pandas: powerful Python data analysis toolkit

Release 0.16.2

Wes McKinney & PyData Development Team

June 13, 2015
1 What's New
   1.1 v0.16.2 (June 12, 2015) ................................................................. 3
   1.2 v0.16.1 (May 11, 2015) ................................................................. 6
   1.3 v0.16.0 (March 22, 2015) ............................................................. 17
   1.4 v0.15.2 (December 12, 2014) ....................................................... 33
   1.5 v0.15.1 (November 9, 2014) ......................................................... 38
   1.6 v0.15.0 (October 18, 2014) ........................................................... 44
   1.7 v0.14.1 (July 11, 2014) ................................................................. 71
   1.8 v0.14.0 (May 31, 2014) ................................................................. 77
   1.9 v0.13.1 (February 3, 2014) ............................................................. 104
   1.10 v0.13.0 (January 3, 2014) ............................................................. 111
   1.11 v0.12.0 (July 24, 2013) ............................................................... 134
   1.12 v0.11.0 (April 22, 2013) ............................................................... 145
   1.13 v0.10.1 (January 22, 2013) ........................................................... 154
   1.14 v0.10.0 (December 17, 2012) ....................................................... 160
   1.15 v0.9.1 (November 14, 2012) ....................................................... 171
   1.16 v0.9.0 (October 7, 2012) ............................................................. 175
   1.17 v0.8.1 (July 22, 2012) ................................................................. 177
   1.18 v0.8.0 (June 29, 2012) ................................................................. 177
   1.19 v0.7.3 (April 12, 2012) ................................................................. 182
   1.20 v0.7.2 (March 16, 2012) ............................................................... 186
   1.21 v0.7.1 (February 29, 2012) ........................................................... 187
   1.22 v0.7.0 (February 9, 2012) ............................................................ 187
   1.23 v0.6.1 (December 13, 2011) ......................................................... 192
   1.24 v0.6.0 (November 25, 2011) ....................................................... 193
   1.25 v0.5.0 (October 24, 2011) ........................................................... 195
   1.26 v0.4.3 through v0.4.1 (September 25 - October 9, 2011) ............... 196

2 Installation ................................................................. 199
   2.1 Python version support ............................................................... 199
   2.2 Installing pandas ................................................................. 199
   2.3 Dependencies ................................................................. 202

3 Contributing to pandas .................................................... 205
   3.1 Where to start? ............................................................... 205
   3.2 Bug Reports/Enhancement Requests ........................................... 206
   3.3 Working with the code .......................................................... 206
   3.4 Contributing to the documentation ........................................... 209
   3.5 Contributing to the code base ............................................... 211
   3.6 Contributing your changes to pandas ...................................... 213
14 MultiIndex / Advanced Indexing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Hierarchical indexing (MultiIndex)</td>
<td>425</td>
</tr>
<tr>
<td>14.2 Advanced indexing with hierarchical index</td>
<td>430</td>
</tr>
<tr>
<td>14.3 The need for sortedness with MultiIndex</td>
<td>439</td>
</tr>
<tr>
<td>14.4 Take Methods</td>
<td>441</td>
</tr>
<tr>
<td>14.5 CategoricalIndex</td>
<td>443</td>
</tr>
<tr>
<td>14.6 Float64Index</td>
<td>445</td>
</tr>
</tbody>
</table>

15 Computational tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1 Statistical functions</td>
<td>449</td>
</tr>
<tr>
<td>15.2 Moving (rolling) statistics / moments</td>
<td>453</td>
</tr>
<tr>
<td>15.3 Expanding window moment functions</td>
<td>461</td>
</tr>
<tr>
<td>15.4 Exponentially weighted moment functions</td>
<td>462</td>
</tr>
</tbody>
</table>

16 Working with missing data

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Missing data basics</td>
<td>467</td>
</tr>
<tr>
<td>16.2 Datetimes</td>
<td>469</td>
</tr>
<tr>
<td>16.3 Inserting missing data</td>
<td>469</td>
</tr>
<tr>
<td>16.4 Calculations with missing data</td>
<td>470</td>
</tr>
<tr>
<td>16.5 Cleaning / filling missing data</td>
<td>471</td>
</tr>
<tr>
<td>16.6 Missing data casting rules and indexing</td>
<td>484</td>
</tr>
</tbody>
</table>

17 Group By: split-apply-combine

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 Splitting an object into groups</td>
<td>487</td>
</tr>
<tr>
<td>17.2 Iterating through groups</td>
<td>492</td>
</tr>
<tr>
<td>17.3 Selecting a group</td>
<td>493</td>
</tr>
<tr>
<td>17.4 Aggregation</td>
<td>493</td>
</tr>
<tr>
<td>17.5 Transformation</td>
<td>497</td>
</tr>
<tr>
<td>17.6 Filtration</td>
<td>501</td>
</tr>
<tr>
<td>17.7 Dispatching to instance methods</td>
<td>502</td>
</tr>
<tr>
<td>17.8 Flexible apply</td>
<td>504</td>
</tr>
<tr>
<td>17.9 Other useful features</td>
<td>506</td>
</tr>
<tr>
<td>17.10 Examples</td>
<td>513</td>
</tr>
</tbody>
</table>

18 Merge, join, and concatenate

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Concatenating objects</td>
<td>515</td>
</tr>
<tr>
<td>18.2 Database-style DataFrame joining/merging</td>
<td>526</td>
</tr>
</tbody>
</table>

