parse_dates`, `tupleize_cols` and `thousands` parameters (GH4770).

• `json_normalize()` is a new method to allow you to create a flat table from semi-structured JSON data. See the docs (GH1067)

• `DataFrame.from_records()` will now accept generators (GH4910)

• `DataFrame.interpolate()` and `Series.interpolate()` have been expanded to include interpolation methods from scipy. (GH4434, GH1892)
- Series now supports a `to_frame` method to convert it to a single-column DataFrame (GH5164)
- DatetimeIndex (and date_range) can now be constructed in a left- or right-open fashion using the `closed` parameter (GH4579)
- Python csv parser now supports `usecols` (GH4335)
- Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. (GH5271)
- `NDFrame.drop()` now accepts names as well as integers for the axis argument. (GH5354)
- Added short docstrings to a few methods that were missing them + fixed the docstrings for Panel flex methods. (GH5336)
- `NDFrame.drop()`, `NDFrame.dropna()`, and `.drop_duplicates()` all accept `inplace` as a keyword argument; however, this only means that the wrapper is updated inplace, a copy is still made internally. (GH1960, GH5247, GH5628, and related GH2325 [still not closed])
- Fixed bug in `tools.plotting.andrews_curvres` so that lines are drawn grouped by color as expected.
- `read_excel()` now tries to convert integral floats (like `1.0`) to int by default. (GH5394)
- Excel writers now have a default option `merge_cells` in `to_excel()` to merge cells in MultiIndex and Hierarchical Rows. Note: using this option it is no longer possible to round trip Excel files with merged MultiIndex and Hierarchical Rows. Set the `merge_cells` to `False` to restore the previous behaviour. (GH5254)
- The FRED DataReader now accepts multiple series (issue ‘3413’)
- StataWriter adjusts variable names to Stata’s limitations (GH5709)

### 37.19.4 API Changes

- `DataFrame.reindex()` and forward/backward filling now raises ValueError if either index is not monotonic (GH4483, GH4484).
- `pandas` now is Python 2/3 compatible without the need for `2to3` thanks to @jratner. 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)
- Deprecated `iterkv`, which will be removed in a future release (was just an alias of `iteritems` used to get around `2to3`'s changes). (GH4384, GH4375, GH4372)
- `Series.get` with negative indexers now returns the same as `[]` (GH4390)
- Allow `ix/loc` for Series/DataFrame/Panel to set on any axis even when the single-key is not currently contained in the index for that axis (GH2578, GH5226, GH5632, GH5720, GH5744, GH5756)
- Default export for `to_clipboard` is now csv with a sep of `t` for compat (GH3368)
- `at` now will enlarge the object inplace (and return the same) (GH2578)
- `DataFrame.plot` will scatter plot `x` versus `y` by passing `kind='scatter'` (GH2215)
- `HDFStore`
  - `append_to_multiple` automatically synchronizes writing rows to multiple tables and adds a `dropna` kwarg (GH4698)
- handle a passed Series in table format (GH4330)

- 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

- 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 model='w' with an OPEN file handle (GH4367)

- allow a passed locations array or mask as a where condition (GH4467)

- 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)

- the format keyword now replaces the table keyword; allowed values are fixed(f)|table(t)

- the Storer format has been renamed to Fixed

- a column multi-index will be recreated properly (GH4710); raise on trying to use a multi-index with data_columns on the same axis

- select_as_coordinates will now return an Int64Index of the resultant selection set

- support timedelta64[ns] as a serialization type (GH3577)

- store datetime.date objects as ordinals rather then timetuples to avoid timezone issues (GH2852), thanks @tavistorph and @numpand

- numexpr 2.2.2 fixes incompatibility in PyTables 2.4 (GH4908)

- flush now accepts an fsync parameter, which defaults to False (GH5364)

- unicode indices not supported on table formats (GH5386)

- pass thru store creation arguments; can be used to support in-memory stores

• JSON

- added date_unit parameter to specify resolution of timestamps. Options are seconds, milliseconds, microseconds and nanoseconds. (GH4362, GH4498).

- added default_handler parameter to allow a callable to be passed which will be responsible for handling otherwise unserializable objects. (GH5138)

• Index and MultiIndex changes (GH4039):

- Setting levels and labels directly on MultiIndex is now deprecated. Instead, you can use the set_levels() and set_labels() methods.

- levels, labels and names properties no longer return lists, but instead return containers that do not allow setting of items (‘mostly immutable’)

- levels, labels and names are validated upon setting and are either copied or shallow-copied.

- inplace setting of levels or labels now correctly invalidates the cached properties. (GH5238)

- __deepcopy__ now returns a shallow copy (currently: a view) of the data - allowing metadata changes.
- MultiIndex.astype() now only allows np.object-like dtypes and now returns a MultiIndex rather than an Index. (GH4039)

- Added is_ method to Index that allows fast equality comparison of views (similar to np.may_share_memory but no false positives, and changes on levels and labels setting on MultiIndex). (GH4859, GH4909)

- Aliased __iadd__ to __add__. (GH4996)

- Added is_ method to Index that allows fast equality comparison of views (similar to np.may_share_memory but no false positives, and changes on levels and labels setting on MultiIndex). (GH4859, GH4909)

- 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. Add .bool() method to NDFrame objects to facilitate evaluating of single-element boolean Series

- DataFrame.update() no longer raises a DataConflictError, it now will raise a ValueError instead (if necessary) (GH4732)

- Series.isin() and DataFrame.isin() now raise a TypeError when passed a string (GH4763). Pass a list of one element (containing the string) instead.

- Remove undocumented/unused kind keyword argument from read_excel, and ExcelFile. (GH4713, GH4712)

- The method argument of NDFrame.replace() is valid again, so that a a list can be passed to to_replace (GH4743).

- provide automatic dtype conversions on _reduce operations (GH3371)

- exclude non-numerics if mixed types with datelike in _reduce operations (GH3371)

- default for tupleize_cols is now False for both to_csv and read_csv. Fair warning in 0.12 (GH3604)

- moved timedeltas support to pandas.tseries.timedeltas.py; add timedeltas string parsing, add top-level to_timedelta function

- NDFrame now is compatible with Python’s toplevel abs() function (GH4821).

- raise a TypeError on invalid comparison ops on Series/DataFrame (e.g. integer/datetime) (GH4968)

- 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. 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 raise a TypeError, e.g. Series(range(5))[3.5:4.5] (GH263, issue:5375)

- Make Categorical repr nicer (GH4368)

- Remove deprecated Factor (GH3650)

- Remove deprecated set_printoptions/reset_printoptions (issue:3046)

- Remove deprecated _verbose_info (GH3215)

- Begin removing methods that don’t make sense on GroupBy objects (GH4887).

- Remove deprecated read_clipboard/to_clipboard/ExcelFile/ExcelWriter from pandas.io.parsers (GH3717)
• 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)

• Arithmetic func factories are now passed real names (suitable for using with super) (GH5240)

• Provide numpy compatibility with 1.7 for a calling convention like np.prod(pandas_object) as numpy call with additional keyword args (GH4435)

• Provide __dir__ method (and local context) for tab completion / remove ipython completers code (GH4501)

• Support non-unique axes in a Panel via indexing operations (GH4960)

• .truncate will raise a ValueError if invalid before and afters dates are given (GH5242)

• Timestamp now supports now/today/utcnow class methods (GH5339)

• default for display.max_seq_len is now 100 rather then None. This activates truncated display ("...") of long sequences in various places. (GH3391)

• All division with NDFrame - likes is now truedivision, regardless of the future import. You can use // and floordiv to do integer division.

```
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) # no future import required
Out[6]:
   0  0.200000
   1  0.666667
   2  1.500000
   3  4.000000
   dtype: float64
```

• raise/warn SettingWithCopyError/Warning exception/warning when setting of a copy thru chained assignment is detected, settable via option mode.chained_assignment

• test the list of NA values in the csv parser. add N/A, #NA as independent default na values (GH5521)

• The refactoring involving ‘Series‘ deriving from NDFrame breaks rpy2<=2.3.8. an Issue has been opened against rpy2 and a workaround is detailed in GH5698. Thanks @JanSchulz.

• 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)

### 37.19.5 Internal Refactoring

In 0.13.0 there is a major refactor primarily to subclass Series from NDFrame, which is the base class currently for DataFrame and Panel, to unify methods and behaviors. Series formerly subclassed directly from ndarray. (GH4080, GH3862, GH816) See Internal Refactoring

• 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
pandas: powerful Python data analysis toolkit, Release 0.21.0

- from_axes, _wrap_array, axes, ix, loc, iloc, shape, empty, swapaxes, transpose, pop
- __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)
- isnul/notnull now available on NDFrame objects

- 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 same API as original DataFrame filter
- fillna refactored to core/generic.py, while > 3ndim is Not Implemented

- Series now inherits from NDFrame rather than directly from ndarray. There are several minor changes that affect the API.
- numpy functions that do not support the array interface will now return ndarrays rather than series, e.g. np.diff, np.ones_like, np.where
- Series(0.5) would previously return the scalar 0.5, this is no longer supported
- 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 now have a _prop_attributes, which can be used to indicate various values to propagate to a new object from an existing (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 class, courtesy of @jtratner

• Bug in Series update where the parent frame is not updating its cache based on changes (GH4080, GH5216) 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`

• Series (for index) / Panel (for items) now as attribute access to its elements (GH1903)

• Refactor `clip` methods to core/generic.py (GH4798)

• Refactor of `_get_numeric_data/_get_bool_data` to core/generic.py, allowing Series/Panel functionality

• Refactor of Series arithmetic with time-like objects (datetime/timedelta/time etc.) into a separate, cleaned up wrapper class. (GH4613)

• Complex compat for Series with `ndarray`. (GH4819)

• Removed unnecessary `rwproperty` from codebase in favor of builtin property. (GH4843)

• Refactor object level numeric methods (mean/sum/min/max...) from object level modules to core/generic.py (GH4435).

• Refactor cum objects to core/generic.py (GH4435), note that these have a more numpy-like function signature.

• `read_html()` now uses `TextParser` to parse HTML data from bs4/lxml (GH4770).

• Removed the `keep_internal` keyword parameter in pandas/core/groupby.py because it wasn’t being used (GH5102).

• Base `DateOffsets` are no longer all instantiated on importing pandas, instead they are generated and cached on the fly. The internal representation and handling of `DateOffsets` has also been clarified. (GH5189, related GH5004)

• `MultiIndex` constructor now validates that passed levels and labels are compatible. (GH5213, GH5214)

• Unity `dropna` for Series/DataFrame signature (GH5250), tests from GH5234, courtesy of @rockg

• Rewrite `assert_almost_equal()` in cython for performance (GH4398)

• Added an internal `_update_inplace` method to facilitate updating `NDFrame` wrappers on inplace ops (only is for convenience of caller, doesn’t actually prevent copies). (GH5247)

### 37.19.6 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 multi-index 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 cparses 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 multi-index 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 date-times (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 laazy 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 multi-indexing 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 multi-index 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 @immerrr
• 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 multi-index 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 seg fault in C parser caused by passing more names than columns in the file. (GH5156)
• Fixed bug where `Series.isin` with date/time-like dtypes (GH5021)
• C and Python Parser can now handle the more common multi-index 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 multi-index 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 multi-index selection in PY3 when using certain keys (GH5725)
• Row-wise concat of differing dtypes failing in certain cases (GH5754)

37.20 pandas 0.12.0

Release date: 2013-07-24

37.20.1 New Features

• pd.read_html() can now parse HTML strings, files or urls and returns a list of DataFrame s courtesy of @cpcloud. (GH3477, GH3605, GH3606)
• Support for reading Amazon S3 files. (GH3504)
• Added module for reading and writing JSON strings/files: pandas.io.json includes to_json DataFrame/Series method, and a read_json top-level reader various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
• Added module for reading and writing Stata files: pandas.io stata (GH1512) includes to_stata DataFrame method, and a read_stata top-level reader
• Added support for writing in to_csv and reading in read_csv, multi-index columns. The header option in read_csv now accepts a list of the rows from which to read the index. Added the option, tupleize_cols to provide compatibility for the pre 0.12 behavior of writing and reading multi-index 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 multi-index column. Note: The default value will change in 0.12 to make the default to write and read multi-index columns in the new format. (GH3571, GH1651, GH3141)
• Add iterator to Series.str (GH3638)
• pd.set_option() now allows N option, value pairs (GH3667).
• Added keyword parameters for different types of scatter_matrix subplots
• A filter method on grouped Series or DataFrames returns a subset of the original (GH3680, GH919)
• Access to historical Google Finance data in pandas.io.data (GH3814)
• DataFrame plotting methods can sample column colors from a Matplotlib colormap via the colormap key- word. (GH3860)

37.20.2 Improvements to existing features

• Fixed various issues with internal pprinting code, the repr() for various objects including TimeStamp and Index now produces valid python code strings and can be used to recreate the object, (GH3038, GH3379, GH3251, GH3460)
• convert_objects now accepts a copy parameter (defaults to True)
• HDFStore
  – will retain index attributes (freq,tz,name) on recreation (GH3499,;issue:4098)
pandas: powerful Python data analysis toolkit, Release 0.21.0

- will warn with a `AttributeConflictWarning` 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)
- table writing performance improvements.
- support python3 (via `PyTables 3.0.0`) (GH3750)

- Add modulo operator to Series, DataFrame
- Add `date` method to `DatetimeIndex`
- Add `dropna` argument to `pivot_table` (:issue: 3820)
- Simplified the API and added a describe method to `Categorical`
- `melt` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned `DataFrame` (GH3649), thanks @hoechenberger. If `var_name` is not specified and `dataframe.columns.name` is not None, then this will be used as the `var_name` (GH4144). Also support for MultiIndex columns.
- clipboard functions use pyperclip (no dependencies on Windows, alternative dependencies offered for Linux) (GH3837).
- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have 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.
- Added Faq section on `repr` display options, to help users customize their setup.
- `where` operations that result in block splitting are much faster (GH3733)
- Series and DataFrame `hist` methods now take a `figsize` argument (GH3834)
- `DatetimeIndex`es no longer try to convert mixed-integer indexes during join operations (GH3877)
- Add `unit` keyword to `Timestamp` and `to_datetime` to enable passing of integers or floats that are in an epoch unit of D, s, ms, us, ns, thanks @mtkini (GH3969) (e.g. unix timestamps or epoch s, with fractional seconds allowed) (GH3540)
- `DataFrame corr` method (spearman) is now cythonized.
- 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)
- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters
- Added `layout` keyword to `DataFrame.hist()` for more customizable layout (GH4050)
- `Timestamp.min` and `Timestamp.max` now represent valid `Timestamp` instances instead of the default date-time.min and date-time.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and `BeautifulSoup==4.2.0` is detected (GH4214)

### 37.20.3 API Changes

- `HDFStore`  
  - When removing an object, `remove(key)` raises `KeyError` if the key is not a valid store object.
- raise a TypeError on passing where or columns to select with a Storer; these are invalid parameters at this time (GH4189)
- can now specify an encoding option to append/put to enable alternate encodings (GH3750)
- enable support for iterator/chunksize with read_hdf

- The repr() for (Multi)Index now obeys display.max_seq_items rather then numpy threshold print options. (GH3426, GH3466)
- Added mangle_dupe_cols option to read_table/csv, allowing users to control legacy behaviour re dupe cols (A, A.1, A.2 vs A, A) (GH3468) Note: The default value will change in 0.12 to the “no mangle” behaviour, If your code relies on this behaviour, explicitly specify mangle_dupe_cols=True in your calls.
- Do not allow astypes 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
- Do not allow datetimelike/timedeltalike creation except with valid types (e.g. cannot pass datetime64[ms]) (GH3432)
- Add squeeze keyword to groupby to allow reduction from DataFrame -> Series if groups are unique. Regression from 0.10.1, partial revert on (GH2893) with (GH3596)
- 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)
- The raise_on_error option to plotting methods is obviated by GH3572, so it is removed. Plots now always raise when data cannot be plotted or the object being plotted has a dtype of object.
- 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)
- Deprecated display.height, display.width is now only a formatting option does not control triggering of summary, similar to < 0.11.0.
- 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)
- io API changes
  - added pandas.io.api for i/o imports
  - removed Excel support to pandas.io.excel
  - added top-level pd.read_sql and to_sql DataFrame methods
  - removed clipboard support to pandas.io.clipboard
  - replace top-level and instance methods save and load with top-level read_pickle and to_pickle instance method, save and load will give deprecation warning.
- the method and axis arguments of DataFrame.replace() are deprecated
- set FutureWarning to require data_source, and to replace year/month with expiry date in pandas.io options. This is in preparation to add options data from Google (GH3822)
• the method and axis arguments of DataFrame.replace() are deprecated
• Implement __nonzero__ for NDFrame objects (GH3691, GH3696)
• as_matrix with mixed signed and unsigned dtypes will result in 2 x the lcd of the unsigned as an int, maxing with int64, to avoid precision issues (GH3733)
• na_values in a list provided to read_csv/read_excel will match string and numeric versions e.g. na_values=["99"] will match 99 whether the column ends up being int, float, or string (GH3611)
• 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
• more consistency in the to_datetime return types (give string/array of string inputs) (GH3888)
• The internal pandas class hierarchy has changed (slightly). The previous PandasObject now is called PandasContainer and a new PandasObject has become the baseclass 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)

### 37.20.4 Experimental Features

• Added experimental CustomBusinessDay class to support DateOffsets with custom holiday calendars and custom weekmasks. (GH2301)