19 Reshaping and Pivot Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Reshaping by pivoting DataFrame objects</td>
<td>539</td>
</tr>
<tr>
<td>19.2 Reshaping by stacking and unstacking</td>
<td>540</td>
</tr>
<tr>
<td>19.3 Reshaping by Melt</td>
<td>545</td>
</tr>
<tr>
<td>19.4 Combining with stats and GroupBy</td>
<td>546</td>
</tr>
<tr>
<td>19.5 Pivot tables and cross-tabulations</td>
<td>547</td>
</tr>
<tr>
<td>19.6 Tiling</td>
<td>551</td>
</tr>
<tr>
<td>19.7 Computing indicator / dummy variables</td>
<td>551</td>
</tr>
<tr>
<td>19.8 Factorizing values</td>
<td>554</td>
</tr>
</tbody>
</table>

20 Time Series / Date functionality

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Time Stamps vs. Time Spans</td>
<td>555</td>
</tr>
<tr>
<td>20.2 Converting to Timestamps</td>
<td>556</td>
</tr>
<tr>
<td>20.3 Generating Ranges of Timestamps</td>
<td>557</td>
</tr>
<tr>
<td>20.4 DatetimeIndex</td>
<td>558</td>
</tr>
<tr>
<td>20.5 Datetimeoffset objects</td>
<td>560</td>
</tr>
<tr>
<td>20.6 Time series-related instance methods</td>
<td>567</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>20.7 Up- and downsampling</td>
<td>579</td>
</tr>
<tr>
<td>20.8 Time Span Representation</td>
<td>581</td>
</tr>
<tr>
<td>20.9 Converting between Representations</td>
<td>587</td>
</tr>
<tr>
<td>20.10 Representing out-of-bounds spans</td>
<td>588</td>
</tr>
<tr>
<td>20.11 Time Zone Handling</td>
<td>589</td>
</tr>
<tr>
<td>21 Time Deltas</td>
<td>597</td>
</tr>
<tr>
<td>21.1 Parsing</td>
<td>597</td>
</tr>
<tr>
<td>21.2 Operations</td>
<td>598</td>
</tr>
<tr>
<td>21.3 Reductions</td>
<td>602</td>
</tr>
<tr>
<td>21.4 Frequency Conversion</td>
<td>603</td>
</tr>
<tr>
<td>21.5 Attributes</td>
<td>604</td>
</tr>
<tr>
<td>21.6 TimedeltaIndex</td>
<td>605</td>
</tr>
<tr>
<td>21.7 Resampling</td>
<td>608</td>
</tr>
<tr>
<td>22 Categorical Data</td>
<td>609</td>
</tr>
<tr>
<td>22.1 Object Creation</td>
<td>609</td>
</tr>
<tr>
<td>22.2 Description</td>
<td>612</td>
</tr>
<tr>
<td>22.3 Working with categories</td>
<td>613</td>
</tr>
<tr>
<td>22.4 Sorting and Order</td>
<td>616</td>
</tr>
<tr>
<td>22.5 Comparisons</td>
<td>618</td>
</tr>
<tr>
<td>22.6 Operations</td>
<td>620</td>
</tr>
<tr>
<td>22.7 Data munging</td>
<td>621</td>
</tr>
<tr>
<td>22.8 Getting Data In/Out</td>
<td>625</td>
</tr>
<tr>
<td>22.9 Missing Data</td>
<td>626</td>
</tr>
<tr>
<td>22.10 Gotchas</td>
<td>628</td>
</tr>
<tr>
<td>23 Plotting</td>
<td>633</td>
</tr>
<tr>
<td>23.1 Basic Plotting: <code>plot</code></td>
<td>633</td>
</tr>
<tr>
<td>23.2 Other Plots</td>
<td>636</td>
</tr>
<tr>
<td>23.3 Plotting with Missing Data</td>
<td>667</td>
</tr>
<tr>
<td>23.4 Plotting Tools</td>
<td>668</td>
</tr>
<tr>
<td>23.5 Plot Formatting</td>
<td>676</td>
</tr>
<tr>
<td>23.6 Plotting directly with matplotlib</td>
<td>699</td>
</tr>
<tr>
<td>23.7 Trellis plotting interface</td>
<td>700</td>
</tr>
<tr>
<td>24 IO Tools (Text, CSV, HDF5, ...)</td>
<td>715</td>
</tr>
<tr>
<td>24.1 CSV &amp; Text files</td>
<td>716</td>
</tr>
<tr>
<td>24.2 JSON</td>
<td>739</td>
</tr>
<tr>
<td>24.3 HTML</td>
<td>747</td>
</tr>
<tr>
<td>24.4 Excel files</td>
<td>755</td>
</tr>
<tr>
<td>24.5 Clipboard</td>
<td>758</td>
</tr>
<tr>
<td>24.6 Pickling</td>
<td>759</td>
</tr>
<tr>
<td>24.7 msgpack (experimental)</td>
<td>760</td>
</tr>
<tr>
<td>24.8 HDF5 (PyTables)</td>
<td>762</td>
</tr>
<tr>
<td>24.9 SQL Queries</td>
<td>789</td>
</tr>
<tr>
<td>24.10 Google BigQuery (Experimental)</td>
<td>797</td>
</tr>
<tr>
<td>24.11 Stata Format</td>
<td>798</td>
</tr>
<tr>
<td>24.12 Other file formats</td>
<td>800</td>
</tr>
<tr>
<td>24.13 Performance Considerations</td>
<td>801</td>
</tr>
<tr>
<td>25 Remote Data Access</td>
<td>803</td>
</tr>
<tr>
<td>25.1 Yahoo! Finance</td>
<td>803</td>
</tr>
<tr>
<td>25.2 Yahoo! Finance Options</td>
<td>804</td>
</tr>
<tr>
<td>25.3 Google Finance</td>
<td>806</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>32.3</td>
<td>GROUP BY</td>
</tr>
<tr>
<td>32.4</td>
<td>JOIN</td>
</tr>
<tr>
<td>32.5</td>
<td>UNION</td>
</tr>
<tr>
<td>32.6</td>
<td>UPDATE</td>
</tr>
<tr>
<td>32.7</td>
<td>DELETE</td>
</tr>
<tr>
<td>33</td>
<td>API Reference</td>
</tr>
<tr>
<td>33.1</td>
<td>Input/Output</td>
</tr>
<tr>
<td>33.2</td>
<td>General functions</td>
</tr>
<tr>
<td>33.3</td>
<td>Series</td>
</tr>
<tr>
<td>33.4</td>
<td>DataFrame</td>
</tr>
<tr>
<td>33.5</td>
<td>Panel</td>
</tr>
<tr>
<td>33.6</td>
<td>Panel4D</td>
</tr>
<tr>
<td>33.7</td>
<td>Index</td>
</tr>
<tr>
<td>33.8</td>
<td>CategoricalIndex</td>
</tr>
<tr>
<td>33.9</td>
<td>DatetimeIndex</td>
</tr>
<tr>
<td>33.10</td>
<td>TimeDeltaIndex</td>
</tr>
<tr>
<td>33.11</td>
<td>GroupBy</td>
</tr>
<tr>
<td>33.12</td>
<td>General utility functions</td>
</tr>
<tr>
<td>34</td>
<td>Internals</td>
</tr>
<tr>
<td>34.1</td>
<td>Indexing</td>
</tr>
<tr>
<td>34.2</td>
<td>Subclassing pandas Data Structures</td>
</tr>
<tr>
<td>35</td>
<td>Release Notes</td>
</tr>
<tr>
<td>35.1</td>
<td>pandas 0.16.2</td>
</tr>
<tr>
<td>35.2</td>
<td>pandas 0.16.1</td>
</tr>
<tr>
<td>35.3</td>
<td>pandas 0.16.0</td>
</tr>
<tr>
<td>35.4</td>
<td>pandas 0.15.2</td>
</tr>
<tr>
<td>35.5</td>
<td>pandas 0.15.1</td>
</tr>
<tr>
<td>35.6</td>
<td>pandas 0.15.0</td>
</tr>
<tr>
<td>35.7</td>
<td>pandas 0.14.1</td>
</tr>
<tr>
<td>35.8</td>
<td>pandas 0.14.0</td>
</tr>
<tr>
<td>35.9</td>
<td>pandas 0.13.1</td>
</tr>
<tr>
<td>35.10</td>
<td>pandas 0.13.0</td>
</tr>
<tr>
<td>35.11</td>
<td>pandas 0.12.0</td>
</tr>
<tr>
<td>35.12</td>
<td>pandas 0.11.0</td>
</tr>
<tr>
<td>35.13</td>
<td>pandas 0.10.1</td>
</tr>
<tr>
<td>35.14</td>
<td>pandas 0.10.0</td>
</tr>
<tr>
<td>35.15</td>
<td>pandas 0.9.1</td>
</tr>
<tr>
<td>35.16</td>
<td>pandas 0.9.0</td>
</tr>
<tr>
<td>35.17</td>
<td>pandas 0.8.1</td>
</tr>
<tr>
<td>35.18</td>
<td>pandas 0.8.0</td>
</tr>
<tr>
<td>35.19</td>
<td>pandas 0.7.3</td>
</tr>
<tr>
<td>35.20</td>
<td>pandas 0.7.2</td>
</tr>
<tr>
<td>35.21</td>
<td>pandas 0.7.1</td>
</tr>
<tr>
<td>35.22</td>
<td>pandas 0.7.0</td>
</tr>
<tr>
<td>35.23</td>
<td>pandas 0.6.1</td>
</tr>
<tr>
<td>35.24</td>
<td>pandas 0.6.0</td>
</tr>
<tr>
<td>35.25</td>
<td>pandas 0.5.0</td>
</tr>
<tr>
<td>35.26</td>
<td>pandas 0.4.3</td>
</tr>
<tr>
<td>35.27</td>
<td>pandas 0.4.2</td>
</tr>
<tr>
<td>35.28</td>
<td>pandas 0.4.1</td>
</tr>
<tr>
<td>35.29</td>
<td>pandas 0.4.0</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.
WHAT’S NEW