### 37.20.5 Bug Fixes

• Fixed an esoteric excel reading bug, xlrd>= 0.9.0 now required for excel support. Should provide python3 support (for reading) which has been lacking. (GH3164)
• Disallow Series constructor called with MultiIndex which caused segfault (GH4187)
• Allow unioning of date ranges sharing a timezone (GH3491)
• Fix to_csv issue when having a large number of rows and NaT in some columns (GH3437)
• .loc was not raising when passed an integer list (GH3449)
• Unordered time series selection was misbehaving when using label slicing (GH3448)
• Fix sorting in a frame with a list of columns which contains datetime64[ns] dtypes (GH3461)
• DataFrames fetched via FRED now handle ‘.’ as a NaN. (GH3469)
• Fix regression in a DataFrame apply with axis=1, objects were not being converted back to base dtypes correctly (GH3480)
• Fix issue when storing uint dtypes in an HDFStore. (GH3493)
• Non-unique index support clarified (GH3468)
  – Addressed handling of dupe columns in df.to_csv new and old (GH3454, GH3457)
  – 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)
- 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.itertuples()` 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)

- Fixed bug in groupby with empty series referencing a variable before assignment. (GH3510)
- Allow index name to be used in groupby for non MultiIndex (GH4014)
- Fixed bug in mixed-frame assignment with aligned series (GH3492)
- Fixed bug in selecting month/quarter/year from a series would not select the time element on the last day (GH3546)
- Fixed a couple of MultiIndex rendering bugs in `df.to_html()` (GH3547, GH3553)
- Properly convert np.datetime64 objects in a Series (GH3416)
- Raise a `TypeError` on invalid datetime/timedelta operations e.g. add datetimes, multiple timedelta x datetime
- Fix `.diff` on datelike and timedelta operations (GH3100)
- `combine_first` not returning the same dtype in cases where it can (GH3552)
- Fixed bug with `Panel.transpose` argument aliases (GH3556)
- Fixed platform bug in `PeriodIndex.take` (GH3579)
- Fixed bug in incorrect conversion of datetime64[ns] in `combine_first` (GH3593)
- Fixed bug in reset_index with `NaN` in a multi-index (GH3586)
- `fillna` methods now raise a `TypeError` when the `value` parameter is a `list` or `tuple`.
- Fixed bug where a time-series was being selected in preference to an actual column name in a frame (GH3594)
- Make `secondary_y` work properly for bar plots (GH3598)
- Fix modulo and integer division on Series,DataFrames to act similary to `float` dtypes to return `np.nan` or `np.inf` as appropriate (GH3590)
- Fix incorrect dtype on groupby with `as_index=False` (GH3610)
- Fix `read_csv/read_excel` to correctly encode identical `na_values`, e.g. `na_values=[-999.0,-999]` was failing (GH3611)
- Disable HTML output in qtconsole again. (GH3657)
• Reworked the new repr display logic, which users found confusing. (GH3663)
• Fix indexing issue in ndim >= 3 with iloc (GH3617)
• Correctly parse date columns with embedded (nan/NaN) into datetime64[ns] dtype in read_csv when parse_dates is specified (GH3062)
• Fix not consolidating before to_csv (GH3624)
• Fix alignment issue when setitem in a DataFrame with a piece of a DataFrame (GH3626) or a mixed DataFrame and a Series (GH3668)
• Fix plotting of unordered DatetimeIndex (GH3601)
• sql.write_frame failing when writing a single column to sqlite (GH3628), thanks to @stonebig
• Fix pivoting with nan in the index (GH3558)
• Fix running of bs4 tests when it is not installed (GH3605)
• Fix parsing of html table (GH3606)
• read_html() now only allows a single backend: html5lib (GH3616)
• convert_objects with convert_dates='coerce' was parsing some single-letter strings into today's date
• DataFrame.from_records did not accept empty recarrays (GH3682)
• DataFrame.to_csv will succeed with the deprecated option nanRep, @tdsmith
• DataFrame.to_html and DataFrame.to_latex now accept a path for their first argument (GH3702)
• Fix file tokenization error with r delimiter and quoted fields (GH3453)
• Groupby transform with item-by-item not upcasting correctly (GH3740)
• Incorrectly read a HDFStore multi-index Frame with a column specification (GH3748)
• read_html now correctly skips tests (GH3741)
• PandasObjects raise TypeError when trying to hash (GH3882)
• Fix incorrect arguments passed to concat that are not list-like (e.g. concat(df1,df2)) (GH3481)
• Correctly parse when passed the dtype=str (or other variable-len string dtypes) in read_csv (GH3795)
• Fix index name not propagating when using loc/ix (GH3880)
• Fix groupby when applying a custom function resulting in a returned DataFrame was not converting dtypes (GH3911)
• Fixed a bug where DataFrame.replace with a compiled regular expression in the to_replace argument wasn’t working (GH3907)
• Fixed __truediv__ in Python 2.7 with numexpr installed to actually do true division when dividing two integer arrays with at least 10000 cells total (GH3764)
• Indexing with a string with seconds resolution not selecting from a time index (GH3925)
• csv parsers would loop infinitely if iterator=True but no chunksize was specified (GH3967), python parser failing with chunksize=1
• Fix index name not propagating when using shift
• Fixed dropna=False being ignored with multi-index stack (GH3997)
• Fixed flattening of columns when renaming MultiIndex columns DataFrame (GH4004)
• Fix `Series.clip` for datetime series. NA/NaN threshold values will now throw ValueError (GH3996)
• Fixed insertion issue into DataFrame, after rename (GH4032)
• 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)
• Fix bug where HDFStore will fail to append because of a different block ordering on-disk (GH4096)
• Better error messages on inserting incompatible columns to a frame (GH4107)
• Fixed bug in `DataFrame.replace` where a nested dict wasn’t being iterated over when regex=False (GH4115)
• Fixed bug in `convert_objects(convert_numeric=True)` where a mixed numeric and object Series/Frame was not converting properly (GH4119)
• Fixed bugs in multi-index selection with column multi-index and duplicates (GH4145, GH4146)
• 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 (GH3990)
• Fixed bug in `Series.where` where broadcasting a single element input vector to the length of the series resulted in multiplying the value inside the input (GH4192)
• 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)

37.21 pandas 0.11.0

Release date: 2013-04-22

37.21.1 New Features

• New documentation section, 10 Minutes to Pandas
• New documentation section, Cookbook
• Allow mixed dtypes (e.g float32/float64/int32/int16/int8) to coexist in DataFrames and propagate in operations
• Add function to pandas.io.data for retrieving stock index components from Yahoo! finance (GH2795)
• Support slicing with time objects (GH2681)
• Added .iloc attribute, to support strict integer based indexing, analogous to .ix (GH2922)
• Added .loc attribute, to support strict label based indexing, analogous to .ix (GH3053)
• Added .iat attribute, to support fast scalar access via integers (replaces iget_value/iset_value)
• Added .at attribute, to support fast scalar access via labels (replaces get_value/set_value)
• Moved functionality from irow,icol,iget_value/iset_value to .iloc indexer (via _ixs methods in each object)
• Added support for expression evaluation using the numexpr library
• Added convert=boolean to take routines to translate negative indices to positive, defaults to True
• Added to_series() method to indices, to facilitate the creation of indexers (GH3275)

37.21.2 Improvements to existing features

• Improved performance of df.to_csv() by up to 10x in some cases. (GH3059)
• added blocks attribute to DataFrames, to return a dict of dtypes to homogeneously dtyped DataFrames
• added keyword convert_numeric to convert_objects() to try to convert object dtypes to numeric types (default is False)
• convert_dates in convert_objects can now be coerce which will return a datetime64[ns] dtype with non-convertibles set as NaT; will preserve an all-nan object (e.g. strings), default is True (to perform soft-conversion
• Series print output now includes the dtype by default
• Optimize internal reindexing routines (GH2819, GH2867)
• describe_option() now reports the default and current value of options.
• Add format option to pandas.to_datetime with faster conversion of strings that can be parsed with datetime.strptime
• Add axes property to Series for compatibility
• Add xs function to Series for compatibility
• Allow setitem in a frame where only mixed numerics are present (e.g. int and float), (GH3037)
• HDFStore
  – 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)
  – support read_hdf/to_hdf API similar to read Csv/to_csv (GH3222)
• Add squeeze method to possibly remove length 1 dimensions from an object.

```
In [1]: 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'])
```
In [2]: p
Out[2]:
<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

In [3]: p.reindex(items=['ItemA']).squeeze()

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-02</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2001-01-03</td>
<td>1.212112</td>
<td>-0.173215 0.119209</td>
<td>-1.044236</td>
<td></td>
</tr>
<tr>
<td>2001-01-04</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>2001-01-05</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
</tbody>
</table>

In [4]: p.reindex(items=['ItemA'], minor=['B']).squeeze()

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-02</td>
<td>-0.282863</td>
<td></td>
</tr>
<tr>
<td>2001-01-03</td>
<td>-0.173215</td>
<td></td>
</tr>
<tr>
<td>2001-01-04</td>
<td>-2.104569</td>
<td></td>
</tr>
<tr>
<td>2001-01-05</td>
<td>-0.706771</td>
<td></td>
</tr>
</tbody>
</table>
Freq: D, Name: B, dtype: float64

• Improvement to Yahoo API access in pd.io.data.Options (GH2758)
• 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.
• Allow selection semantics via a string with a datelike index to work in both Series and DataFrames (GH3070)
• added option `display.mpl_style` providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).

• Improved performance across several core functions by taking memory ordering of arrays into account. Courtesy of @stephenwlin (GH3130)

• Improved performance of groupby transform method (GH2121)

• Handle “ragged” CSV files missing trailing delimiters in rows with missing fields when also providing explicit list of column names (so the parser knows how many columns to expect in the result) (GH2981)

• On a mixed DataFrame, allow setting with indexers with ndarray/DataFrame on rhs (GH3216)

• Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)

• Add `time` method to DatetimeIndex (GH3180)

• Return NA when using Series.str[...] for values that are not long enough (GH3223)

• Display cursor coordinate information in time-series plots (GH1670)

• `to_html()` now accepts an optional “escape” argument to control reserved HTML character escaping (enabled by default) and escapes &, in addition to < and >. (GH2919)

### 37.21.3 API Changes

• Do not automatically upcast numeric specified dtypes to `int64` or `float64` (GH622 and GH797)

• DataFrame construction of lists and scalars, with no dtype present, will result in casting to `int64` or `float64`, regardless of platform. This is not an apparent change in the API, but noting it.

• Guarantee that `convert_objects()` for Series/DataFrame always returns a copy

• groupby operations will respect dtypes for numeric float operations (float32/float64); other types will be operated on, and will try to cast back to the input dtype (e.g. if an int is passed, as long as the output doesn’t have nans, then an int will be returned)

• backfill/pad/take/diff/ohlc will now support `float32/int16/int8` operations

• Block types will upcast as needed in where/masking operations (GH2793)

• Series now automatically will try to set the correct dtype based on passed datetimelike objects (datetime/Timestamp)
  – timedelta64 are returned in appropriate cases (e.g. Series - Series, when both are datetime64)
  – mixed datetimes and objects (GH2751) in a constructor will be cast correctly
  – astype on datetimes to object are now handled (as well as NaT conversions to np.nan)
  – all timedelta like objects will be correctly assigned to `timedelta64` with mixed NaN and/or NaT allowed

• arguments to DataFrame.clip were inconsistent to numpy and Series clipping (GH2747)

• util.testing.assert_frame_equal now checks the column and index names (GH2964)

• Constructors will now return a more informative ValueError on failures when invalid shapes are passed

• Don’t suppress TypeError in GroupBy.agg (GH3238)

• Methods return None when inplace=True (GH1893)
• **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 will now automatically create `data_columns` for passed keys.

• Downcast on pivot if possible (GH3283), adds argument `downcast` to `fillna`.

• Introduced options `display.height/width` for explicitly specifying terminal height/width in characters. Deprecated `display.line_width`, now replaced by `display.width`. These defaults are in effect for scripts as well, so unless disabled, previously very wide output will now be output as “expand_repr” style wrapped output.

• Various defaults for options (including `display.max_rows`) have been revised, after a brief survey concluded they were wrong for everyone. Now at w=80, h=60.

• HTML repr output in IPython qtconsole is once again controlled by the option `display.notebook_repr_html`, and on by default.

### 37.21.4 Bug Fixes

• Fix seg fault on empty data frame when `fillna` with `pad` or `backfill` (GH2778).

• Single element ndarrays of datetimelike objects are handled (e.g. `np.array(datetime(2001,1,1,0,0)))`, w/o dtype being passed.

• 0-dim ndarrays with a passed dtype are handled correctly (e.g. `np.array(0., dtype='float32')`)

• Fix some boolean indexing inconsistencies in `Series.__getitem__/__setitem__` (GH2776).

• Fix issues with `DataFrame` and `Series` constructor with integers that overflow `int64` and some mixed typed type lists (GH2845).

• **HDFStore**
  - Fix weird PyTables error when using too many selectors in a where also correctly filter on any number of values in a Term expression (so not using numexpr filtering, but isn filtering).
  - Internally, change all variables to be private-like (now have leading underscore).
  - Fixes for query parsing to correctly interpret boolean and `!=` (GH2849, GH2973).
  - Fixes for pathological case on SparseSeries with 0-len array and compression (GH2931).
  - Fixes bug with writing rows if part of a block was all-nan (GH3012).
  - Exceptions are now `ValueError` or `TypeError` as needed.
  - A table will now raise if `min_itemsize` contains fields which are not queryables.

• Bug showing up in applymap where some object type columns are converted (GH2909) had an incorrect default in `convert_objects`.

• **TimeDeltas**
  - Series ops with a Timestamp on the rhs was throwing an exception (GH2898) added tests for Series ops with datetimes, timedeltas, Timestamps, and datelike Series on both lhs and rhs.
  - Fixed subtle timedelta64 inference issue on py3 & numpy 1.7.0 (GH3094).
  - Fixed some formatting issues on timedelta when negative.
  - Support null checking on timedelta64, representing (and formatting) with NaT.
  - Support `setitem` with `np.nan` value, converts to NaT.
- Support min/max ops in a Dataframe (abs not working, nor do we error on non-supported ops)
- Support idxmin/idxmax/abs/max/min in a Series (GH2989, GH2982)

- Bug on in-place putmasking on an integer series that needs to be converted to float (GH2746)
- Bug in argsort of datetime64[ns] Series with NaT (GH2967)
- Bug in value_counts of datetime64[ns] Series (GH3002)
- Fixed printing of NaT in an index
- Bug in idxmin/idxmax of datetime64[ns] Series with NaT (GH2982)
- Bug in icol, take with negative indicies was producing incorrect return values (see GH2922, GH2892), also check for out-of-bounds indices (GH3029)
- Bug in DataFrame column insertion when the column creation fails, existing frame is left in an irrecoverable state (GH3010)
- Bug in DataFrame update, combine_first where non-specified values could cause dtype changes (GH3016, GH3041)
- Bug in groupby with first/last where dtypes could change (GH3041, GH2763)
- Formatting of an index that has nan was inconsistent or wrong (would fill from other values), (GH2850)
- Unstack of a frame with no nans would always cause dtype upcasting (GH2929)
- Fix scalar datetime.datetime parsing bug in read_csv (GH3071)
- Fixed slow printing of large Dataframes, due to inefficient dtype reporting (GH2807)
- Fixed a segfault when using a function as grouper in groupby (GH3035)
- Fix pretty-printing of infinite data structures (closes GH2978)
- Fixed exception when plotting timeseries bearing a timezone (closes GH2877)
- str.contains ignored na argument (GH2806)
- Substitute warning for segfault when grouping with categorical grouper of mismatched length (GH3011)
- Fix exception in SparseSeries.density (GH2083)
- Fix upsampling bug with closed='left' and daily to daily data (GH3020)
- Fixed missing tick bars on scatter_matrix plot (GH3063)
- Fixed bug in Timestamp(d,tz=foo) when d is date() rather then datetime() (GH2993)
- series.plot(kind='bar') now respects pylab color schem (GH3115)
- Fixed bug in reshape if not passed correct input, now raises TypeError (GH2719)
- Fixed a bug where Series ctor did not respect ordering if OrderedDict passed in (GH3282)
- Fix NameError issue on RESO_US (GH2787)
- Allow selection in an unordered timeseries to work similiary to an ordered timeseries (GH2437).
- Fix implemented .xs when called with axes=1 and a level parameter (GH2903)
- Timestamp now supports the class method fromordinal similar to datetimes (GH3042)
- Fix issue with indexing a series with a boolean key and specifying a 1-len list on the rhs (GH2745) or a list on the rhs (GH3235)
- Fixed bug in groupby apply when kernel generate list of arrays having unequal len (GH1738)
• fixed handling of rolling_corr with center=True which could produce corr>1 (GH3155)
• Fixed issues where indices can be passed as ‘index/column’ in addition to 0/1 for the axis parameter
• PeriodIndex.tolist now boxes to Period (GH3178)
• PeriodIndex.get_loc KeyError now reports Period instead of ordinal (GH3179)
• df.to_records bug when handling MultiIndex (GH3189)
• Fix Series.__getitem__ segfault when index less than -length (GH3168)
• Fix bug when using Timestamp as a date parser (GH2932)
• Fix bug creating date range from Timestamp with time zone and passing same time zone (GH2926)
• Add comparison operators to Period object (GH2781)
• Fix bug when concatenating two Series into a DataFrame when they have the same name (GH2797)
• Fix automatic color cycling when plotting consecutive timeseries without color arguments (GH2816)
• fixed bug in the pickling of PeriodIndex (GH2891)
• Upcast/split blocks when needed in a mixed DataFrame when setitem with an indexer (GH3216)
• Invoking df.applymap on a dataframe with dupe cols now raises a ValueError (GH2786)
• Apply with invalid returned indices raise correct Exception (GH2808)
• Fixed a bug in plotting log-scale bar plots (GH3247)
• df.plot() grid on/off now obeys the mpl default style, just like series.plot(). (GH3233)
• Fixed a bug in the legend of plotting.andrews_curves() (GH3278)
• Produce a series on apply if we only generate a singular series and have a simple index (GH2893)
• Fix Python ASCII file parsing when integer falls outside of floating point spacing (GH3258)
• fixed pretty printing of sets (GH3294)
• Panel() and Panel.from_dict() now respects ordering when give OrderedDict (GH3303)
• DataFrame where with a datetimelike incorrectly selecting (GH3311)
• Ensure index casts work even in Int64Index
• Fix set_index segfault when passing MultiIndex (GH3308)
• Ensure pickles created in py2 can be read in py3
• Insert ellipsis in MultiIndex summary repr (GH3348)
• Groupby will handle mutation among an input groups columns (and fallback to non-fast apply) (GH3380)
• Eliminated unicode errors on FreeBSD when using MPL GTK backend (GH3360)
• Period.strftime should return unicode strings always (GH3363)
• Respect passed read_* chunksize in get_chunk function (GH3406)