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

1.1 v0.16.2 (June 12, 2015)

This is a minor bug-fix release from 0.16.1 and includes a large number of bug fixes along some new features (pipe() method), enhancements, and performance improvements.

We recommend that all users upgrade to this version.

Highlights include:

- A new pipe method, see here
- Documentation on how to use numba with pandas, see here

What’s new in v0.16.2

- New features
  - Pipe
  - Other Enhancements
- API Changes
- Performance Improvements
- Bug Fixes

1.1.1 New features

Pipe

We’ve introduced a new method DataFrame.pipe(). As suggested by the name, pipe should be used to pipe data through a chain of function calls. The goal is to avoid confusing nested function calls like

```python
# df is a DataFrame
# f, g, and h are functions that take and return DataFrames
f(g(h(df), arg1=1), arg2=2, arg3=3)
```

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

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

```
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: Sat, 13 Jun 2015 Pseudo R-squ.: 0.6878
Time: 15:15:58 Log-Likelihood: -143.91
converged: True LL-Null: -460.91
LLR p-value: 6.774e-136
==============================================================================
coef std err z P>|z| [95.0% Conf. Int.]
-------------------------------------------------------------------------------
Intercept -1267.3636 457.867 -2.768 0.006 -2164.767 -369.960
C(lg)[T.NL] -0.2057 0.101 -2.044 0.041 -0.403 -0.008
ln_h 0.9280 0.191 4.866 0.000 0.554 1.302
year 0.6301 0.228 2.762 0.006 0.183 1.077
g 0.0099 0.004 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)
```

Other Enhancements

- Added rsplit to Index/Series StringMethods (GH10303)
• Removed the hard-coded size limits on the DataFrame HTML representation in the IPython notebook, and leave this to IPython itself (only for IPython v3.0 or greater). This eliminates the duplicate scroll bars that appeared in the notebook with large frames (GH10231).

Note that the notebook has a toggle output scrolling feature to limit the display of very large frames (by clicking left of the output). You can also configure the way DataFrames are displayed using the pandas options, see here.

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

1.1.2 API Changes

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

1.1.3 Performance Improvements

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

1.1.4 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

• 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 datetim64 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).

1.2 v0.16.1 (May 11, 2015)

This is a minor bug-fix release from 0.16.0 and includes a 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 un-
derstand 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` datatime-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)

### 1.2.1 Enhancements

#### CategoricalIndex

We introduce a `CategoricalIndex`, a new type of index object that is useful for supporting indexing with dupli-
cates. 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`.

```python
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    int32
```

1.2. v0.16.1 (May 11, 2015)
**pandas: powerful Python data analysis toolkit, Release 0.16.2**

B category
dtype: object

In [4]: df.B.cat.categories
Out[4]: Index([u'c', u'a', u'b'], dtype='object')

setting the index, will create create a CategoricalIndex

In [5]: df2 = df.set_index('B')

In [6]: df2.index
Out[6]: CategoricalIndex([u'a', u'a', u'b', u'b', u'c', u'a'], categories=[u'c', u'a', u'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]:
   B
a 0
a 1
a 5

and preserves the CategoricalIndex

In [8]: df2.loc['a'].index
Out[8]: CategoricalIndex([u'a', u'a', u'a'], categories=[u'c', u'a', u'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 5
   a 1
   b 2
   b 3

groupby operations on the index will preserve the index nature as well

In [10]: df2.groupby(level=0).sum()
Out[10]:
   A
B
   c 4
   a 6
   b 5

In [11]: df2.groupby(level=0).sum().index
Out[11]: CategoricalIndex([u'c', u'a', u'b'], categories=[u'c', u'a', u'b'], ordered=False, name='B', dtype='category')

reindexing operations, will return a resulting index based on the type of the passed indexer, meaning that passing a list will return a plain-old-Index; indexing with a Categorical will return a CategoricalIndex, indexed according to the categories of the PASSED Categorical dtype. This allows one to arbitrarily index these even with values NOT in the categories, similarly to how you can reindex ANY pandas index.
In [12]: df2.reindex(['a','e'])
Out[12]:
   A  B
  a 0  a
  a 1
  a 5
  e NaN

In [13]: df2.reindex(['a','e']).index
Out[13]: Index([u'a', u'a', u'a', u'e'], dtype='object', name=u'B')

In [14]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde')))
Out[14]:
   A  B
  a 0  a
  a 1
  a 5
  e NaN

In [15]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde'))).index
Out[15]: CategoricalIndex([u'a', u'a', u'a', u'e'], categories=[u'a', u'b', u'c', u'd', u'e'], ordered=False, name=u'B', dtype='category')

See the documentation for more. (GH7629, GH10038, GH10039)

**Sample**

Series, DataFrames, and Panels now have a new method: `.sample()`. The method accepts a specific number of rows or columns to return, or a fraction of the total number 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 [16]: example_series = Series([0,1,2,3,4,5])

# When no arguments are passed, returns 1
In [17]: example_series.sample()
Out[17]:
   0
0 0
dtype: int64

# One may specify either a number of rows:
In [18]: example_series.sample(n=3)
Out[18]:
   4  3  5
4 4 3 5
dtype: int64

# Or a fraction of the rows:
In [19]: example_series.sample(frac=0.5)
Out[19]:
   2  3  1
2 2 3 1
dtype: int64
# weights are accepted.
In [20]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [21]: example_series.sample(n=3, weights=example_weights)
Out[21]:
2 2
5 5
3 3
dtype: int64

# weights will also be normalized if they do not sum to one,
# and missing values will be treated as zeros.
In [22]: example_weights2 = [0.5, 0, 0, 0, None, np.nan]

In [23]: example_series.sample(n=1, weights=example_weights2)
Out[23]:
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 [24]: df = DataFrame({'col1':[9,8,7,6], 'weight_column':[0.5, 0.4, 0.1, 0]})

In [25]: df.sample(n=3, weights='weight_column')
Out[25]:
col1 weight_column
1 8 0.4
0 9 0.5
2 7 0.1

String Methods Enhancements

Continuing from v0.16.0, the following enhancements make string operations easier and more consistent with standard python string operations.

- **Added** StringMethods (.str accessor) to Index (GH9068)

  The .str accessor is now available for both Series and Index.

  In [26]: idx = Index([' jack', 'jill ', ' jesse ', 'frank'])

  In [27]: idx.str.strip()
  Out[27]: Index([u'jack', u'jill', u'jesse', u'frank'], dtype='object')

One special case for the .str accessor on Index is that if a string method returns bool, the .str accessor will return a np.array instead of a boolean Index (GH8875). This enables the following expression to work naturally:

In [28]: idx = Index(['a1', 'a2', 'b1', 'b2'])

In [29]: s = Series(range(4), index=idx)

In [30]: s
Out[30]:
a1 0
a2 1
b1 2
b2  3
dtype: int64

In [31]: idx.str.startswith('a')
Out[31]: array([ True, True, False, False], dtype=bool)

In [32]: s[s.index.str.startswith('a')]
Out[32]:
  a1  0
  a2  1
dtype: int64

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

<table>
<thead>
<tr>
<th>Methods</th>
<th>Capitalize()</th>
<th>Swapcase()</th>
<th>Normalize()</th>
<th>Partition()</th>
<th>Rpartition()</th>
</tr>
</thead>
<tbody>
<tr>
<td>index()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rindex()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>translate()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• split now takes expand keyword to specify whether to expand dimensionality. return_type is deprecated. (GH9847)