37.22 pandas 0.10.1

Release date: 2013-01-22
37.22.1 New Features

- Add data interface to World Bank WDI pandas.io.wb (GH2592)

37.22.2 API Changes

- Restored inplace=True behavior returning self (same object) with deprecation warning until 0.11 (GH1893)
- HDFStore
  - refactored HDFStore to deal with non-table stores as objects, will allow future enhancements
  - removed keyword compression from put (replaced by keyword complib to be consistent across library)
  - warn PerformanceWarning if you are attempting to store types that will be pickled by PyTables

37.22.3 Improvements to existing features

- HDFStore
  - enables storing of multi-index dataframes (closes GH1277)
  - support data column indexing and selection, via data_columns keyword in append
  - support write chunking to reduce memory footprint, via chunksize keyword to append
  - support automagic indexing via index keyword to append
  - support expectedrows keyword in append to inform PyTables about the expected tablesize
  - support start and stop keywords in select to limit the row selection space
  - added get_store context manager to automatically import with pandas
  - added column filtering via columns keyword in select
  - added methods append_to_multiple/select_as_multiple/select_as_coordinates to do multiple-table append/selection
  - added support for datetime64 in columns
  - added method unique to select the unique values in an indexable or data column
  - added method copy to copy an existing store (and possibly upgrade)
  - show the shape of the data on disk for non-table stores when printing the store
  - added ability to read PyTables flavor tables (allows compatibility to other HDF5 systems)

- Add logx option to DataFrame/Series.plot (GH2327, GH2565)
- Support reading gzipped data from file-like object
- pivot_table aggfunc can be anything used in GroupBy.aggregate (GH2643)
- Implement DataFrame merges in case where set cardinalities might overflow 64-bit integer (GH2690)
- Raise exception in C file parser if integer dtype specified and have NA values. (GH2631)
- Attempt to parse ISO8601 format dates when parse_dates=True in read_csv for major performance boost in such cases (GH2698)
- Add methods neg and inv to Series
• Implement kind option in ExcelFile to indicate whether it’s an XLS or XLSX file (GH2613)
• Documented a fast-path in pd.read_csv when parsing iso8601 datetime strings yielding as much as a 20x speedup. (GH5993)

37.22.4 Bug Fixes

• Fix read_csv/read_table multithreading issues (GH2608)
• HDFStore
  – correctly handle nan elements in string columns; serialize via the nan_rep keyword to append
  – raise correctly on non-implemented column types (unicode/date)
  – handle correctly Term passed types (e.g. index<1000, when index is Int64), (closes GH512)
  – handle Timestamp correctly in data_columns (closes GH2637)
  – contains correctly matches on non-natural names
  – correctly store float32 dtypes in tables (if not other float types in the same table)
• Fix DataFrame.info bug with UTF8-encoded columns. (GH2576)
• Fix DatetimeIndex handling of FixedOffset tz (GH2604)
• More robust detection of being in IPython session for wide DataFrame console formatting (GH2585)
• Fix platform issues with file:/// in unit test (GH2564)
• Fix bug and possible segfault when grouping by hierarchical level that contains NA values (GH2616)
• Ensure that MultiIndex tuples can be constructed with NAs (GH2616)
• Fix int64 overflow issue when unstacking MultiIndex with many levels (GH2616)
• Exclude non-numeric data from DataFrame.quantile by default (GH2625)
• Fix a Cython C int64 boxing issue causing read_csv to return incorrect results (GH2599)
• Fix groupby summing performance issue on boolean data (GH2692)
• Don’t bork Series containing datetime64 values with to_datetime (GH2699)
• Fix DataFrame.from_records corner case when passed columns, index column, but empty record list (GH2633)
• Fix C parser-tokenizer bug with trailing fields. (GH2668)
• Don’t exclude non-numeric data from GroupBy.max/min (GH2700)
• Don’t lose time zone when calling DatetimeIndex.drop (GH2621)
• Fix setitem on a Series with a boolean key and a non-scalar as value (GH2686)
• Box datetime64 values in Series.apply/map (GH2627, GH2689)
• Upconvert datetime + datetime64 values when concatenating frames (GH2624)
• Raise a more helpful error message in merge operations when one DataFrame has duplicate columns (GH2649)
• Fix partial date parsing issue occuring only when code is run at EOM (GH2618)
• Prevent MemoryError when using counting sort in sortlevel with high-cardinality MultiIndex objects (GH2684)
• Fix Period resampling bug when all values fall into a single bin (GH2070)
• Fix buggy interaction with usecols argument in read_csv when there is an implicit first index column (GH2654)
• Fix bug in `Index.summary()` where string format methods were being called incorrectly. (GH3869)

37.23 pandas 0.10.0

Release date: 2012-12-17

37.23.1 New Features

• Brand new high-performance delimited file parsing engine written in C and Cython. 50% or better performance in many standard use cases with a fraction as much memory usage. (GH407, GH821)
• Many new file parser (read_csv, read_table) features:
  – Support for on-the-fly gzip or bz2 decompression (`compression` option)
  – Ability to get back numpy.recarray instead of DataFrame (`as_recarray=True`)
  – `dtype` option: explicit column dtypes
  – `usecols` option: specify list of columns to be read from a file. Good for reading very wide files with many irrelevant columns (GH1216 GH926, GH2465)
  – Enhanced unicode decoding support via `encoding` option
  – `skipinitialspace` dialect option
  – Can specify strings to be recognized as True (`true_values`) or False (`false_values`)
  – High-performance `delim_whitespace` option for whitespace-delimited files; a preferred alternative to the ‘s+’ regular expression delimiter
  – Option to skip “bad” lines (wrong number of fields) that would otherwise have caused an error in the past (`error_bad_lines` and `warn_bad_lines` options)
  – Substantially improved performance in the parsing of integers with thousands markers and lines with comments
  – Easy of European (and other) decimal formats (`decimal` option) (GH584, GH2466)
  – Custom line terminators (e.g. `lineterminator='~'`) (GH2457)
  – Handling of no trailing commas in CSV files (GH2333)
  – Ability to handle fractional seconds in date_converters (GH2209)
  – `read_csv` allow scalar arg to `na_values` (GH1944)
  – Explicit column dtype specification in `read_*` functions (GH1858)
  – Easier CSV dialect specification (GH1743)
  – Improve parser performance when handling special characters (GH1204)
• Google Analytics API integration with easy oauth2 workflow (GH2283)
• Add error handling to `Series.str.encode/decode` (GH2276)
• Add `where` and `mask` to Series (GH2337)
• Grouped histogram via `by` keyword in `Series/DataFrame.hist` (GH2186)
• Support optional `min_periods` keyword in `corr` and `cov` for both Series and DataFrame (GH2002)
• Add `duplicated` and `drop_duplicates` functions to Series (GH1923)
• Add docs for HDFStore table format
• ‘density’ property in SparseSeries (GH2384)
• Add ffill and bfill convenience functions for forward- and backfilling time series data (GH2284)
• New option configuration system and functions set_option, get_option, describe_option, and reset_option. Deprecate set_printoptions and reset_printoptions (GH2393). You can also access options as attributes via pandas.options.X
• Wide DataFrames can be viewed more easily in the console with new expand_frame_repr and line_width configuration options. This is on by default now (GH2436)
• Scikits.timeseries-like moving window functions via rolling_window (GH1270)

37.23.2 Experimental Features

• Add support for Panel4D, a named 4 Dimensional structure
• Add support for ndpanel factory functions, to create custom, domain-specific N-Dimensional containers

37.23.3 API Changes

• The default binning/labeling behavior for resample has been changed to closed='left', label='left' for daily and lower frequencies. This had been a large source of confusion for users. See “what’s new” page for more on this. (GH2410)
• Methods with inplace option now return None instead of the calling (modified) object (GH1893)
• The special case DataFrame - TimeSeries doing column-by-column broadcasting has been deprecated. Users should explicitly do e.g. df.sub(ts, axis=0) instead. This is a legacy hack and can lead to subtle bugs.
• inf/-inf are no longer considered as NA by isnull/notnull. To be clear, this is legacy cruft from early pandas. This behavior can be globally re-enabled using the new option mode.use_inf_as_null (GH2050, GH1919)
• pandas.merge will now default to sort=False. For many use cases sorting the join keys is not necessary, and doing it by default is wasteful
• Specify header=0 explicitly to replace existing column names in file in read_* functions.
• Default column names for header-less parsed files (yielded by read_csv, etc.) are now the integers 0, 1, .... A new argument prefix has been added; to get the v0.9.x behavior specify prefix='X' (GH2034). This API change was made to make the default column names more consistent with the DataFrame constructor’s default column names when none are specified.
• DataFrame selection using a boolean frame now preserves input shape
• If function passed to Series.apply yields a Series, result will be a DataFrame (GH2316)
• Values like YES/NO/yes/no will not be considered as boolean by default any longer in the file parsers. This can be customized using the new true_values and false_values options (GH2360)
• obj.fillna() is no longer valid; make method='pad' no longer the default option, to be more explicit about what kind of filling to perform. Add ffill/bfill convenience functions per above (GH2284)
• HDFStore.keys() now returns an absolute path-name for each key
• to_string() now always returns a unicode string. (GH224)
• File parsers will not handle NA sentinel values arising from passed converter functions
37.23.4 Improvements to existing features

- Add `nrows` option to DataFrame.from_records for iterators (GH1794)
- Unstack/reshape algorithm rewrite to avoid high memory use in cases where the number of observed key-tuples is much smaller than the total possible number that could occur (GH2278). Also improves performance in most cases.
- Support duplicate columns in DataFrame.from_records (GH2179)
- Add `normalize` option to Series/DataFrame.asfreq (GH2137)
- SparseSeries and SparseDataFrame construction from empty and scalar values now no longer create dense ndarrays unnecessarily (GH2322)
- HDFStore now supports hierarchical keys (GH2397)
- Support multiple query selection formats for HDFStore tables (GH1996)
- Support `del store['df']` syntax to delete HDFStores
- Add multi-dtype support for HDFStore tables
- `min_itemsize` parameter can be specified in HDFStore table creation
- Indexing support in HDFStore tables (GH698)
- Add `line_terminator` option to DataFrame.to_csv (GH2383)
- added implementation of `str(x)/unicode(x)/bytes(x)` to major pandas data structures, which should do the right thing on both py2.x and py3.x. (GH2224)
- Reduce groupby:apply overhead substantially by low-level manipulation of internal NumPy arrays in DataFrames (GH535)
- Implement `value_vars` in melt and add melt to pandas namespace (GH2412)
- Added boolean comparison operators to Panel
- Enable Series.str.strip/strip methods to take an argument (GH2411)
- The DataFrame ctor now respects column ordering when given an OrderedDict (GH2455)
- Assigning DatetimeIndex to Series changes the class to TimeSeries (GH2139)
- Improve performance of .value_counts method on non-integer data (GH2480)
- `get_level_values` method for MultiIndex return Index instead of ndarray (GH2449)
- `convert_to_r_dataframe` conversion for datetime values (GH2351)
- Allow DataFrame.to_csv to represent inf and nan differently (GH2026)
- Add `min_i` argument to nancorr to specify minimum required observations (GH2002)
- Add `inplace` option to sortlevel / sort functions on DataFrame (GH1873)
- Enable DataFrame to accept scalar constructor values like Series (GH1856)
- DataFrame.from_records now takes optional `size` parameter (GH1794)
- include iris dataset (GH1709)
- No datetime64 DataFrame column conversion of datetime.datetime with tzinfo (GH1581)
- Micro-optimizations in DataFrame for tracking state of internal consolidation (GH217)
- Format parameter in DataFrame.to_csv (GH1525)
pandas: powerful Python data analysis toolkit, Release 0.21.0

- Partial string slicing for DatetimeIndex for daily and higher frequencies (GH2306)
- Implement col_space parameter in to_html and to_string in DataFrame (GH1000)
- Override Series.tolist and box datetime64 types (GH2447)
- Optimize unstack memory usage by compressing indices (GH2278)
- Fix HTML repr in IPython qtconsole if opening window is small (GH2275)
- Escape more special characters in console output (GH2492)
- df.select now invokes bool on the result of crit(x) (GH2487)

37.23.5 Bug Fixes

- Fix major performance regression in DataFrame.iteritems (GH2273)
- Fixes bug when negative period passed to Series/DataFrame.diff (GH2266)
- Escape tabs in console output to avoid alignment issues (GH2038)
- Properly box datetime64 values when retrieving cross-section from mixed-dtype DataFrame (GH2272)
- Fix concatenation bug leading to GH2057, GH2257
- Fix regression in Index console formatting (GH2319)
- Box Period data when assigning PeriodIndex to frame column (GH2243, GH2281)
- Raise exception on calling reset_index on Series with inplace=True (GH2277)
- Enable setting multiple columns in DataFrame with hierarchical columns (GH2295)
- Respect dtype=object in DataFrame constructor (GH2291)
- Fix DatetimeIndex.join bug with tz-aware indexes and how='outer' (GH2317)
- pop(...) and del works with DataFrame with duplicate columns (GH2349)
- Treat empty strings as NA in date parsing (rather than let dateutil do something weird) (GH2263)
- Prevent uint64 -> int64 overflows (GH2355)
- Enable joins between MultiIndex and regular Index (GH2024)
- Fix time zone metadata issue when unioning non-overlapping DatetimeIndex objects (GH2367)
- Raise/handle int64 overflows in parsers (GH2247)
- Deleting of consecutive rows in HDFStore tables is much faster than before
- Appending on a HDFStore would fail if the table was not first created via put
- Use col_space argument as minimum column width in DataFrame.to_html (GH2328)
- Fix tz-aware DatetimeIndex.to_period (GH2232)
- Fix DataFrame row indexing case with MultiIndex (GH2314)
- Fix to_excel exporting issues with Timestamp objects in index (GH2294)
- Fixes assigning scalars and array to hierarchical column chunk (GH1803)
- Fixed a UnicodeDecodeError with series tidy_repr (GH2225)
- Fixed issued with duplicate keys in an index (GH2347, GH2380)
- Fixed issues re: Hash randomization, default on starting w/ py3.3 (GH2331)
- Fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- Fix Timestamp formatting with tzoffset time zone in dateutil 2.1 (GH2443)
- Fix GroupBy.apply issue when using BinGrouper to do ts binning (GH2300)
- Fix issues resulting from datetime.datetime columns being converted to datetime64 when calling DataFrame.apply. (GH2374)
- Raise exception when calling to_panel on non uniquely-indexed frame (GH2441)
- Improved detection of console encoding on IPython zmq frontends (GH2458)
- Preserve time zone when append-ing two time series (GH2260)
- Box timestamps when calling reset_index on time-zone-aware index rather than creating a tz-less datetime64 column (GH2262)
- Enable searching non-string columns in DataFrame.filter(like=...) (GH2467)
- Fixed issue with losing nanosecond precision upon conversion to DatetimeIndex (GH2252)
- Handle timezones in Datetime.normalize (GH2338)
- Fix test case where dtype specification with endianness causes failures on big endian machines (GH2318)
- Fix plotting bug where upsampling causes data to appear shifted in time (GH2448)
- Fix read_csv failure for UTF-16 with BOM and skiprows(GH2298)
- read_csv with names arg not implicitly setting header=None (GH2459)
- Unrecognized compression mode causes segfault in read_csv (GH2474)
- In read_csv, header=0 and passed names should discard first row (GH2269)
- Correctly route to stdout/stderr in read_table (GH2071)
- Fix exception when Timestamp.to_datetime is called on a Timestamp with tzoffset (GH2471)
- Fixed unintentional conversion of datetime64 to long in groupby.first() (GH2133)
- Union of empty DataFrames now return empty with concatenated index (GH2307)
- DataFrame.sort_index raises more helpful exception if sorting by column with duplicates (GH2488)
- DataFrame.to_string formatters can be list, too (GH2520)
- DataFrame.combine_first will always result in the union of the index and columns, even if one DataFrame is length-zero (GH2525)
- Fix several DataFrame.iloc/irow with duplicate indices issues (GH2228, GH2259)
- Use Series names for column names when using concat with axis=1 (GH2489)
- Raise Exception if start, end, periods all passed to date_range (GH2538)
- Fix Panel resampling issue (GH2537)

37.24 pandas 0.9.1

Release date: 2012-11-14
37.24.1 New Features

- Can specify multiple sort orders in DataFrame/Series.sort/sort_index (GH928)
- New `top` and `bottom` options for handling NAs in rank (GH1508, GH2159)
- Add `where` and `mask` functions to DataFrame (GH2109, GH2151)
- Add `at_time` and `between_time` functions to DataFrame (GH2149)
- Add flexible `pow` and `rpow` methods to DataFrame (GH2190)

37.24.2 API Changes

- Upsampling period index “spans” intervals. Example: annual periods upsampled to monthly will span all months in each year
- `Period.end_time` will yield timestamp at last nanosecond in the interval (GH2124, GH2125, GH1764)
- File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