In [33]: s = Series(['a,b', 'a,c', 'b,c'])

# return Series
In [34]: s.str.split(','
Out[34]:
   0  [a, b]
   1  [a, c]
   2  [b, c]
dtype: object

# return DataFrame
In [35]: s.str.split(',', expand=True)
Out[35]:
   0  a  b
   1  a  c
   2  b  c

In [36]: idx = Index(['a,b', 'a,c', 'b,c'])

# return Index
In [37]: idx.str.split(',')
Out[37]: Index([['a', 'b'], ['a', 'c'], ['b', 'c']], dtype='object')

# return MultiIndex
In [38]: idx.str.split(',', expand=True)
Out[38]:
   MultiIndex(levels=[[u'a', u'b'], [u'a', u'c'], [u'b', u'c']], labels=[[0, 0, 1], [0, 1, 1]])

• Improved extract and get_dummies methods for Index.str (GH9980)

Other Enhancements

• BusinessHour offset is now supported, which represents business hours starting from 09:00 - 17:00 on BusinessDay by default. See Here for details. (GH7905)
In [39]: from pandas.tseries.offsets import BusinessHour

In [40]: Timestamp('2014-08-01 09:00') + BusinessHour()
Out[40]: Timestamp('2014-08-01 10:00:00')

In [41]: Timestamp('2014-08-01 07:00') + BusinessHour()
Out[41]: Timestamp('2014-08-01 10:00:00')

In [42]: Timestamp('2014-08-01 16:30') + BusinessHour()
Out[42]: Timestamp('2014-08-04 09:30:00')

• DataFrame.diff now takes an axis parameter that determines the direction of differencing (GH9727)
• Allow clip, clip_lower, and clip_upper to accept array-like arguments as thresholds (This is a regression from 0.11.0). These methods now have an axis parameter which determines how the Series or DataFrame will be aligned with the threshold(s). (GH6966)
• DataFrame.mask() and Series.mask() now support same keywords as where (GH8801)
• drop function can now accept errors keyword to suppress ValueError raised when any of label does not exist in the target data. (GH6736)

In [43]: df = DataFrame(np.random.randn(3, 3), columns=['A', 'B', 'C'])

In [44]: df.drop(['A', 'X'], axis=1, errors='ignore')
Out[44]:
   B  C
0 -0.064034 -1.282782
1 -1.071357  0.441153
2  0.583787  0.221471

• 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)
1.2.2 API changes

- When passing in an ax to `df.plot(...)`, the `sharex` kwarg will now default to `False`. The result is that the visibility of xlabels and xticklabels will not anymore be changed. You have to do that by yourself for the right axes in your figure or set `sharex=True` explicitly (but this changes the visible for all axes in the figure, not only the one which is passed in!). If pandas creates the subplots itself (e.g. no passed in `ax` kwarg), then the default is still `sharex=True` and the visibility changes are applied.

- `assign()` now inserts new columns in alphabetical order. Previously the order was arbitrary. (GH9777)

- By default, `read_csv` and `read_table` will now try to infer the compression type based on the file extension. Set `compression=None` to restore the previous behavior (no decompression). (GH9770)

Deprecations

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

1.2.3 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, ...

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 [45]: pd.set_option('display.width', 80)

In [46]: pd.Index(range(4), name='foo')
Out[46]: Int64Index([0, 1, 2, 3], dtype='int64', name='foo')

In [47]: pd.Index(range(30), name='foo')
Out[47]:
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],
            dtype='int64', name='foo')
```
In [48]: pd.Index(range(104), name='foo')
Out[48]:
Int64Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 
94, 95, 96, 97, 98, 99, 100, 101, 102, 103],
dtype='int64', name='foo', length=104)

In [49]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar')
Out[49]: CategoricalIndex([u'a', u'bb', u'ccc', u'dddd'], categories=[u'a', u'bb', u'ccc', u'dddd'], ordered=True, name='foobar', dtype='category')

In [50]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd']*10, ordered=True, name='foobar')
Out[50]: CategoricalIndex([u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', 
categories=[u'a', u'bb', u'ccc', u'dddd'], ordered=True, name='foobar', dtype='category')

In [51]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd']*100, ordered=True, name='foobar')
Out[51]: CategoricalIndex([u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', 
... u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd', u'a', u'bb', u'ccc', u'dddd'], categories=[u'a', u'bb', u'ccc', u'dddd'], ordered=True, name='foobar', dtype='category')

In [52]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')
Out[52]: 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]', name='foo', freq='D', tz='US/Eastern')

In [53]: pd.date_range('20130101', periods=25, freq='D')

In [54]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')
Out[54]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00', '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00', '2013-01-05 00:00:00-05:00', '2013-01-06 00:00:00-05:00', '2013-01-07 00:00:00-05:00', '2013-01-08 00:00:00-05:00', '2013-01-09 00:00:00-05:00', '2013-01-10 00:00:00-05:00', 
... '2013-04-05 00:00:00-04:00', '2013-04-06 00:00:00-04:00', '2013-04-07 00:00:00-04:00', '2013-04-08 00:00:00-04:00', '2013-04-09 00:00:00-04:00', '2013-04-10 00:00:00-04:00', '2013-04-11 00:00:00-04:00', '2013-04-12 00:00:00-04:00', '2013-04-13 00:00:00-04:00', '2013-04-14 00:00:00-04:00'],
1.2.4 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)

1.2.5 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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(["abc"], ordered=True)) > "d"`). This returned False for all elements, but now raises a `TypeError`. Equality comparisons also now return `False` for `==` and `True` for `!=`. (GH9848)
- Bug in `DataFrame __setitem__` when right hand side is a dictionary (GH9874)
- Bug in `where` when `dtype` is `datetime64/timedelta64`, but `dtype` of other is not (GH9804)
- Bug in `MultiIndex.sortlevel()` results in unicode level name breaks (GH9856)
- Bug in which `groupby.transform` incorrectly enforced output dtypes to match input dtypes. (GH9807)
- Bug in `DataFrame constructor` when `columns` parameter is set, and `data` is an empty list (GH9939)
- Bug in bar plot with `log=True` raises `TypeError` if all values are less than 1 (GH9905)
- Bug in horizontal bar plot ignores `log=True` (GH9905)
- Bug in PyTables queries that did not return proper results using the index (GH8265, GH9676)
- Bug where dividing a dataframe containing values of type `Decimal` by another `Decimal` would raise. (GH9787)
- Bug where using DataFrames asfreq would remove the name of the index. (GH9885)
- Bug causing extra index point when resample BM/BQ (GH9756)
- Changed caching in `AbstractHolidayCalendar` to be at the instance level rather than at the class level as the latter can result in unexpected behaviour. (GH9552)
• Fixed latex output for 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)

1.3 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**

1.3.1 New features

**DataFrame Assign**

Inspired by dplyr's `mutate` verb, DataFrame has a new `assign()` method. The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. The new values are inserted, and the entire DataFrame (with all original and new columns) is returned.