37.24.3 Improvements to existing features

- Time rule inference for week-of-month (e.g. WOM-2FRI) rules (GH2140)
- Improve performance of datetime + business day offset with large number of offset periods
- Improve HTML display of DataFrame objects with hierarchical columns
- Enable referencing of Excel columns by their column names (GH1936)
- DataFrame.dot can accept ndarrays (GH2042)
- Support negative periods in Panel.shift (GH2164)
- Make `.drop(...)` work with non-unique indexes (GH2101)
- Improve performance of Series/DataFrame.diff (re: GH2087)
- Support unary ~ (__invert__) in DataFrame (GH2110)
- Turn off pandas-style tick locators and formatters (GH2205)
- DataFrame[DataFrame] uses DataFrame.where to compute masked frame (GH2230)

37.24.4 Bug Fixes

- Fix some duplicate-column DataFrame constructor issues (GH2079)
- Fix bar plot color cycle issues (GH2082)
- Fix off-center grid for stacked bar plots (GH2157)
- Fix plotting bug if inferred frequency is offset with N > 1 (GH2126)
- Implement comparisons on date offsets with fixed delta (GH2078)
- Handle inf/-inf correctly in read_* parser functions (GH2041)
- Fix matplotlib unicode interaction bug
- Make WLS r-squared match statsmodels 0.5.0 fixed value
- Fix zero-trimming DataFrame formatting bug
- Correctly compute/box datetime64 min/max values from Series.min/max (GH2083)
- Fix unstacking edge case with unrepresented groups (GH2100)
- Fix Series.str failures when using pipe pattern ‘|’ (GH2119)
- Fix pretty-printing of dict entries in Series, DataFrame (GH2144)
- Cast other datetime64 values to nanoseconds in DataFrame ctor (GH2095)
- Alias Timestamp.astimezone to tz_convert, so will yield Timestamp (GH2060)
- Fix timedelta64 formatting from Series (GH2165, GH2146)
- Handle None values gracefully in dict passed to Panel constructor (GH2075)
- Box datetime64 values as Timestamp objects in Series/DataFrame.iget (GH2148)
- Fix Timestamp indexing bug in DatetimeIndex.insert (GH2155)
- Use index name(s) (if any) in DataFrame.to_records (GH2161)
- Don’t lose index names in Panel.to_frame/DataFrame.to_panel (GH2163)
- Work around length-0 boolean indexing NumPy bug (GH2096)
- Fix partial integer indexing bug in DataFrame.xs (GH2107)
- Fix variety of cut/qcut string-bin formatting bugs (GH1978, GH1979)
- Raise Exception when xs view not possible of MultiIndex’d DataFrame (GH2117)
- Fix groupby(...) .first() issue with datetime64 (GH2133)
- Better floating point error robustness in some rolling_ * functions (GH2114, GH2527)
- Fix ewma NA handling in the middle of Series (GH2128)
- Fix numerical precision issues in diff with integer data (GH2087)
- Fix bug in MultiIndex.__getitem__ with NA values (GH2008)
- Fix DataFrame.from_records dict-arg bug when passing columns (GH2179)
- Fix Series and DataFrame.diff for integer dtypes (GH2087, GH2174)
- Fix bug when taking intersection of DatetimeIndex with empty index (GH2129)
- Pass through timezone information when calling DataFrame.align (GH2127)
- Properly sort when joining on datetime64 values (GH2196)
- Fix indexing bug in which False/True were being coerced to 0/1 (GH2199)
- Many unicode formatting fixes (GH2201)
- Fix improper MultiIndex conversion issue when assigning e.g. DataFrame.index (GH2200)
- Fix conversion of mixed-type DataFrame to ndarray with dup columns (GH2236)
- Fix duplicate columns issue (GH2218, GH2219)
- Fix SparseSeries.__pow__ issue with NA input (GH2220)
- Fix icol with integer sequence failure (GH2228)
- Fixed resampling tz-aware time series issue (GH2245)
- SparseDataFrame.icol was not returning SparseSeries (GH2227, GH2229)
• Enable ExcelWriter to handle PeriodIndex (GH2240)
• Fix issue constructing DataFrame from empty Series with name (GH2234)
• Use console-width detection in interactive sessions only (GH1610)
• Fix parallel_coordinates legend bug with mpl 1.2.0 (GH2237)
• Make tz_localize work in corner case of empty Series (GH2248)

37.25 pandas 0.9.0

Release date: 10/7/2012

37.25.1 New Features

• Add `str.encode` and `str.decode` to Series (GH1706)
• Add `to_latex` method to DataFrame (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)
• Recognize and convert more boolean values in file parsing (Yes, No, TRUE, FALSE, variants thereof) (GH1691, GH1295)
• Add Panel.update method, analogous to DataFrame.update (GH1999, GH1988)

37.25.2 Improvements to existing features

• Proper handling of NA values in merge operations (GH1990)
• Add flags option for `re.compile` in some Series.str methods (GH1659)
• Parsing of UTC date strings in read_* functions (GH1693)
• Handle generator input to Series (GH1679)
• Add `na_action='ignore'` to Series.map to quietly propagate NAs (GH1661)
• Add args/kwds options to Series.apply (GH1829)
• Add inplace option to Series/DataFrame.reset_index (GH1797)
• Add `level` parameter to Series.reset_index
• Add quoting option for DataFrame.to_csv (GH1902)
• Indicate long column value truncation in DataFrame output with ... (GH1854)
• DataFrame.dot will not do data alignment, and also work with Series (GH1915)
• Add `na` option for missing data handling in some vectorized string methods (GH1689)
• If index_label=False in DataFrame.to_csv, do not print fields/commas in the text output. Results in easier importing into R (GH1583)
• Can pass tuple/list of axes to DataFrame.dropna to simplify repeated calls (dropping both columns and rows) (GH924)
• Improve DataFrame.to_html output for hierarchically-indexed rows (do not repeat levels) (GH1929)
• TimeSeries.between_time can now select times across midnight (GH1871)
• Enable skip_footer parameter in ExcelFile.parse (GH1843)

37.25.3 API Changes

• Change default header names in read_* functions to more Pythonic X0, X1, etc. instead of X.1, X.2. (GH2000)
• Deprecated day_of_year API removed from PeriodIndex, use dayofyear (GH1723)
• Don’t modify NumPy suppress printoption 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)
• Enable skipfooter parameter in text parsers as an alias for skip_footer

37.25.4 Bug Fixes

• Perform arithmetic column-by-column in mixed-type DataFrame to avoid type upcasting issues. Caused downstream DataFrame.diff bug (GH1896)
• Fix matplotlib auto-color assignment when no custom spectrum passed. Also respect passed color keyword argument (GH1711)
• Fix resampling logical error with closed=’left’ (GH1726)
• Fix critical DatetimeIndex.union bugs (GH1730, GH1719, GH1745, GH1702, GH1753)
• Fix critical DatetimeIndex.intersection bug with unanchored offsets (GH1708)
• Fix MM-YYYY time series indexing case (GH1672)
• Fix case where Categorical group key was not being passed into index in GroupBy result (GH1701)
• Handle Ellipsis in Series.__getitem__/__setitem__ (GH1721)
• Fix some bugs with handling datetime64 scalars of other units in NumPy 1.6 and 1.7 (GH1717)
• Fix performance issue in MultiIndex.format (GH1746)
• Fixed GroupBy bugs interacting with DatetimeIndex asof / map methods (GH1677)
• Handle factors with NAs in pandas.rpy (GH1615)
• Fix statsmodels import in pandas.stats.var (GH1734)
• Fix DataFrame repr/info summary with non-unique columns (GH1700)
• Fix Series.iget_value for non-unique indexes (GH1694)
- Don’t lose tzinfo when passing DatetimeIndex as DataFrame column (GH1682)
- Fix tz conversion with time zones that haven’t had any DST transitions since first date in the array (GH1673)
- Fix field access with UTC->local conversion on unsorted arrays (GH1756)
- Fix isnull handling of array-like (list) inputs (GH1755)
- Fix regression in handling of Series in Series constructor (GH1671)
- Fix comparison of Int64Index with DatetimeIndex (GH1681)
- Fix min_periods handling in new rolling_max/min at array start (GH1695)
- Fix errors with how=’median’ and generic NumPy resampling in some cases caused by SeriesBinGrouper (GH1648, GH1688)
- When grouping by level, exclude unobserved levels (GH1697)
- Don’t lose tzinfo in DatetimeIndex when shifting by different offset (GH1683)
- Hack to support storing data with a zero-length axis in HDFStore (GH1707)
- Fix DatetimeIndex tz-aware range generation issue (GH1674)
- Fix method=’time’ interpolation with intraday data (GH1698)
- Don’t plot all-NA DataFrame columns as zeros (GH1696)
- Fix bug in scatter_plot with by option (GH1716)
- Fix performance problem in infer_freq with lots of non-unique stamps (GH1686)
- Fix handling of PeriodIndex as argument to create MultiIndex (GH1705)
- Fix re: unicode MultiIndex level names in Series/DataFrame repr (GH1736)
- Handle PeriodIndex in to_datetime instance method (GH1703)
- Support StaticTzInfo in DatetimeIndex infrastructure (GH1692)
- Allow MultiIndex setops with length-0 other type indexes (GH1727)
- Fix handling of DatetimeIndex in DataFrame.to_records (GH1720)
- Fix handling of general objects in isnull on which bool(...) fails (GH1749)
- Fix .ix indexing with MultiIndex ambiguity (GH1678)
- Fix .ix setting logic error with non-unique MultiIndex (GH1750)
- Basic indexing now works on MultiIndex with > 1000000 elements, regression from earlier version of pandas (GH1757)
- Handle non-float64 dtypes in fast DataFrame.corr/cov code paths (GH1761)
- Fix DatetimeIndex.isin to function properly (GH1763)
- Fix conversion of array of tz-aware datetime.datetime to DatetimeIndex with right time zone (GH1777)
- Fix DST issues with generating anchored date ranges (GH1778)
- Fix issue calling sort on result of Series.unique (GH1807)
- Fix numerical issue leading to square root of negative number in rolling_std (GH1840)
- Let Series.str.split accept no arguments (like str.split) (GH1859)
- Allow user to have dateutil 2.1 installed on a Python 2 system (GH1851)
- Catch ImportError less aggressively in pandas/__init__.py (GH1845)
- Fix pip source installation bug when installing from GitHub (GH1805)
- Fix error when window size > array size in rolling_apply (GH1850)
- Fix pip source installation issues via SSH from GitHub
- Fix OLS.summary when column is a tuple (GH1837)
- Fix bug in __doc__ patching when -OO passed to interpreter (GH1792 GH1741 GH1774)
- Fix unicode console encoding issue in IPython notebook (GH1782, GH1768)
- Fix unicode formatting issue with Series.name (GH1782)
- Fix bug in DataFrame.duplicated with datetime64 columns (GH1833)
- Fix bug in Panel internals resulting in error when doing fillna after truncate not changing size of panel (GH1823)
- Prevent segfault due to MultiIndex not being supported in HDFStore table format (GH1848)
- Fix UnboundLocalError in Panel.__setitem__ and add better error (GH1826)
- Fix to_csv issues with list of string entries. Isnull works on list of strings now too (GH1791)
- Fix Timestamp comparisons with datetime values outside the nanosecond range (1677-2262)
- Revert to prior behavior of normalize_date with datetime.date objects (return datetime)
- Fix broken interaction between np.nansum and Series.any/all
- Fix bug with multiple column date parsers (GH1866)
- DatetimeIndex.union(Int64Index) was broken
- Make plot x vs y interface consistent with integer indexing (GH1842)
- set_index inplace modified data even if unique check fails (GH1831)
- Only use Q-OCT/NOV/DEC in quarterly frequency inference (GH1789)
- Upcast to dtype=object when unstacking boolean DataFrame (GH1820)
- Fix float64/float32 merging bug (GH1849)
- Fixes to Period.start_time for non-daily frequencies (GH1857)
- Fix failure when converter used on index_col in read_csv (GH1835)
- Implement PeriodIndex.append so that pandas.concat works correctly (GH1815)
- Avoid Cython out-of-bounds access causing segfault sometimes in pad_2d, backfill_2d
- Fix resampling error with intraday times and anchored target time (like AS-DEC) (GH1772)
- Fix .ix indexing bugs with mixed-integer indexes (GH1799)
- Respect passed color keyword argument in Series.plot (GH1890)
- Fix rolling_min/max when the window is larger than the size of the input array. Check other malformed inputs (GH1899, GH1897)
- Rolling variance / standard deviation with only a single observation in window (GH1884)
- Fix unicode sheet name failure in to_excel (GH1828)
- Override DatetimeIndex.min/max to return Timestamp objects (GH1895)
- Fix column name formatting issue in length-truncated column (GH1906)
- Fix broken handling of copying Index metadata to new instances created by view(...) calls inside the NumPy infrastructure
• Support datetime.date again in DateOffset.rollback/rollforward
• Raise Exception if set passed to Series constructor (GH1913)
• Add TypeError when appending HDFStore table w/ wrong index type (GH1881)
• Don’t raise exception on empty inputs in EW functions (e.g. ewma) (GH1900)
• Make asof work correctly with PeriodIndex (GH1883)
• Fix extlinks in doc build
• Fill boolean DataFrame with NaN when calling shift (GH1814)
• Fix setuptools bug causing pip not to Cythonize .pyx files sometimes
• Fix negative integer indexing regression in .ix from 0.7.x (GH1888)
• Fix error while retrieving timezone and utc offset from subclasses of datetime.tzinfo without .zone and ._utcoff-set attributes (GH1922)
• Fix DataFrame formatting of small, non-zero FP numbers (GH1911)
• Various fixes by upcasting of date -> datetime (GH1395)
• Raise better exception when passing multiple functions with the same name, such as lambdas, to GroupBy.aggregate
• Fix DataFrame.apply with axis=1 on a non-unique index (GH1878)
• Proper handling of Index subclasses in pandas.unique (GH1759)
• Set index names in DataFrame.from_records (GH1744)
• Fix time series indexing error with duplicates, under and over hash table size cutoff (GH1821)
• Handle list keys in addition to tuples in DataFrame.xs when partial-indexing a hierarchically-indexed DataFrame (GH1796)
• Support multiple column selection in DataFrame.__getitem__ with duplicate columns (GH1943)
• Fix time zone localization bug causing improper fields (e.g. hours) in time zones that have not had a UTC transition in a long time (GH1946)
• Fix errors when parsing and working with with fixed offset timezones (GH1922, GH1928)
• Fix text parser bug when handling UTC datetime objects generated by dateutil (GH1693)
• Fix plotting bug when ‘B’ is the inferred frequency but index actually contains weekends (GH1668, GH1669)
• Fix plot styling bugs (GH1666, GH1665, GH1658)
• Fix plotting bug with index/columns with unicode (GH1685)
• Fix DataFrame constructor bug when passed Series with datetime64 dtype in a dict (GH1680)
• Fixed regression in generating DatetimeIndex using timezone aware datetime.datetime (GH1676)
• Fix DataFrame bug when printing concatenated DataFrames with duplicated columns (GH1675)
• Fixed bug when plotting time series with multiple intraday frequencies (GH1732)
• Fix bug in DataFrame.duplicated to enable iterables other than list-types as input argument (GH1773)
• Fix resample bug when passed list of lambdas as how argument (GH1808)
• Repr fix for MultiIndex level with all NAs (GH1971)
• Fix PeriodIndex slicing bug when slice start/end are out-of-bounds (GH1977)
• Fix read_table bug when parsing unicode (GH1975)
• Fix BlockManager.iget bug when dealing with non-unique MultiIndex as columns (GH1970)
• Fix reset_index bug if both drop and level are specified (GH1957)
• Work around unsafe NumPy object->int casting with Cython function (GH1987)
• Fix datetime64 formatting bug in DataFrame.to_csv (GH1993)
• Default start date in pandas.io.data to 1/1/2000 as the docs say (GH2011)

37.26 pandas 0.8.1

Release date: July 22, 2012

37.26.1 New Features

• Add vectorized, NA-friendly string methods to Series (GH1621, GH620)
• Can pass dict of per-column line styles to DataFrame.plot (GH1559)
• Selective plotting to secondary y-axis on same subplot (GH1640)
• Add new bootstrap_plot plot function
• Add new parallel_coordinates plot function (GH1488)
• Add radviz plot function (GH1566)
• Add multi_sparse option to set_printoptions to modify display of hierarchical indexes (GH1538)
• Add dropna method to Panel (GH171)

37.26.2 Improvements to existing features

• Use moving min/max algorithms from Bottleneck in rolling_min/rolling_max for > 100x speedup. (GH1504, GH50)
• Add Cython group median method for >15x speedup (GH1358)
• Drastically improve to_datetime performance on ISO8601 datetime strings (with no time zones) (GH1571)
• Improve single-key groupby performance on large data sets, accelerate use of groupby with a Categorical variable
• Add ability to append hierarchical index levels with set_index and to drop single levels with reset_index (GH1569, GH1577)
• Always apply passed functions in resample, even if upsampling (GH1596)
• Avoid unnecessary copies in DataFrame constructor with explicit dtypes (GH1572)
• Cleaner DatetimeIndex string representation with 1 or 2 elements (GH1611)
• Improve performance of array-of-Period to PeriodIndex, convert such arrays to PeriodIndex inside Index (GH1215)
• More informative string representation for weekly Period objects (GH1503)
• Accelerate 3-axis multi data selection from homogeneous Panel (GH979)
• Add `adjust` option to `ewma` to disable adjustment factor (GH1584)
• Add new `matplotlib` converters for high frequency time series plotting (GH1599)
• Handling of tz-aware datetime.datetime objects in `to_datetime`; raise Exception unless `utc=True` given (GH1581)

37.26.3 Bug Fixes

• Fix NA handling in `DataFrame.to_panel` (GH1582)
• Handle `TypeError` issues inside `PyObject_RichCompareBool` calls in `khash` (GH1318)
• Fix resampling bug to lower case daily frequency (GH1588)
• Fix `kendall/spearman DataFrame.corr` bug with no overlap (GH1595)
• Fix bug in `DataFrame.set_index` (GH1592)
• Don’t ignore axes in `boxplot` if by specified (GH1565)
• Fix `Panel ix` indexing with integers bug (GH1603)
• Fix Partial indexing bugs (years, months, ...) with `PeriodIndex` (GH1601)
• Fix `MultiIndex` console formatting issue (GH1606)
• Unordered index with duplicates doesn’t yield scalar location for single entry (GH1586)
• Fix resampling of tz-aware time series with “anchored” freq (GH1591)
• Fix `DataFrame.rank` error on integer data (GH1589)
• Selection of multiple `SparseDataFrame` columns by list in `__getitem__` (GH1585)
• Override `Index.tolist` for compatibility with `MultiIndex` (GH1576)
• Fix hierarchical summing bug with `MultiIndex` of length 1 (GH1568)
• Work around numpy.concatenate use/bug in `Series.set_value` (GH1561)
• Ensure `Series/DataFrame` are sorted before resampling (GH1580)
• Fix unhandled `IndexError` when indexing very large time series (GH1562)
• Fix `DatetimeIndex` intersection logic error with irregular indexes (GH1551)
• Fix unit test errors on Python 3 (GH1550)
• Fix `ix` indexing bugs in duplicate `DataFrame` index (GH1201)
• Better handle errors with non-existing objects in `HDFStore` (GH1254)
• Don’t copy int64 array data in `DatetimeIndex` when `copy=False` (GH1624)
• Fix resampling of conforming periods quarterly to annual (GH1622)
• Don’t lose index name on resampling (GH1631)
• Support `python-dateutil` version 2.1 (GH1637)
• Fix broken `scatter_matrix` axis labeling, esp. with time series (GH1625)
• Fix cases where extra keywords weren’t being passed on to `matplotlib` from `Series.plot` (GH1636)
• Fix `BusinessMonthBegin` logic for dates before 1st bday of month (GH1645)
• Ensure string alias converted (valid in `DatetimeIndex.get_loc`) in `DataFrame.xs / __getitem__` (GH1644)
• Fix use of string alias timestamps with tz-aware time series (GH1647)
• Fix Series.max/min and Series.describe on len-0 series (GH1650)
• Handle None values in dict passed to concat (GH1649)
• Fix Series.interpolate with method='values' and DatetimeIndex (GH1646)
• Fix IndexError in left merges on a DataFrame with 0-length (GH1628)
• Fix DataFrame column width display with UTF-8 encoded characters (GH1620)
• Handle case in pandas.io.data.get_data_yahoo where Yahoo! returns duplicate dates for most recent business
day
• Avoid downsampling when plotting mixed frequencies on the same subplot (GH1619)
• Fix read_csv bug when reading a single line (GH1553)
• Fix bug in C code causing monthly periods prior to December 1969 to be off (GH1570)

37.27 pandas 0.8.0

Release date: 6/29/2012

37.27.1 New Features

• New unified DatetimeIndex class for nanosecond-level timestamp data
• New Timestamp datetime.datetime subclass with easy time zone conversions, and support for nanoseconds
• New PeriodIndex class for timespans, calendar logic, and Period scalar object
• High performance resampling of timestamp and period data. New resample method of all pandas data structures
• New frequency names plus shortcut string aliases like ‘15h’, ‘1h30min’
• Time series string indexing shorthand (GH222)
• Add week, dayofyear array and other timestamp array-valued field accessor functions to DatetimeIndex
• Add GroupBy.prod optimized aggregation function and ‘prod’ fast time series conversion method (GH1018)
• Implement robust frequency inference function and inferred_freq attribute on DatetimeIndex (GH391)
• New tz_convert and tz_localize methods in Series / DataFrame
• Convert DatetimeIndexes to UTC if time zones are different in join/setops (GH864)
• Add limit argument for forward/backward filling to reindex, fillna, etc. (GH825 and others)
• Add support for indexes (dates or otherwise) with duplicates and common sense indexing/selection functionality
• Series/DataFrame.update methods, in-place variant of combine_first (GH961)
• Add match function to API (GH502)
• Add Cython-optimized first, last, min, max, prod functions to GroupBy (GH994, GH1043)
• Dates can be split across multiple columns (GH1227, GH1186)
• Add experimental support for converting pandas DataFrame to R data.frame via rpy2 (GH350, GH1212)
• Can pass list of (name, function) to GroupBy.aggregate to get aggregates in a particular order (GH610)
• Can pass dicts with lists of functions or dicts to GroupBy aggregate to do much more flexible multiple function
aggregation (GH642, GH610)
• New ordered_merge functions for merging DataFrames with ordered data. Also supports group-wise merging for panel data (GH813)
  • Add keys() method to DataFrame
  • Add flexible replace method for replacing potentially values to Series and DataFrame (GH929, GH1241)
  • Add ‘kde’ plot kind for Series/DataFrame.plot (GH1059)
  • More flexible multiple function aggregation with GroupBy
  • Add pct_change function to Series/DataFrame
  • Add option to interpolate by Index values in Series.interpolate (GH1206)
  • Add max_colwidth option for DataFrame, defaulting to 50
  • Conversion of DataFrame through rpy2 to R data.frame (GH1282, )
  • Add keys() method on DataFrame (GH1240)
  • Add new match function to API (similar to R) (GH502)
  • Add dayfirst option to parsers (GH854)
  • Add method argument to align method for forward/backward fillin (GH216)
  • Add Panel.transpose method for rearranging axes (GH695)
  • Add new cut function (patterned after R) for discretizing data into equal range-length bins or arbitrary breaks of your choosing (GH415)
  • Add new qcut for cutting with quantiles (GH1378)
  • Add value_counts top level array method (GH1392)
  • Added Andrews curves plot tupe (GH1325)
  • Add lag plot (GH1440)
  • Add autocorrelation_plot (GH1425)
  • Add support for tox and Travis CI (GH1382)
  • Add support for Categorical use in GroupBy (GH292)
  • Add any and all methods to DataFrame (GH1416)
  • Add secondary_y option to Series.plot
  • Add experimental lreshape function for reshaping wide to long

37.27.2 Improvements to existing features

• Switch to klib/khash-based hash tables in Index classes for better performance in many cases and lower memory footprint
• Shipping some functions from scipy.stats to reduce dependency, e.g. Series.describe and DataFrame.describe (GH1092)
• Can create MultiIndex by passing list of lists or list of arrays to Series, DataFrame constructor, etc. (GH831)
• Can pass arrays in addition to column names to DataFrame.set_index (GH402)
• Improve the speed of “square” reindexing of homogeneous DataFrame objects by significant margin (GH836)
• Handle more dtypes when passed MaskedArrays in DataFrame constructor (GH406)
• Improved performance of join operations on integer keys (GH682)
• Can pass multiple columns to GroupBy object, e.g. grouped[[col1, col2]] to only aggregate a subset of the value columns (GH383)
• Add histogram / kde plot options for scatter_matrix diagonals (GH1237)
• Add inplace option to Series/DataFrame.rename and sort_index, DataFrame.drop_duplicates (GH805, GH207)
• More helpful error message when nothing passed to Series.reindex (GH1267)
• Can mix array and scalars as dict-value inputs to DataFrame ctor (GH1329)
• Use DataFrame columns’ name for legend title in plots
• Preserve frequency in DatetimeIndex when possible in boolean indexing operations
• Promote datetime.date values in data alignment operations (GH867)
• Add order method to Index classes (GH1028)
• Avoid hash table creation in large monotonic hash table indexes (GH1160)
• Store time zones in HDFStore (GH1232)
• Enable storage of sparse data structures in HDFStore (GH85)
• Enable Series.asof to work with arrays of timestamp inputs
• Cython implementation of DataFrame.corr speeds up by > 100x (GH1349, GH1354)
• Exclude “nuisance” columns automatically in GroupBy.transform (GH1364)
• Support functions-as-strings in GroupBy.transform (GH1362)
• Use index name as xlabel/ylabel in plots (GH1415)
• Add convert_dtype option to Series.apply to be able to leave data as dtype=object (GH1414)
• Can specify all index level names in concat (GH1419)
• Add dialect keyword to parsers for quoting conventions (GH1363)
• Enable DataFrame[bool_DataFrame] += value (GH1366)
• Add retries argument to get_data_yahoo to try to prevent Yahoo! API 404s (GH826)
• Improve performance of reshaping by using O(N) categorical sorting
• Series names will be used for index of DataFrame if no index passed (GH1494)
• Header argument in DataFrame.to_csv can accept a list of column names to use instead of the object’s columns (GH921)
• Add raise_conflict argument to DataFrame.update (GH1526)
• Support file-like objects in ExcelFile (GH1529)

37.27.3 API Changes

• Rename pandas._series to pandas.lib
• Rename Factor to Categorical and add improvements. Numerous Categorical bug fixes
• Frequency name overhaul, WEEKDAY/EOM and rules with @ deprecated. get Legacy_offset_name backwards compatibility function added
• Raise ValueError in DataFrame.__nonzero__, so “if df” no longer works (GH1073)
• Change BDay (business day) to not normalize dates by default (GH506)
• Remove deprecated DataMatrix name
• Default merge suffixes for overlap now have underscores instead of periods to facilitate tab completion, etc. (GH1239)
• Deprecation of offset, time_rule timeRule parameters throughout codebase
• Series.append and DataFrame.append no longer check for duplicate indexes by default, add verify_integrity parameter (GH1394)
• Refactor Factor class, old constructor moved to Factor.from_array
• Modified internals of MultiIndex to use less memory (no longer represented as array of tuples) internally, speed up construction time and many methods which construct intermediate hierarchical indexes (GH1467)

37.27.4 Bug Fixes
• Fix OverflowError from storing pre-1970 dates in HDFStore by switching to datetime64 (GH179)
• Fix logical error with February leap year end in YearEnd offset
• Series([False, nan]) was getting casted to float64 (GH1074)
• Fix binary operations between boolean Series and object Series with booleans and NAs (GH1074, GH1079)
• Couldn’t assign whole array to column in mixed-type DataFrame via .ix (GH1142)
• Fix label slicing issues with float index values (GH1167)
• Fix segfault caused by empty groups passed to groupby (GH1048)
• Fix occasionally misbehaved reindexing in the presence of NaN labels (GH522)
• Fix imprecise logic causing weird Series results from .apply (GH1183)
• Unstack multiple levels in one shot, avoiding empty columns in some cases. Fix pivot table bug (GH1181)
• Fix formatting of MultiIndex on Series/DataFrame when index name coincides with label (GH1217)
• Handle Excel 2003 #N/A as NaN from xld (GH1213, GH1225)
• Fix timestamp locale-related deserialization issues with HDFStore by moving to datetime64 representation (GH1081, GH809)
• Fix DataFrame.duplicated/drop_duplicates NA value handling (GH557)
• Actually raise exceptions in fast reducer (GH1243)
• Fix various timezone-handling bugs from 0.7.3 (GH969)
• GroupBy on level=0 discarded index name (GH1313)
• Better error message with unmergeable DataFrames (GH1307)
• Series._repr__ alignment fix with unicode index values (GH1279)
• Better error message if nothing passed to reindex (GH1267)
• More robust NA handling in DataFrame.drop_duplicates (GH557)
• Resolve locale-based and pre-epoch HDF5 timestamp deserialization issues (GH973, GH1081, GH179)
• Implement Series.repeat (GH1229)
• Fix indexing with namedtuple and other tuple subclasses (GH1026)
• Fix float64 slicing bug (GH1167)
• Parsing integers with commas (GH796)
• Fix groupby improper data type when group consists of one value (GH1065)
• Fix negative variance possibility in nanvar resulting from floating point error (GH1090)
• Consistently set name on groupby pieces (GH184)
• Treat dict return values as Series in GroupBy.apply (GH823)
• Respect column selection for DataFrame in in GroupBy.transform (GH1365)
• Fix MultiIndex partial indexing bug (GH1352)
• Enable assignment of rows in mixed-type DataFrame via .ix (GH1432)
• Reset index mapping when grouping Series in Cython (GH1423)
• Fix outer/inner DataFrame.join with non-unique indexes (GH1421)
• Fix MultiIndex groupby bugs with empty lower levels (GH1401)
• Calling fillna with a Series will have same behavior as with dict (GH1486)
• SparseSeries reduction bug (GH1375)
• Fix unicode serialization issue in HDFStore (GH1361)
• Pass keywords to pyplot.boxplot in DataFrame.boxplot (GH1493)
• Bug fixes in MonthBegin (GH1483)
• Preserve MultiIndex names in drop (GH1513)
• Fix Panel DataFrame slice-assignment bug (GH1533)
• Don’t use locals() in read_* functions (GH1547)

37.28 pandas 0.7.3

Release date: April 12, 2012

37.28.1 New Features

• Support for non-unique indexes: indexing and selection, many-to-one and many-to-many joins (GH1306)
• Added fixed-width file reader, read_fwf (GH952)
• Add group_keys argument to groupby to not add group names to MultiIndex in result of apply (GH938)
• DataFrame can now accept non-integer label slicing (GH946). Previously only DataFrame.ix was able to do so.
• DataFrame.apply now retains name attributes on Series objects (GH983)
• Numeric DataFrame comparisons with non-numeric values now raises proper TypeError (GH943). Previously raise “PandasError: DataFrame constructor not properly called!”
• Add kurt methods to Series and DataFrame (GH964)
• Can pass dict of column -> list/set NA values for text parsers (GH754)
• Allows users specified NA values in text parsers (GH754)
• Parsers checks for openpyxl dependency and raises ImportError if not found (GH1007)
• New factory function to create HDFStore objects that can be used in a with statement so users do not have to explicitly call HDFStore.close (GH1005)
• pivot_table is now more flexible with same parameters as groupby (GH941)
• Added stacked bar plots (GH987)
• scatter_matrix method in pandas/tools/plotting.py (GH935)
• DataFrame.boxplot returns plot results for ex-post styling (GH985)
• Short version number accessible as pandas.version.short_version (GH930)
• Additional documentation in panel.to_frame (GH942)
• More informative Series.apply docstring regarding element-wise apply (GH977)
• Notes on rpy2 installation (GH1006)
• Add rotation and font size options to hist method (GH1012)
• Use exogenous / X variable index in result of OLS.y_predict. Add OLS.predict method (GH1027, GH1008)

37.28.2 API Changes

• Calling apply on grouped Series, e.g. describe(), will no longer yield DataFrame by default. Will have to call unstack() to get prior behavior
• NA handling in non-numeric comparisons has been tightened up (GH933, GH953)
• No longer assign dummy names key_0, key_1, etc. to groupby index (GH1291)

37.28.3 Bug Fixes

• Fix logic error when selecting part of a row in a DataFrame with a MultiIndex index (GH1013)
• Series comparison with Series of differing length causes crash (GH1016).
• Fix bug in indexing when selecting section of hierarchically-indexed row (GH1013)
• DataFrame.plot(logy=True) has no effect (GH1011).
• Broken arithmetic operations between SparsePanel-Panel (GH1015)
• Unicode repr issues in MultiIndex with non-ASCII characters (GH1010)
• DataFrame.lookup() returns inconsistent results if exact match not present (GH1001)
• DataFrame arithmetic operations not treating None as NA (GH992)
• DataFrameGroupBy.apply returns incorrect result (GH991)
• Series.reshape returns incorrect result for multiple dimensions (GH989)
• Series.std and Series.var ignores ddof parameter (GH934)
• DataFrame.append loses index names (GH980)
• DataFrame.plot(kind="bar") ignores color argument (GH958)
• Inconsistent Index comparison results (GH948)
• Improper int dtype DataFrame construction from data with NaN (GH846)
• Removes default ‘result’ name in groupby results (GH995)
• DataFrame.from_records no longer mutate input columns (GH975)
• Use Index name when grouping by it (GH1313)

37.29 pandas 0.7.2

Release date: March 16, 2012

37.29.1 New Features

• Add additional tie-breaking methods in DataFrame.rank (GH874)
• Add ascending parameter to rank in Series, DataFrame (GH875)
• Add sort_columns parameter to allow unsorted plots (GH918)
• IPython tab completion on GroupBy objects

37.29.2 API Changes

• Series.sum returns 0 instead of NA when called on an empty series. Analogously for a DataFrame whose rows or columns are length 0 (GH844)

37.29.3 Improvements to existing features

• Don’t use groups dict in Grouper.size (GH860)
• Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
• Enable column access via attributes on GroupBy (GH882)
• Enable setting existing columns (only) via attributes on DataFrame, Panel (GH883)
• Intercept __builtin__.sum in groupby (GH885)
• Can pass dict to DataFrame.fillna to use different values per column (GH661)
• Can select multiple hierarchical groups by passing list of values in .ix (GH134)
• Add level keyword to drop for dropping values from a level (GH159)
• Add coerce_float option on DataFrame.from_records (GH893)
• Raise exception if passed date_parser fails in read_csv
• Add axis option to DataFrame.fillna (GH174)
• Fixes to Panel to make it easier to subclass (GH888)

37.29.4 Bug Fixes

• Fix overflow-related bugs in groupby (GH850, GH851)
• Fix unhelpful error message in parsers (GH856)
• Better err msg for failed boolean slicing of dataframe (GH859)
• Series.count cannot accept a string (level name) in the level argument (GH869)
• Group index platform int check (GH870)
• concat on axis=1 and ignore_index=True raises TypeError (GH871)
• Further unicode handling issues resolved (GH795)
• Fix failure in multiindex-based access in Panel (GH880)
• Fix DataFrame boolean slice assignment failure (GH881)
• Fix combineAdd NotImplementedError for SparseDataFrame (GH887)
• Fix DataFrame.to_html encoding and columns (GH890, GH891, GH909)
• Fix na-filling handling in mixed-type DataFrame (GH910)
• Fix to DataFrame.set_value with non-existant row/col (GH911)
• Fix malformed block in groupby when excluding nuisance columns (GH916)
• Fix inconsistant NA handling in dtype=object arrays (GH925)
• Fix missing center-of-mass computation in ewmcov (GH862)
• Don’t raise exception when opening read-only HDF5 file (GH847)
• Fix possible out-of-bounds memory access in 0-length Series (GH917)