```
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

```python
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'))
```

See the documentation for more. (GH9229)

**Interaction with scipy.sparse**

Added `SparseSeries.to_coo()` and `SparseSeries.from_coo()` methods (GH8048) for converting to and from `scipy.sparse.coo_matrix` instances (see [here](#)). For example, given a SparseSeries with MultiIndex we can convert to a `scipy.sparse.coo_matrix` by specifying the row and column labels as index levels:

```python
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
```

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
| 1 | 2 | a | 0 | 3
|   |   |   | 1 | NaN
| 1 | b | 0 | 1|
```
1 3
2 1 b 0 NaN
1 NaN
dtype: float64

# SparseSeries
In [10]: ss = s.to_sparse()

In [11]: ss
Out[11]:
A B C D
1 2 a 0 3
1 NaN
1 b 0 1
1 3
2 1 b 0 NaN
1 NaN
dtype: float64

BlockIndex
Block locations: array([0, 2])
Block lengths: array([1, 2])

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 '<type 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [14]: A.todense()
Out[14]:
matrix([[ 3., 0., 0., 0.],
[ 0., 0., 1., 3.],
[ 0., 0., 0., 0.]])

In [15]: rows
Out[15]: [(1L, 2L), (1L, 1L), (2L, 1L)]

In [16]: columns
Out[16]: [('a', 0L), ('a', 1L), ('b', 0L), ('b', 1L)]

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 '<type 'numpy.float64'>'
with 3 stored elements in COOrdinate format>
In [20]: A.todense()
Out[20]:
matrix([[ 0., 0., 1., 2.],
        [ 3., 0., 0., 0.],
        [ 0., 0., 0., 0.]])

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

In [22]: ss
Out[22]:
0 2 1
 3 2
1 0 3
dtype: float64
BlockIndex
Block locations: array([0])
Block lengths: array([3])

String Methods Enhancements

• Following new methods are accesible via .str accessor to apply the function to each values. This is intended to make it more consistent with standard methods on strings. (GH9282, GH9352, GH9386, GH9387, GH9439)

<table>
<thead>
<tr>
<th>Method</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum()</td>
<td>isalpha() isdigit() isdigit() isspace() isdecimal()</td>
</tr>
<tr>
<td>islower()</td>
<td>isupper() istitle() isnumeric()</td>
</tr>
<tr>
<td>find()</td>
<td>rfind()  ljust()  rjust()  zfill()</td>
</tr>
</tbody>
</table>

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

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

In [25]: s.str.find('ab')
Out[25]:
0   0
1  -1
2  -1
dtype: int64

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

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

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

• Added Series.str.slice_replace(), which previously raised NotImplementedError (GH8888)
In [28]: s = Series(['ABCD', 'EFGH', 'IJK'])

In [29]: s.str.slice_replace(1, 3, 'X')
Out[29]:
   0 AXD  
   1 EXH  
   2 IX   
   dtype: object

# replaced with empty char
In [30]: s.str.slice_replace(0, 1)
Out[30]:
   0 BCD  
   1 FGH  
   2 JK   
   dtype: object

Other enhancements

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

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)

   # Returns the 1st and 4th sheet, as a dictionary of DataFrames.
   pd.read_excel('path_to_file.xls',sheetname=['Sheet1',3])

• Allow Stata files to be read incrementally with an iterator; support for long strings in Stata files. See the docs here (GH9493:).

• Paths beginning with ~ will now be expanded to begin with the user’s home directory (GH9066)

• Added time interval selection in get_data_yahoo (GH9071)

• Added Timestamp.to_datetime64() to complement Timedelta.to_timedelta64() (GH9255)

• tseries.frequencies.to_offset() now accepts Timedelta as input (GH9064)

• Lag parameter was added to the autocorrelation method of Series, defaults to lag-1 autocorrelation (GH9192)

• Timedelta will now accept nanoseconds keyword in constructor (GH9273)

• SQL code now safely escapes table and column names (GH8986)

• Added auto-complete for Series.str.<tab>, Series.dt.<tab> and Series.cat.<tab> (GH9322)
• **Index.get_indexer** now supports method=’pad’ and method=’backfill’ even for any target array, not just monotonic targets. These methods also work for monotonic decreasing as well as monotonic increasing indexes (GH9258).

• **Index.asof** now works on all index types (GH9258).

• A **verbose** argument has been augmented in io.read_excel(), defaults to False. Set to True to print sheet names as they are parsed. (GH9450)

• Added **days_in_month** (compatibility alias daysinmonth) property to Timestamp, DatetimeIndex, Period, PeriodIndex, and Series.dt (GH9572)

• Added **decimal** option in to_csv to provide formatting for non-’.’ decimal separators (GH781)

• Added **normalize** option for Timestamp to normalized to midnight (GH8794)

• Added example for DataFrame import to R using HDF5 file and rhdf5 library. See the documentation for more (GH9636).

### 1.3.2 Backwards incompatible API changes

#### Changes in Timedelta

In v0.15.0 a new scalar type Timedelta was introduced, that is a sub-class of datetime.timedelta. Mentioned here 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**

In [2]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [3]: t.days
Out[3]: 1

In [4]: t.seconds
Out[4]: 12

In [5]: t.microseconds
Out[5]: 123

**New Behavior**

In [33]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [34]: t.days
Out[34]: 1L

In [35]: t.seconds
Out[35]: 36672L

In [36]: t.microseconds
Out[36]: 100123L
Using `.components` allows the full component access

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