37.30 pandas 0.7.1

Release date: February 29, 2012

37.30.1 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

37.30.2 Improvements to existing features

• Improve performance and memory usage of fillna on DataFrame
• Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH787)
37.30.3 Bug Fixes

- Fix memory leak when inserting large number of columns into a single DataFrame (GH790)
- Appending length-0 DataFrame with new columns would not result in those new columns being part of the resulting concatenated DataFrame (GH782)
- Fixed groupby corner case when passing dictionary grouper and as_index is False (GH819)
- Fixed bug whereby bool array sometimes had object dtype (GH820)
- Fix exception thrown on np.diff (GH816)
- Fix to_records where columns are non-strings (GH822)
- Fix Index.intersection where indices have incomparable types (GH811)
- Fix ExcelFile throwing an exception for two-line file (GH837)
- Add clearer error message in csv parser (GH835)
- Fix loss of fractional seconds in HDFStore (GH513)
- Fix DataFrame join where columns have datetimes (GH787)
- Work around numpy performance issue in take (GH817)
- Improve comparison operations for NA-friendliness (GH801)
- Fix indexing operation for floating point values (GH780, GH798)
- Fix groupby case resulting in malformed dataframe (GH814)
- Fix behavior of reindex of Series dropping name (GH812)
- Improve on redundant groupby computation (GH775)
- Catch possible NA assignment to int/bool series with exception (GH839)

37.31 pandas 0.7.0

Release date: 2/9/2012

37.31.1 New Features

- New 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 concat function for concatenating DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of DataFrame.append (GH468, GH479, GH273)
- Handle differently-indexed output values in DataFrame.apply (GH498)
- Can pass list of dicts (e.g., a list of shallow JSON objects) to DataFrame constructor (GH526)
- Add reorder_levels method to Series and DataFrame (GH534)
- Add dict-like get function to DataFrame and Panel (GH521)
- DataFrame.iterrows method for efficiently iterating through the rows of a DataFrame
- Added DataFrame.to_panel with code adapted from LongPanel.to_long
• 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 (PR GH554)
• Add logy option to Series.plot for log-scaling on the Y axis
• Add index, header, and justify options to DataFrame.to_string. Add option to (GH570, GH571)
• Can pass multiple DataFrames to DataFrame.join to join on index (GH115)
• Can pass multiple Panels to Panel.join (GH115)
• Can pass multiple DataFrames to DataFrame.append to concatenate (stack) and multiple Series to Series.append too
• 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)
• Add verbose option to read_csv and read_table to show number of NA values inserted in non-numeric columns (GH614)
• Can pass a list of dicts or Series to DataFrame.append to concatenate multiple rows (GH464)
• Add level argument to DataFrame.xs for selecting data from other MultiIndex levels. Can take one or more levels with potentially a tuple of keys for flexible retrieval of data (GH371, GH629)
• New crosstab function for easily computing frequency tables (GH170)
• Can pass a list of functions to aggregate with groupby on a DataFrame, yielding an aggregated result with hierarchical columns (GH166)
• Add integer-indexing functions iget in Series and irow/iget in DataFrame (GH628)
• Add new Series.unique function, significantly faster than numpy.unique (GH658)
• Add new cummin and cummax instance methods to Series and DataFrame (GH647)
• Add new value_range function to return min/max of a dataframe (GH288)
• Add drop parameter to reset_index method of DataFrame and added method to Series as well (GH699)
• Add isin method to Index objects, works just like Series.isin (GH GH657)
• Implement array interface on Panel so that ufuncs work (re: GH740)
• Add sort option to DataFrame.join (GH731)
• Improved handling of NAs (propagation) in binary operations with dtype=object arrays (GH737)
• Add abs method to Pandas objects
• Added algorithms module to start collecting central algos
37.31.2 API Changes

- Label-indexing with integer indexes now raises KeyError if a label is not found instead of falling back on location-based indexing (GH700)
- Label-based slicing via `ix` or `[]` on Series will now only work if exact matches for the labels are found or if the index is monotonic (for range selections)
- Label-based slicing and sequences of labels can be passed to `[]` on a Series for both getting and setting (GH86)
- `[]` operator (`__getitem__` and `__setitem__`) will raise KeyError with integer indexes when an index is not contained in the index. The prior behavior would fall back on position-based indexing if a key was not found in the index which would lead to subtle bugs. This is now consistent with the behavior of `.ix` on DataFrame and friends (GH328)
- Rename `DataFrame.delevel` to `DataFrame.reset_index` and add deprecation warning
- `Series.sort` (an in-place operation) called on a Series which is a view on a larger array (e.g. a column in a DataFrame) will generate an Exception to prevent accidentally modifying the data source (GH316)
- Refactor to remove deprecated `LongPanel` class (GH552)
- Deprecated `Panel.to_long`, renamed to `to_frame`
- Deprecated `colSpace` argument in `DataFrame.to_string`, renamed to `col_space`
- Rename `precision` to `accuracy` in engineering float formatter (GH395)
- The default delimiter for `read_csv` is comma rather than letting `csv.Sniffer` infer it
- Rename `col_or_columns` argument in `DataFrame.drop_duplicates` (GH734)

37.31.3 Improvements to existing features

- 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 (GH 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 Cythoned 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)
• Some performance enhancements in constructing a Panel from a dict of DataFrame objects
• Made Index._get_duplicates a public method by removing the underscore
• Prettier printing of floats, and column spacing fix (GH395, GH571)
• Add bold_rows option to DataFrame.to_html (GH586)
• Improve the performance of DataFrame.sort_index by up to 5x or more when sorting by multiple columns
• Substantially improve performance of DataFrame and Series constructors when passed a nested dict or dict, respectively (GH540, GH621)
• Modified setup.py so that pip / setuptools will install dependencies (GH507, various pull requests)
• Unstack called on DataFrame with non-MultiIndex will return Series (GH477)
• Improve DataFrame.to_string and console formatting to be more consistent in the number of displayed digits (GH395)
• Use bottleneck if available for performing NaN-friendly statistical operations that it implemented (GH91)
• Monkey-patch context to traceback in DataFrame.apply to indicate which row/column the function application failed on (GH614)
• Improved ability of read_table and read_clipboard to parse console-formatted DataFrames (can read the row of index names, etc.)
• Can pass list of group labels (without having to convert to an ndarray yourself) to groupby in some cases (GH659)
• Use kind argument to Series.order for selecting different sort kinds (GH668)
• Add option to Series.to_csv to omit the index (GH684)
• Add delimiter as an alternative to sep in read_csv and other parsing functions
• Substantially improved performance of groupby on DataFrames with many columns by aggregating blocks of columns all at once (GH745)
• Can pass a file handle or StringIO to Series/DataFrame.to_csv (GH765)
• Can pass sequence of integers to DataFrame.irow(icol) and Series.iget, (GH GH654)
• Prototypes for some vectorized string functions
• Add float64 hash table to solve the Series.unique problem with NAs (GH714)
• Memoize objects when reading from file to reduce memory footprint
• Can get and set a column of a DataFrame with hierarchical columns containing “empty” (‘’ ) lower levels without passing the empty levels (PR GH768)

37.31.4 Bug Fixes

• Raise exception in out-of-bounds indexing of Series instead of seg-faulting, regression from earlier releases (GH495)
• Fix error when joining DataFrames of different dtypes within the same typeclass (e.g. float32 and float64) (GH486)
• Fix bug in Series.min/Series.max on objects like datetime.datetime (GH GH487)
• Preserve index names in Index.union (GH501)
- Fix bug in Index joining causing subclass information (like DateRange type) to be lost in some cases (GH500)
- Accept empty list as input to DataFrame constructor, regression from 0.6.0 (GH491)
- Can output DataFrame and Series with ndarray objects in a dtype=object array (GH490)
- Return empty string from Series.to_string when called on empty Series (GH GH488)
- Fix exception passing empty list to DataFrame.from_records
- Fix Index.format bug (excluding name field) with datetimes with time info
- Fix scalar value access in Series to always return NumPy scalars, regression from prior versions (GH510)
- Handle rows skipped at beginning of file in read_* functions (GH505)
- Handle improper dtype casting in set_value methods
- Unary `-` / __neg__ operator on DataFrame was returning integer values
- Unbox 0-dim ndarrays from certain operators like all, any in Series
- Fix handling of missing columns (was combine_first-specific) in DataFrame.combine for general case (GH529)
- Fix type inference logic with boolean lists and arrays in DataFrame indexing
- Use centered sum of squares in R-square computation if entity_effects=True in panel regression
- Handle all NA case in Series.{corr, cov}, was raising exception (GH548)
- Aggregating by multiple levels with level argument to DataFrame, Series stat method, was broken (GH545)
- Fix Cython buf when converter passed to read_csv produced a numeric array (buffer dtype mismatch when passed to Cython type inference function) (GH GH546)
- Fix exception when setting scalar value using .ix on a DataFrame with a MultiIndex (GH551)
- Fix outer join between two DateRanges with different offsets that returned an invalid DateRange
- Cleanup DataFrame.from_records failure where index argument is an integer
- Fix Data.from_records failure when passed a dictionary
- Fix NA handling in {Series, DataFrame}.rank with non-floating point dtypes
- Fix bug related to integer type-checking in .ix-based indexing
- Handle non-string index name passed to DataFrame.from_records
- DataFrame.insert caused the columns name(s) field to be discarded (GH527)
- Fix erroneous in monotonic many-to-one left joins
- Fix DataFrame.to_string to remove extra column white space (GH571)
- Format floats to default to same number of digits (GH395)
- Added decorator to copy docstring from one function to another (GH449)
- Fix error in monotonic many-to-one left joins
- Fix __eq__ comparison between DateOffsets with different relativedelta keywords passed
- Fix exception caused by parser converter returning strings (GH583)
- Fix MultiIndex formatting bug with integer names (GH601)
- Fix bug in handling of non-numeric aggregates in Series.groupby (GH612)
- Fix TypeError with tuple subclasses (e.g. namedtuple) in DataFrame.from_records (GH611)
- Catch misreported console size when running IPython within Emacs
- Fix minor bug in pivot table margins, loss of index names and length-1 ‘All’ tuple in row labels
- Add support for legacy WidePanel objects to be read from HDFStore
- Fix out-of-bounds segfault in pad_object and backfill_object methods when either source or target array are empty
- Could not create a new column in a DataFrame from a list of tuples
- Fix bugs preventing SparseDataFrame and SparseSeries working with groupby (GH666)
- Use sort kind in Series.sort / argsort (GH668)
- Fix DataFrame operations on non-scalar, non-pandas objects (GH672)
- Don’t convert DataFrame column to integer type when passing integer to __setitem__ (GH669)
- Fix downstream bug in pivot_table caused by integer level names in MultiIndex (GH678)
- Fix SparseSeries.combine_first when passed a dense Series (GH687)
- Fix performance regression in HDFStore loading when DataFrame or Panel stored in table format with datetimes
- Raise Exception in DateRange when offset with n=0 is passed (GH683)
- Fix get/set inconsistency with .ix property and integer location but non-integer index (GH707)
- Use right dropna function for SparseSeries. Return dense Series for NA fill value (GH730)
- Fix Index.format bug causing incorrectly string-formatted Series with datetime indexes (GH726, GH758)
- Fix errors caused by object dtype arrays passed to ols (GH759)
- Fix error where column names lost when passing list of labels to DataFrame.__getitem__. (GH662)
- Fix error whereby top-level week iterator overwrote week instance
- Fix circular reference causing memory leak in sparse array / series / frame, (GH663)
- Fix integer-slicing from integers-as-floats (GH670)
- Fix zero division errors in nanops from object dtype arrays in all NA case (GH676)
- Fix csv encoding when using unicode (GH705, GH717, GH738)
- Fix assumption that each object contains every unique block type in concat, (GH708)
- Fix sortedness check of multiindex in to_panel (GH719, 720)
- Fix that None was not treated as NA in PyObjectHashtable
- Fix hashing dtype because of endianness confusion (GH747, GH748)
- Fix SparseSeries.dropna to return dense Series in case of NA fill value (GH GH730)
- Use map_infer instead of np.vectorize. handle NA sentinels if converter yields numeric array, (GH753)
- Fixes and improvements to DataFrame.rank (GH742)
- Fix catching AttributeError instead of NameError for bottleneck
- Try to cast non-MultiIndex to better dtype when calling reset_index (GH726 GH440)
- Fix #1.QNAN0` float bug on 2.6/win64
- Allow subclasses of dicts in DataFrame constructor, with tests
- Fix problem whereby set_index destroys column multiindex (GH764)
• Hack around bug in generating DateRange from naive DateOffset (GH770)
• Fix bug in DateRange.intersection causing incorrect results with some overlapping ranges (GH771)

37.31.5 Thanks

• Craig Austin
• Chris Billington
• Marius Cobzarenco
• Mario Gamboa-Cavazos
• Hans-Martin Gaudecker
• Arthur Gerigk
• Yaroslav Halchenko
• Jeff Hammerbacher
• Matt Harrison
• Andreas Hilboll
• Luc Kesters
• Adam Klein
• Gregg Lind
• Solomon Negusse
• Wouter Overmeire
• Christian Prinoth
• Jeff Reback
• Sam Reckoner
• Craig Reeson
• Jan Schulz
• Skipper Seabold
• Ted Square
• Graham Taylor
• Aman Thakral
• Chris Uga
• Dieter Vandenbussche
• Texas P.
• Pinxing Ye
• ... and everyone I forgot
37.32 pandas 0.6.1

Release date: 12/13/2011

37.32.1 API Changes

- Rename names argument in DataFrame.from_records to columns. Add deprecation warning.
- Boolean get/set operations on Series with boolean Series will reindex instead of requiring that the indexes be exactly equal (GH429)

37.32.2 New Features

- Can pass Series to DataFrame.append with ignore_index=True for appending a single row (GH430)
- Add Spearman and Kendall correlation options to Series.corr and DataFrame.corr (GH428)
- Add new get_value and set_value methods to Series, DataFrame, and Panel to very low-overhead access to scalar elements. df.get_value(row, column) is about 3x faster than df[column][row] by handling fewer cases (GH437, GH438). Add similar methods to sparse data structures for compatibility
- Add Qt table widget to sandbox (GH435)
- DataFrame.align can accept Series arguments, add axis keyword (GH461)
- Implement new SparseList and SparseArray data structures. SparseSeries now derives from SparseArray (GH463)
- max_columns / max_rows options in set_printoptions (GH453)
- Implement Series.rank and DataFrame.rank, 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 (GH GH114)
- Add Series.from_csv function (GH482)

37.32.3 Improvements to existing features

- Improve memory usage of DataFrame.describe (do not copy data unnecessarily) (GH425)
- Use same formatting function for outputting floating point Series to console as in DataFrame (GH420)
- DataFrame.delevel will try to infer better dtype for new columns (GH440)
- Exclude non-numeric types in DataFrame.{corr, cov}
- Override Index.astype to enable dtype casting (GH412)
- Use same float formatting function for Series.__repr__ (GH420)
- Use available console width to output DataFrame columns (GH453)
- Accept ndarrays when setting items in Panel (GH452)
- Infer console width when printing __repr__ of DataFrame to console (PR GH453)
• Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
• Can pass DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH462)
• 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 GH158)
• MultiIndex.get_level_values can take the level name
• More helpful error message when DataFrame.plot fails on one of the columns (GH478)
• Improve performance of DataFrame.{index, columns} attribute lookup

37.32.4 Bug Fixes

• Fix O(K^2) memory leak caused by inserting many columns without consolidating, had been present since 0.4.0 (GH467)
• DataFrame.count should return Series with zero instead of NA with length-0 axis (GH423)
• Fix Yahoo! Finance API usage in pandas.io.data (GH419, GH427)
• Fix upstream bug causing failure in Series.align with empty Series (GH434)
• Function passed to DataFrame.apply can return a list, as long as it’s the right length. Regression from 0.4 (GH432)
• Don’t “accidentally” upcast scalar values when indexing using .ix (GH431)
• Fix groupby exception raised with as_index=False and single column selected (GH421)
• Implement DateOffset.__ne__ causing downstream bug (GH456)
• Fix __doc__-related issue when converting py -> pyo with py2exe
• Bug fix in left join Cython code with duplicate monotonic labels
• Fix bug when unstacking multiple levels described in GH451
• Exclude NA values in dtype=object arrays, regression from 0.5.0 (GH469)
• Use Cython map_infer function in DataFrame.applymap to properly infer output type, handle tuple return values and other things that were breaking (GH465)
• Handle floating point index values in HDFStore (GH454)
• Fixed stale column reference bug (cached Series object) caused by type change / item deletion in DataFrame (GH473)
• Index.get_loc should always raise Exception when there are duplicates
• Handle differently-indexed Series input to DataFrame constructor (GH475)
• Omit nuisance columns in multi-groupby with Python function
• Buglet in handling of single grouping in general apply
• Handle type inference properly when passing list of lists or tuples to DataFrame constructor (GH484)
• Preserve Index / MultiIndex names in GroupBy.apply concatenation step (GH GH481)
37.32.5 Thanks

- Ralph Bean
- Luca Beltrame
- Marius Cobzarenco
- Andreas Hilboll
- Jev Kuznetsov
- Adam Lichtenstein
- Wouter Overmeire
- Fernando Perez
- Nathan Pinger
- Christian Prinoth
- Alex Reyfman
- Joon Ro
- Chang She
- Ted Square
- Chris Uga
- Dieter Vandenbussche

37.33 pandas 0.6.0

Release date: 11/25/2011

37.33.1 API Changes

- Arithmetic methods like `sum` will attempt to sum dtype=object values by default instead of excluding them (GH382)

37.33.2 New Features

- Add `melt` function to `pandas.core.reshape`
- Add `level` parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Add `head` and `tail` methods to Series, analogous to to DataFrame (PR GH296)
- Add `Series.isin` function which checks if each value is contained in a passed sequence (GH289)
- Add `float_format` option to `Series.to_string`
- Add `skip_footer` (GH291) and `converters` (GH343) options to `read_csv` and `read_table`
- Add proper, tested weighted least squares to standard and panel OLS (GH GH303)
- Add `drop_duplicates` and `duplicated` functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
• Implement logical (boolean) operators &, |, ^ on DataFrame (GH347)
• Add `Series.mad`, mean absolute deviation, matching DataFrame
• Add `QuarterEnd` DateOffset (GH321)
• Add matrix multiplication function `dot` to DataFrame (GH65)
• Add `orient` option to `Panel.from_dict` to ease creation of mixed-type Panels (GH359, GH301)
• Add `DataFrame.from_dict` with similar `orient` option
• Can now pass list of tuples or list of lists to `DataFrame.from_records` for fast conversion to DataFrame (GH357)
• Can pass multiple levels to groupby, e.g. `df.groupby(level=[0, 1])` (GH GH103)
• Can sort by multiple columns in `DataFrame.sort_index` (GH92, GH362)
• Add fast `get_value` and `put_value` methods to DataFrame and micro-performance tweaks (GH360)
• Add `cov` instance methods to Series and DataFrame (GH194, GH362)
• Add bar plot option to `DataFrame.plot` (GH348)
• Add `idxmin` and `idxmax` functions to Series and DataFrame for computing index labels achieving maximum and minimum values (GH286)
• Add `read_clipboard` function for parsing DataFrame from OS clipboard, should work across platforms (GH300)
• Add `nunique` function to Series for counting unique elements (GH297)
• DataFrame constructor will use Series name if no columns passed (GH373)
• Support regular expressions and longer delimiters in read_table/read_csv, but does not handle quoted strings yet (GH364)
• Add `DataFrame.to_html` for formatting DataFrame to HTML (GH387)
• MaskedArray can be passed to DataFrame constructor and masked values will be converted to NaN (GH396)
• Add `DataFrame.boxplot` function (GH368, others)
• Can pass extra args, kwds to DataFrame.apply (GH376)

### 37.33.3 Improvements to existing features

• Raise more helpful exception if date parsing fails in DateRange (GH298)
• Vastly improved performance of GroupBy on axes with a MultiIndex (GH299)
• Print level names in hierarchical index in Series repr (GH305)
• Return DataFrame when performing GroupBy on selected column and as_index=False (GH308)
• Can pass vector to `on` argument in `DataFrame.join` (GH312)
• Don’t show Series name if it’s None in the repr, also omit length for short Series (GH317)
• Show legend by default in `DataFrame.plot`, add `legend` boolean flag (GH GH324)
• Significantly improved performance of `Series.order`, which also makes `np.unique` called on a Series faster (GH327)
• Faster cythonized count by level in Series and DataFrame (GH341)
• Raise exception if dateutil 2.0 installed on Python 2.x runtime (GH346)
• Significant GroupBy performance enhancement with multiple keys with many “empty” combinations
• New Cython vectorized function `map_infer` speeds up `Series.apply` and `Series.map` significantly when passed elementwise Python function, motivated by GH355

• Cythonized `cache_readonly`, resulting in substantial micro-performance enhancements throughout the codebase (GH361)

• Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than `np.apply_along_axis` (GH309)

• Add raw option to `DataFrame.apply` for getting better performance when the passed function only requires an ndarray (GH309)

• Improve performance of `MultiIndex.from_tuples`

• Can pass multiple levels to `stack` and `unstack` (GH370)

• Can pass multiple values columns to `pivot_table` (GH381)

• Can call `DataFrame.delevel` with standard Index with name set (GH393)

• Use Series name in GroupBy for result index (GH363)

• Refactor Series/DataFrame stat methods to use common set of NaN-friendly function

• Handle NumPy scalar integers at C level in Cython conversion routines

### 37.33.4 Bug Fixes

• Fix bug in `DataFrame.to_csv` when writing a DataFrame with an index name (GH290)

• DataFrame should clear its Series caches on consolidation, was causing “stale” Series to be returned in some corner cases (GH304)

• DataFrame constructor failed if a column had a list of tuples (GH293)

• Ensure that `Series.apply` always returns a Series and implement `Series.round` (GH314)

• Support boolean columns in Cythonized groupby functions (GH315)

• `DataFrame.describe` should not fail if there are no numeric columns, instead return categorical describe (GH323)

• Fixed bug which could cause columns to be printed in wrong order in `DataFrame.to_string` if specific list of columns passed (GH325)

• Fix legend plotting failure if DataFrame columns are integers (GH326)

• Shift start date back by one month for Yahoo! Finance API in pandas.io.data (GH329)

• Fix `DataFrame.join` failure on unconsolidated inputs (GH331)

• DataFrame.min/max will no longer fail on mixed-type DataFrame (GH337)

• Fix `read_csv / read_table` failure when passing list to `index_col` that is not in ascending order (GH349)

• Fix failure passing Int64Index to Index.union when both are monotonic

• Fix error when passing SparseSeries to (dense) DataFrame constructor

• Added missing bang at top of setup.py (GH352)

• Change `is_monotonic` on MultiIndex so it properly compares the tuples

• Fix MultiIndex outer join logic (GH351)

• Set index name attribute with single-key groupby (GH358)
• Bug fix in reflexive binary addition in Series and DataFrame for non-commutative operations (like string concatenation) (GH353)
• setupegg.py will invoke Cython (GH192)
• Fix block consolidation bug after inserting column into MultiIndex (GH366)
• Fix bug in join operations between Index and Int64Index (GH367)
• Handle min_periods=0 case in moving window functions (GH365)
• Fixed corner cases in DataFrame.apply/pivot with empty DataFrame (GH378)
• Fixed repr exception when Series name is a tuple
• Always return DateRange from asfreq (GH390)
• Pass level names to swaplavel (GH379)
• Don’t lose index names in MultiIndex.droplevel (GH394)
• Infer more proper return type in DataFrame.apply when no columns or rows depending on whether the passed function is a reduction (GH389)
• Always return NA/NaN from Series.min/max and DataFrame.min/max when all of a row/column/values are NA (GH384)
• Enable partial setting with .ix / advanced indexing (GH397)
• Handle mixed-type DataFrames correctly in unstack, do not lose type information (GH403)
• Fix integer name formatting bug in Index.format and in Series.__repr__
• Handle label types other than string passed to groupby (GH405)
• Fix bug in .ix-based indexing with partial retrieval when a label is not contained in a level
• Index name was not being pickled (GH408)
• Level name should be passed to result index in GroupBy.apply (GH416)

37.33.5 Thanks

• Craig Austin
• Marius Cobzarenco
• Joel Cross
• Jeff Hammerbacher
• Adam Klein
• Thomas Kluyver
• Jev Kuznetsov
• Kieran O’Mahony
• Wouter Overmeire
• Nathan Pinger
• Christian Prinoth
• Skipper Seabold
• Chang She
**37.34 pandas 0.5.0**

**Release date:** 10/24/2011

This release of pandas includes a number of API changes (see below) and cleanup of deprecated APIs from pre-0.4.0 releases. There are also bug fixes, new features, numerous significant performance enhancements, and includes a new ipython completer hook to enable tab completion of DataFrame columns accesses and attributes (a new feature).

In addition to the changes listed here from 0.4.3 to 0.5.0, the minor releases 4.1, 0.4.2, and 0.4.3 brought some significant new functionality and performance improvements that are worth taking a look at.

Thanks to all for bug reports, contributed patches and generally providing feedback on the library.

### 37.34.1 API Changes

- `read_table`, `read_csv`, and `ExcelFile.parse` default arguments for `index_col` is now None. To use one or more of the columns as the resulting DataFrame’s index, these must be explicitly specified now
- Parsing functions like `read_csv` no longer parse dates by default (GH GH225)
- Removed `weights` option in panel regression which was not doing anything principled (GH155)
- Changed `buffer` argument name in `Series.to_string` to `buf`
- `Series.to_string` and `DataFrame.to_string` now return strings by default instead of printing to sys.stdout
- Deprecated `nanRep` argument in various `to_string` and `to_csv` functions in favor of `na_rep`. Will be removed in 0.6 (GH275)
- Renamed `delimiter` to `sep` in `DataFrame.from_csv` for consistency
- Changed order of `Series.clip` arguments to match those of `numpy.clip` and added (unimplemented) `out` argument so `numpy.clip` can be called on a Series (GH272)
- Series functions renamed (and thus deprecated) in 0.4 series have been removed:
  - `asOf`, use `asof`
  - `toDict`, use `to_dict`
  - `toString`, use `to_string`
  - `toCSV`, use `to_csv`
  - `merge`, use `map`
  - `applymap`, use `apply`
  - `combineFirst`, use `combine_first`
  - `_firstTimeWithValue` use `first_valid_index`
DataFrame functions renamed / deprecated in 0.4 series have been removed:
- `asMatrix` method, use `as_matrix` or `values` attribute
- `combineFirst`, use `combine_first`
- `getXS`, use `xs`
- `merge`, use `join`
- `fromRecords`, use `from_records`
- `fromcsv`, use `from_csv`
- `toRecords`, use `to_records`
- `toDict`, use `to_dict`
- `toString`, use `to_string`
- `toCSV`, use `to_csv`
- `_lastTimeWithValue` use `last_valid_index`
- `asMatrix` method, use `as_matrix` or `values` attribute
- `combineFirst`, use `combine_first`
- `getXS`, use `xs`
- `merge`, use `join`
- `fromRecords`, use `from_records`
- `fromcsv`, use `from_csv`
- `toRecords`, use `to_records`
- `toDict`, use `to_dict`
- `toString`, use `to_string`
- `toCSV`, use `to_csv`
- `_lastTimeWithValue` use `last_valid_index`
- `toDataMatrix` is no longer needed
- `rows()` method, use `index` attribute
- `cols()` method, use `columns` attribute
- `dropEmptyRows()`, use `dropna(how='all')`
- `dropIncompleteRows()`, use `dropna()`
- `tapply(f)`, use `apply(f, axis=1)`
- `tgroupby(keyfunc, aggfunc)`, use `groupby` with `axis=1`

### 37.34.2 Deprecations Removed
- `indexField` argument in `DataFrame.from_records`
- `missingAtEnd` argument in `Series.order`. Use `na_last` instead
- `Series.fromValue` classmethod, use regular `Series` constructor instead
- Functions `parseCSV`, `parseText`, and `parseExcel` methods in `pandas.io.parsers` have been removed
- `Index.asOfDate` function
- `Panel.getMinorXS` (use `minor_xs`) and `Panel.getMajorXS` (use `major_xs`)
- `Panel.toWide`, use `Panel.to_wide` instead

### 37.34.3 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
- Add `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
- Added column attribute access to DataFrame, e.g. `df.A` equivalent to `df["A"]` if ‘A’ is a column in the DataFrame (GH213)
- Added IPython tab completion hook for DataFrame columns. (GH233, GH230)
- Implement `Series.describe` for Series containing objects (GH241)
- Add inner join option to `DataFrame.join` when joining on key(s) (GH248)
- Can select set of DataFrame columns by passing a list to `__getitem__` (GH GH253)
- Can use `&` and `|` to intersection / union Index objects, respectively (GH 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
- Implemented `Panel.take`
- Add `set_eng_float_format` function for setting alternate DataFrame floating point string formatting
- Add convenience `set_index` function for creating a DataFrame index from its existing columns

### 37.34.4 Improvements to existing features

- Major performance improvements in file parsing functions `read_csv` and `read_table`
- Added Cython function for converting tuples to ndarray very fast. Speeds up many MultiIndex-related operations
- File parsing functions like `read_csv` and `read_table` will explicitly check if a parsed index has duplicates and raise a more helpful exception rather than deferring the check until later
- Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance (GH211)
- Improved speed of `DataFrame.xs` on mixed-type DataFrame objects by about 5x, regression from 0.3.0 (GH215)
- With new `DataFrame.align` method, speeding up binary operations between differently-indexed DataFrame objects by 10-25%.
- Significantly sped up conversion of nested dict into DataFrame (GH212)
- Can pass hierarchical index level name to `groupby` instead of the level number if desired (GH223)
- Add support for different delimiters in `DataFrame.to_csv` (GH244)
- Add more helpful error message when importing pandas post-installation from the source directory (GH250)
- Significantly speed up `DataFrame.__repr__` and `count` on large mixed-type DataFrame objects
- Better handling of pxt file dependencies in Cython module build (GH271)
37.34.5 Bug Fixes

- **read_csv / read_table fixes**
  - Be less aggressive about converting float->int in cases of floating point representations of integers like 1.0, 2.0, etc.
  - “True”/”False” will not get correctly converted to boolean
  - Index name attribute will get set when specifying an index column
  - Passing column names should force header=None (GH257)
  - Don’t modify passed column names when index_col is not None (GH258)
  - Can sniff CSV separator in zip file (since seek is not supported, was failing before)

- Worked around matplotlib “bug” in which series[:, np.newaxis] fails. Should be reported upstream to matplotlib (GH224)

- DataFrame.iteritems was not returning Series with the name attribute set. Also neither was DataFrame._series

- Can store datetime.date objects in HDFStore (GH231)

- Index and Series names are now stored in HDFStore

- Fixed problem in which data would get upcasted to object dtype in GroupBy.apply operations (GH237)

- Fixed outer join bug with empty DataFrame (GH238)

- Can create empty Panel (GH239)

- Fix join on single key when passing list with 1 entry (GH246)

- Don’t raise Exception on plotting DataFrame with an all-NA column (GH251, GH254)

- Bug min/max errors when called on integer DataFrames (GH241)

- DataFrame.iteritems and DataFrame._series not assigning name attribute

- Panel.__repr__ raised exception on length-0 major/minor axes

- DataFrame.join on key with empty DataFrame produced incorrect columns

- Implemented MultiIndex.diff (GH260)

- Int64Index.take and MultiIndex.take lost name field, fix downstream issue GH262

- Can pass list of tuples to Series (GH270)

- Can pass level name to DataFrame.stack

- Support set operations between MultiIndex and Index

- Fix many corner cases in MultiIndex set operations - Fix MultiIndex-handling bug with GroupBy.apply when returned groups are not indexed the same

- Fix corner case bugs in DataFrame.apply

- Setting DataFrame index did not cause Series cache to get cleared

- Various int32 -> int64 platform-specific issues

- Don’t be too aggressive converting to integer when parsing file with MultiIndex (GH285)

- Fix bug when slicing Series with negative indices before beginning
37.34.6 Thanks

- Thomas Kluyver
- Daniel Fortunov
- Aman Thakral
- Luca Beltrame
- Wouter Overmeire

37.35 pandas 0.4.3

Release date: 10/9/2011

is is largely a bugfix release from 0.4.2 but also includes a handful of new and enhanced features. Also, pandas can now be installed and used on Python 3 thanks Thomas Kluyver!.

37.35.1 New Features

- Python 3 support using 2to3 (GH200, Thomas Kluyver)
- Add name attribute to Series and added relevant logic and tests. Name now prints as part of Series.__repr__
- Add name attribute to standard Index so that stacking / unstacking does not discard names and so that indexed DataFrame objects can be reliably round-tripped to flat files, pickle, HDF5, etc.
- Add isnull and notnull as instance methods on Series (GH209, GH203)

37.35.2 Improvements to existing features

- Skip xlrd-related unit tests if not installed
- Index.append and MultiIndex.append can accept a list of Index objects to concatenate together
- 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)
- Refactored Series.__repr__ to be a bit more clean and consistent

37.35.3 API Changes

- Series.describe and DataFrame.describe now bring the 25% and 75% quartiles instead of the 10% and 90% deciles. The other outputs have not changed
- Series.toString will print deprecation warning, has been de-camelCased to to_string

37.35.4 Bug Fixes

- Fix broken interaction between Index and Int64Index when calling intersection. Implement Int64Index.intersection
- MultiIndex.sortlevel discarded the level names (GH202)
- Fix bugs in groupby, join, and append due to improper concatenation of MultiIndex objects (GH201)
• Fix regression from 0.4.1, isnull and notnull ceased to work on other kinds of Python scalar objects like datetime.datetime
• Raise more helpful exception when attempting to write empty DataFrame or LongPanel to HDFStore (GH204)
• Use stdlib csv module to properly escape strings with commas in DataFrame.to_csv (GH206, Thomas Kluyver)
• Fix Python ndarray access in Cython code for sparse blocked index integrity check
• Fix bug writing Series to CSV in Python 3 (GH209)
• Miscellaneous Python 3 bugfixes

37.35.5 Thanks

• Thomas Kluyver
• rsamson

37.36 pandas 0.4.2

Release date: 10/3/2011

is a performance optimization release with several bug fixes. The new t64Index and new merging / joining Cython code and related Python infrastructure are the main new additions

37.36.1 New Features

• Added fast Int64Index type with specialized join, union, intersection. Will result in significant performance enhancements for int64-based time series (e.g. using NumPy’s datetime64 one day) and also faster operations on DataFrame objects storing record array-like data.
• Refactored Index classes to have a join method and associated data alignment routines throughout the codebase to be able to leverage optimized joining / merging routines.
• Added Series.align method for aligning two series with choice of join method
• Wrote faster Cython data alignment / merging routines resulting in substantial speed increases
• Added is_monotonic property to Index classes with associated Cython code to evaluate the monotonicity of the Index values
• Add method get_level_values to MultiIndex
• Implemented shallow copy of BlockManager object in DataFrame internals

37.36.2 Improvements to existing features

• Improved performance of isnull and notnull, a regression from v0.3.0 (GH187)
• Wrote templating / code generation script to auto-generate Cython code for various functions which need to be available for the 4 major data types used in pandas (float64, bool, object, int64)
• 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
• Improved performance of DateRange.union with overlapping ranges and non-cacheable offsets (like Minute). Implemented analogous fast DateRange.intersection for overlapping ranges.