```python
In [38]: t.components.seconds
Out[38]: 12L
```

**Indexing Changes**

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

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

```python
In [39]: df = DataFrame(np.random.randn(5,4),
       ...:             columns=list('ABCD'),
       ...:             index=date_range('20130101',periods=5))
       ...
In [40]: df
```

```plaintext
A   B   C   D
2013-01-01 -0.744471 0.758527 1.729689 -0.964980
2013-01-02 -0.845696 -1.340896 1.846883 -1.328865
2013-01-03 1.682706 -1.717693 0.888782 0.228440
2013-01-04 0.901805 1.171216 0.520260 -1.197071
2013-01-05 -1.066969 -0.303421 -0.858447 0.306996
```

```python
In [41]: s = Series(range(5),[-2,-1,1,2,3])
In [42]: s
```

```plaintext
-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]'
```

```python
In [6]: s.loc[-10:3]
KeyError: 'start bound [-10] is not the [index]'
```

**New Behavior**

```python
In [43]: df.loc['2013-01-02':'2013-01-10']
Out[43]:
       A        B         C         D
2013-01-02 -0.845696 -1.340896  1.846883 -1.328865
2013-01-03  1.682706 -1.717693  0.888782  0.228440
2013-01-04  0.901805  1.171216  0.520260 -1.197071
```
• Allow slicing with float-like values on an integer index for `.ix`. Previously this was only enabled for `.loc`:

Previous Behavior

```python
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

```python
In [45]: s.ix[-1.0:2]
Out[45]:
-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

```python
In [4]: df.loc[2:3]
KeyError: 'start bound [2] is not the [index]'
```

New Behavior

```python
In [4]: df.loc[2:3]
TypeError: Cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'> with <type 'int'> keys
```

### Categorical Changes

In prior versions, Categoricals that had an unspecified ordering (meaning no `ordered` keyword was passed) were defaulted as ordered Categoricals. Going forward, the `ordered` keyword in the Categorical constructor will default to `False`. Ordering must now be explicit.

Furthermore, previously you could change the `ordered` attribute of a Categorical by just setting the attribute, e.g. `cat.ordered=True`; This is now deprecated and you should use `cat.as_ordered()` or `cat.as_unordered()`. These will by default return a new object and not modify the existing object. (GH9347, GH9190)

Previous Behavior

```python
In [3]: s = Series([0,1,2], dtype='category')
In [4]: s
Out[4]:
0   0
```

```python
In [4]: s
Out[4]:
0   0
```
1 1
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 [46]: s = Series([0, 1, 2], dtype='category')

In [47]: s
Out[47]:
0 0
1 1
2 2
dtype: category
Categories (3, int64): [0, 1, 2]

In [48]: s.cat.ordered
Out[48]: False

In [49]: s = s.cat.as_ordered()

In [50]: s
Out[50]:
0 0
1 1
2 2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [51]: s.cat.ordered
Out[51]: True

# you can set in the constructor of the Categorical
In [52]: s = Series(Categorical([0, 1, 2], ordered=True))

In [53]: s
Out[53]:
0 0
1 1
2 2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [54]: s.cat.ordered
Out[54]: True

For ease of creation of series of categorical data, we have added the ability to pass keywords when calling .astype(). These are passed directly to the constructor.

In [55]: s = Series(["a","b","c","a"]).astype('category',ordered=True)

In [56]: s
Out[56]:
   0  a
   1  b
   2  c
   3  a
dtype: category
Categories (3, object): [a < b < c]

In [57]: s = Series(["a","b","c","a"]).astype('category',categories=list('abcdef'),ordered=False)

In [58]: s
Out[58]:
   0  a
   1  b
   2  c
   3  a
dtype: category
Categories (6, object): [a, b, c, d, e, f]

Other API Changes

• Index.duplicated now returns np.array(dtype=bool) rather than Index(dtype=object) containing bool values. (GH8875)

• DataFrame.to_json now returns accurate type serialisation for each column for frames of mixed dtype (GH9037)

Previously data was coerced to a common dtype before serialisation, which for example resulted in integers being serialised to floats:

In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out[2]: '{"f":{"0":3.0,\"1\":4.2},\"i\":{"0":1.0,\"1\":2.0}}'

Now each column is serialised using its correct dtype:

In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out[2]: '{\"f\":\"0.3,\"1\":4.2\",\"i\":\"0\",\"1\":2\"}'}

• DatetimeIndex, PeriodIndex and TimedeltaIndex.summary now output the same format. (GH9116)

• TimedeltaIndex.freqstr now output the same string format as DatetimeIndex. (GH9116)

• Bar and horizontal bar plots no longer add a dashed line along the info axis. The prior style can be achieved with matplotlib’s axhline or axvline methods (GH9088).

• Series accessors .dt, .cat and .str now raise AttributeError instead of TypeError if the series does not contain the appropriate type of data (GH9617). This follows Python’s built-in exception hierarchy more closely and ensures that tests like hasattr(s, ‘cat’) are consistent on both Python 2 and 3.
• **Series** now supports bitwise operation for integral types (GH9016). Previously even if the input dtypes were integral, the output dtype was coerced to **bool**.

**Previous Behavior**

```
In [2]: pd.Series([0,1,2,3], list('abcd')) | pd.Series([4,4,4,4], list('abcd'))
Out[2]:
a   True
b   True
c   True
d   True
dtype: bool
```

**New Behavior.** If the input dtypes are integral, the output dtype is also integral and the output values are the result of the bitwise operation.