• 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

• Added informative Exception when passing dict to DataFrame groupby aggregation with axis != 0

### 37.36.3 API Changes

### 37.36.4 Bug Fixes

• Fixed minor unhandled exception in Cython code implementing fast groupby aggregation operations

• Fixed bug in unstacking code manifesting with more than 3 hierarchical levels

• Throw exception when step specified in label-based slice (GH185)

• Fix isnull to correctly work with np.float32. Fix upstream bug described in GH182

• Finish implementation of as_index=False in groupby for DataFrame aggregation (GH181)

• Raise SkipTest for pre-epoch HDFStore failure. Real fix will be sorted out via datetime64 dtype

### 37.36.5 Thanks

• Uri Laserson

• Scott Sinclair

### 37.37 pandas 0.4.1

*Release date: 9/25/2011*

is is primarily a bug fix release but includes some new features and improvements

### 37.37.1 New Features

• Added new DataFrame methods get_dtype_counts and property dtypes

• Setting of values using .ix indexing attribute in mixed-type DataFrame objects has been implemented (fixes GH135)

• read_csv can read multiple columns into a MultiIndex. DataFrame’s to_csv method will properly write out a MultiIndex which can be read back (GH151, thanks to Skipper Seabold)

• Wrote fast time series merging / joining methods in Cython. Will be integrated later into DataFrame.join and related functions

• Added ignore_index option to DataFrame.append for combining unindexed records stored in a DataFrame
37.37.2 Improvements to existing features

- Some speed enhancements with internal Index type-checking function
- `DataFrame.rename` has a new `copy` parameter which can rename a DataFrame in place
- Enable unstacking by level name (GH142)
- Enable sortlevel to work by level name (GH141)
- `read_csv` can automatically “sniff” other kinds of delimiters using `csv.Sniffer` (GH146)
- Improved speed of unit test suite by about 40%
- Exception will not be raised calling `HDFStore.remove` on non-existent node with where clause
- Optimized `_ensure_index` function resulting in performance savings in type-checking Index objects

37.37.3 API Changes

37.37.4 Bug Fixes

- Fixed DataFrame constructor bug causing downstream problems (e.g. `.copy()` failing) when passing a Series as the values along with a column name and index
- Fixed single-key groupby on DataFrame with `as_index=False` (GH160)
- `Series.shift` was failing on integer Series (GH154)
- `unstack` methods were producing incorrect output in the case of duplicate hierarchical labels. An exception will now be raised (GH147)
- Calling `count` with level argument caused reduceat failure or segfault in earlier NumPy (GH169)
- Fixed `DataFrame.corrwith` to automatically exclude non-numeric data (GH GH144)
- Unicode handling bug fixes in `DataFrame.to_string` (GH138)
- Excluding OLS degenerate unit test case that was causing platform specific failure (GH149)
- Skip blosc-dependent unit tests for PyTables < 2.2 (GH137)
- Calling `copy` on `DateRange` did not copy over attributes to the new object (GH168)
- Fix bug in `HDFStore` in which Panel data could be appended to a Table with different item order, thus resulting in an incorrect result read back

37.37.5 Thanks

- Yaroslav Halchenko
- Jeff Reback
- Skipper Seabold
- Dan Lovell
- Nick Pentreath
37.38 pandas 0.4.0

Release date: 9/12/2011

37.38.1 New Features

- *pandas.core.sparse* module: “Sparse” (mostly-NA, or some other fill value) versions of *Series*, *DataFrame*, and *Panel*. For low-density data, this will result in significant performance boosts, and smaller memory footprint. Added *to_sparse* methods to *Series*, *DataFrame*, and *Panel*. See online documentation for more on these.

- Fancy indexing operator on *Series*/*DataFrame*, e.g. via .ix operator. Both getting and setting of values is supported; however, setting values will only currently work on homogeneously-typed *DataFrame* objects. Things like:
  - series.ix[[d1, d2, d3]]
  - frame.ix[5:10, ['C', 'B', 'A']], frame.ix[5:10, 'A':'C']
  - frame.ix[date1:date2]

- Significantly enhanced *groupby* functionality
  - Can groupby multiple keys, e.g. df.groupby(['key1', 'key2']). Iteration with multiple groupings products a flattened tuple
  - “Nuisance” columns (non-aggregatable) will automatically be excluded from *DataFrame* aggregation operations
  - Added automatic “dispatching to *Series*/*DataFrame* methods to more easily invoke methods on groups. e.g. s.groupby(crit).std() will work even though *std* is not implemented on the *GroupBy* class

- Hierarchical / multi-level indexing
  - New the *MultiIndex* class. Integrated *MultiIndex* into *Series* and *DataFrame* fancy indexing, slicing, __getitem__ and __setitem__, reindexing, etc. Added *level* keyword argument to *groupby* to enable grouping by a level of a *MultiIndex*

- New data reshaping functions: *stack* and *unstack* on *DataFrame* and *Series*
  - Integrate with *MultiIndex* to enable sophisticated reshaping of data

- *Index* objects (labels for axes) are now capable of holding tuples

- *Series.describe*, *DataFrame.describe*: produces an R-like table of summary statistics about each data column

- *DataFrame.quantile*, *Series.quantile* for computing sample quantiles of data across requested axis

- Added general *DataFrame.dropna* method to replace *dropIncompleteRows* and *dropEmptyRows*, deprecated those.

- *Series* arithmetic methods with optional *fill_value* for missing data, e.g. a.add(b, fill_value=0). If a location is missing for both it will still be missing in the result though.

- *fill_value* option has been added to *DataFrame*.{add, mul, sub, div} methods similar to *Series*

- Boolean indexing with *DataFrame* objects: data[data > 0.1] = 0.1 or data[data> other] = 1.

- *pytz* / tzinfo support in *DateRange*
  - *tz_localize*, *tz_normalize*, and *tz_validate* methods added

- Added *ExcelFile* class to *pandas.io.parsers* for parsing multiple sheets out of a single Excel 2003 document
• GroupBy aggregations can now optionally broadcast, e.g. produce an object of the same size with the aggregated value propagated

• Added select function in all data structures: reindex axis based on arbitrary criterion (function returning boolean value), e.g. frame.select(lambda x: ‘foo’ in x, axis=1)

• DataFrame.consolidate method, API function relating to redesigned internals

• DataFrame.insert method for inserting column at a specified location rather than the default __setitem__ behavior (which puts it at the end)

• HDFStore class in pandas.io.pytables has been largely rewritten using patches from Jeff Reback from others. It now supports mixed-type DataFrame and Series data and can store Panel objects. It also has the option to query DataFrame and Panel data. Loading data from legacy HDFStore files is supported explicitly in the code

• Added set_printoptions method to modify appearance of DataFrame tabular output

• rolling_quantile functions; a moving version of Series.quantile / DataFrame.quantile

• Generic rolling_apply moving window function

• New drop method added to Series, DataFrame, etc. which can drop a set of labels from an axis, producing a new object

• reindex methods now sport a copy option so that data is not forced to be copied then the resulting object is indexed the same

• Added sort_index methods to Series and Panel. Renamed DataFrame.sort to sort_index. Leaving DataFrame.sort for now.

• Added skipna option to statistical instance methods on all the data structures

• pandas.io.data module providing a consistent interface for reading time series data from several different sources

37.38.2 Improvements to existing features

• The 2-dimensional DataFrame and DataMatrix classes have been extensively redesigned internally into a single class DataFrame, preserving where possible their optimal performance characteristics. This should reduce confusion from users about which class to use.

  – Note that under the hood there is a new essentially “lazy evaluation” scheme within respect to adding columns to DataFrame. During some operations, like-typed blocks will be “consolidated” but not before.

• DataFrame accessing columns repeatedly is now significantly faster than DataMatrix used to be in 0.3.0 due to an internal Series caching mechanism (which are all views on the underlying data)

• Column ordering for mixed type data is now completely consistent in DataFrame. In prior releases, there was inconsistent column ordering in DataMatrix

• Improved console / string formatting of DataMatrix with negative numbers

• Improved tabular data parsing functions, read_table and read_csv:

  – Added skiprows and na_values arguments to pandas.io.parsers functions for more flexible IO

  – parseCSV / read_csv functions and others in pandas.io.parsers now can take a list of custom NA values, and also a list of rows to skip

• Can slice DataFrame and get a view of the data (when homogeneously typed), e.g. frame.xs(idx, copy=False) or frame.ix[idx]

• Many speed optimizations throughout Series and DataFrame
• Eager evaluation of groups when calling `groupby` functions, so if there is an exception with the grouping function it will raised immediately versus sometime later on when the groups are needed

• `datetools.WeekOfMonth` offset can be parameterized with `n` different than 1 or -1.

• Statistical methods on `DataFrame` like `mean`, `std`, `var`, `skew` will now ignore non-numerical data. Before a not very useful error message was generated. A flag `numeric_only` has been added to `DataFrame.sum` and `DataFrame.count` to enable this behavior in those methods if so desired (disabled by default)

• `DataFrame.pivot` generalized to enable pivoting multiple columns into a `DataFrame` with hierarchical columns

• `DataFrame` constructor can accept structured / record arrays

• `Panel` constructor can accept a dict of `DataFrame`-like objects. Do not need to use `from_dict` anymore (`from_dict` is there to stay, though).

### 37.38.3 API Changes

• The `DataMatrix` variable now refers to `DataFrame`, will be removed within two releases

• `WidePanel` is now known as `Panel`. The `WidePanel` variable in the pandas namespace now refers to the renamed `Panel` class

• `LongPanel` and `Panel / WidePanel` now no longer have a common subclass. `LongPanel` is now a subclass of `DataFrame` having a number of additional methods and a hierarchical index instead of the old `LongPanelIndex` object, which has been removed. Legacy `LongPanel` pickles may not load properly

• Cython is now required to build pandas from a development branch. This was done to avoid continuing to check in cythonized C files into source control. Builds from released source distributions will not require Cython

• Cython code has been moved up to a top level `pandas/src` directory. Cython extension modules have been renamed and promoted from the `lib` subpackage to the top level, i.e.

  - `pandas.lib.tseries` -> `pandas._tseries`
  - `pandas.lib.sparse` -> `pandas._sparse`

• `DataFrame` pickling format has changed. Backwards compatibility for legacy pickles is provided, but it’s recommended to consider PyTables-based `HDFStore` for storing data with a longer expected shelf life

• A `copy` argument has been added to the `DataFrame` constructor to avoid unnecessary copying of data. Data is no longer copied by default when passed into the constructor

• Handling of boolean dtype in `DataFrame` has been improved to support storage of boolean data with NA / NaN values. Before it was being converted to float64 so this should not (in theory) cause API breakage

• To optimize performance, Index objects now only check that their labels are unique when uniqueness matters (i.e. when someone goes to perform a lookup). This is a potentially dangerous tradeoff, but will lead to much better performance in many places (like groupby).

• Boolean indexing using Series must now have the same indices (labels)

• Backwards compatibility support for `begin/end/nPeriods` keyword arguments in `DateRange` class has been removed

• More intuitive / shorter filling aliases `ffill` (for `pad`) and `bfill` (for `backfill`) have been added to the functions that use them: `reindex`, `asfreq`, `fillna`.

• `pandas.core.mixins` code moved to `pandas.core.generic`

• `buffer` keyword arguments (e.g. `DataFrame.toString`) renamed to `buf` to avoid using Python built-in name

• `DataFrame$rows()` removed (use `DataFrame.index`)
• Added deprecation warning to `DataFrame.cols()`, to be removed in next release
• `DataFrame` deprecations and de-camelCasing: `merge, asMatrix, toDataMatrix, _firstTimeWithValue, _lastTimeWithValue, toRecords, fromRecords, tgroupby, toString`

`pandas.io.parsers` method deprecations
  – `parseCSV` is now `read_csv` and keyword arguments have been de-camelCased
  – `parseText` is now `read_table`
  – `parseExcel` is replaced by the `ExcelFile` class and its `parse` method
• `fillna` arguments (deprecated in prior release) removed, should be replaced with `method`
• `Series.fill`, `DataFrame.fill`, and `Panel.fill` removed, use `fillna` instead
• `groupby` functions now exclude NA / NaN values from the list of groups. This matches R behavior with NAs in factors e.g. with the `tapply` function
• Removed `parseText, parseCSV` and `parseExcel` from pandas namespace
• `Series.combineFunc` renamed to `Series.combine` and made a bit more general with a `fill_value` keyword argument defaulting to NaN
• Removed `pandas.core.pytools` module. Code has been moved to `pandas.core.common`
• Tacked on `groupName` attribute for groups in GroupBy renamed to `name`
• `Panel/LongPanel dims` attribute renamed to `shape` to be more conformant
• Slicing a `Series` returns a view now
• More Series deprecations / renaming: `toCSV` to `to_csv`, `asOf` to `asof`, `merge` to `map`, `applymap` to `apply`, `toDict` to `to_dict`, `combineFirst` to `combine_first`. Will print `FutureWarning`
• `DataFrame.to_csv` does not write an “index” column label by default anymore since the output file can be read back without it. However, there is a new `index_label` argument. So you can do `index_label='index'` to emulate the old behavior
• `datetools.Week` argument renamed from `dayOfWeek` to `weekday`
• `timeRule` argument in `shift` has been deprecated in favor of using the `offset` argument for everything. So you can still pass a time rule string to `offset`
• Added optional `encoding` argument to `read_csv, read_table, to_csv, from_csv` to handle unicode in python 2.x

37.38.4 Bug Fixes

• Column ordering in `pandas.io.parsers.parseCSV` will match CSV in the presence of mixed-type data
• Fixed handling of Excel 2003 dates in `pandas.io.parsers`
• `DateRange` caching was happening with high resolution `DateOffset` objects, e.g. `DateOffset(seconds=1)`. This has been fixed
• Fixed `__truediv__` issue in `DataFrame`
• Fixed `DataFrame.toCSV` bug preventing IO round trips in some cases
• Fixed bug in `Series.plot` causing matplotlib to barf in exceptional cases
• Disabled `Index` objects from being hashable, like ndarrays
• Added `__ne__` implementation to `Index` so that operations like `ts[ts != idx]` will work
• Added `__ne__` implementation to `DataFrame`
pandas: powerful Python data analysis toolkit, Release 0.21.0

- Bug / unintuitive result when calling `fillna` on unordered labels
- Bug calling `sum` on boolean DataFrame
- Bug fix when creating a DataFrame from a dict with scalar values
- Series.{sum, mean, std, ...} now return NA/NaN when the whole Series is NA
- NumPy 1.4 through 1.6 compatibility fixes
- Fixed bug in bias correction in `rolling_cov`, was affecting `rolling_corr` too
- R-square value was incorrect in the presence of fixed and time effects in the `PanelOLS` classes
- `HDFStore` can handle duplicates in table format, will take

### 37.38.5 Thanks

- Joon Ro
- Michael Pennington
- Chris Uga
- Chris Withers
- Jeff Reback
- Ted Square
- Craig Austin
- William Ferreira
- Daniel Fortunov
- Tony Roberts
- Martin Felder
- John Marino
- Tim McNamara
- Justin Berka
- Dieter Vandenbussche
- Shane Conway
- Skipper Seabold
- Chris Jordan-Squire

### 37.39 pandas 0.3.0

**Release date:** February 20, 2011
37.39.1 New features

• *corrwith* function to compute column- or row-wise correlations between two DataFrame objects
• Can boolean-index DataFrame objects, e.g. df[df > 2] = 2, px(px > last_px] = 0
• Added comparison magic methods (__lt__, __gt__, etc.)
• Flexible explicit arithmetic methods (add, mul, sub, div, etc.)
• Added *reindex_like* method
• Added *reindex_like* method to WidePanel
• Convenience functions for accessing SQL-like databases in *pandas.io.sql* module
• Added (still experimental) HDFStore class for storing pandas data structures using HDF5 / PyTables in *pandas.io.pytables* module
• Added WeekOfMonth date offset
• *pandas.rpy* (experimental) module created, provide some interfacing / conversion between rpy2 and pandas

37.39.2 Improvements to existing features

• Unit test coverage: 100% line coverage of core data structures
• Speed enhancement to rolling_{median, max, min}
• Column ordering between DataFrame and DataMatrix is now consistent: before DataFrame would not respect column order
• Improved {Series, DataFrame}.plot methods to be more flexible (can pass matplotlib Axis arguments, plot DataFrame columns in multiple subplots, etc.)

37.39.3 API Changes

• Exponentially-weighted moment functions in *pandas.stats.moments* have a more consistent API and accept a min_periods argument like their regular moving counterparts.
• *fillMethod* argument in Series, DataFrame changed to *method*, *FutureWarning* added.
• *fill* method in Series, DataFrame/DataMatrix, WidePanel renamed to *fillna*, *FutureWarning* added to *fill*
• Renamed *DataFrame.getXS* to *xs*, *FutureWarning* added
• Removed *cap* and *floor* functions from DataFrame, renamed to *clip_upper* and *clip_lower* for consistency with NumPy

37.39.4 Bug Fixes

• Fixed bug in IndexableSkiplist Cython code that was breaking rolling_max function
• Numerous numpy.int64-related indexing fixes
• Several NumPy 1.4.0 NaN-handling fixes
• Bug fixes to pandas.io.parsers.parseCSV
• Fixed *DateRange* caching issue with unusual date offsets
• Fixed bug in *DateRange.union*
• Fixed corner case in IndexableSkipList implementation
BIBLIOGRAPHY

[R9] https://docs.python.org/3/library/pickle.html
[R5] https://docs.python.org/3/library/pickle.html
pandas, 1