```
In [2]: pd.Series([0,1,2,3], list('abcd')) | pd.Series([4,4,4,4], list('abcd'))
Out[2]:
a   4
b   5
c   6
d   7
dtype: int64
```

• During division involving a **Series** or **DataFrame**, 0/0 and 0//0 now give `np.nan` instead of `np.inf` (GH9144, GH8445)

**Previous Behavior**

```
In [2]: p = pd.Series([0, 1])
In [3]: p / 0
Out[3]:
0   inf
1   inf
dtype: float64

In [4]: p // 0
Out[4]:
0   inf
1   inf
dtype: float64
```

**New Behavior**

```
In [59]: p = pd.Series([0, 1])
In [60]: p / 0
Out[60]:
0   NaN
1   inf
dtype: float64

In [61]: p // 0
Out[61]:
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:

  ```python
  In [4]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
  Out[4]: Timestamp('2000-01-31 00:00:00')
  ```

  Fixed behavior:

  ```python
  In [62]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
  Out[62]: Timestamp('2000-02-28 00:00:00')
  ```

  To reproduce the old behavior, simply add more precision to the label (e.g., use `2000-02-01` instead of `2000-02`).

### Deprecations

- The `rplot` trellis plotting interface is deprecated and will be removed in a future version. We refer to external packages like seaborn for similar but more refined functionality (GH3445). The documentation includes some examples how to convert your existing code using rplot to seaborn: [rplot docs](#).

- The `pandas.sandbox.qt` pandas 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)

### 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)

### 1.3.3 Performance Improvements

- Fixed a performance regression for `.loc` indexing with an array or list-like (GH9126).

- `DataFrame.to_json` 30x performance improvement for mixed dtypes 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)

1.3.4 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)
• 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 mising 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']

Chapter 1. What’s New
1.4 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**

1.4.1 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     x  0.751953
       x  0.561512
   1     z  0.572214
       y  0.740693

In [3]: df.index.lexsort_depth
Out[3]: 1

# in prior versions this would raise a KeyError
# will now show a PerformanceWarning
In [4]: df.loc[(1, 'z')]
Out[4]:
   jolie
   jim   joe
   1     z  0.572214

# lexically sorting
In [5]: df2 = df.sortlevel()

In [6]: df2
Out[6]:
   jolie
   jim   joe
   0     x  0.751953
       x  0.561512
   1     y  0.740693
       z  0.572214

In [7]: df2.index.lexsort_depth
Out[7]: 2
In [8]: df2.loc[(1, 'z')]
Out[8]:
     jolie
gim joe
1   z  0.572214

• Bug in unique of Series with category dtype, which returned all categories regardless whether they were “used” or not (see GH8559 for the discussion). Previous behaviour was to return all categories:

In [3]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])
In [4]: cat
Out[4]:
[a, b, a]
Categories (3, object): [a < b < c]
In [5]: cat.unique()
Out[5]: array(['a', 'b', 'c'], dtype=object)

Now, only the categories that do effectively occur in the array are returned:

In [9]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])
In [10]: cat.unique()
Out[10]: array(['a', 'b'], dtype=object)

• Series.all and Series.any now support the level and skipna parameters. Series.all, Series.any, Index.all, and Index.any no longer support the out and keepdims parameters, which existed for compatibility with ndarray. Various index types no longer support the all and any aggregation functions and will now raise TypeError (GH8302).

• Allow equality comparisons of Series with a categorical dtype and object dtype; previously these would raise TypeError (GH8938)

• Bug in NDFrame: conflicting attribute/column names now behave consistently between getting and setting. Previously, when both a column and attribute named y existed, data.y would return the attribute, while data.y = z would update the column (GH8994)

In [11]: data = pd.DataFrame({'x': [1, 2, 3]})
In [12]: data.y = 2
In [13]: data['y'] = [2, 4, 6]
In [14]: data
Out[14]:
     x  y
0  1  2
1  2  4
2  3  6

# 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], dtype=int64)

• 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 2
b 1
a 0
dtype: int32

1.4.2 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   False
    1   True   False
    2   False   False
    3   True   True
```

• Added support for `utcfromtimestamp()`, `fromtimestamp()`, and `combine()` on `Timestamp` class (GH5351).

• Added Google Analytics (pandas.io.ga) basic documentation (GH8835). See here.

• Timedelta arithmetic returns `NotImplemented` in unknown cases, allowing extensions by custom classes (GH8813).

• Timedelta now supports arithmetic with `numpy.ndarray` objects of the appropriate dtype (numpy 1.8 or newer only) (GH8884).

• Added `Timedelta.to_timedelta64()` method to the public API (GH8884).

• Added `gbq.generate_bq_schema()` function to the gbq module (GH8325).

• Series now works with map objects the same way as generators (GH8909).

• Added context manager to `HDFStore` for automatic closing (GH8791).

• `to_datetime` gains an `exact` keyword to allow for a format to not require an exact match for a provided format string (if its `False`). `exact` defaults to `True` (meaning that exact matching is still the default) (GH8904)

• Added `axvlines` boolean option to parallel_coordinates plot function, determines whether vertical lines will be printed, default is True

• Added ability to read table footers to `read_html` (GH8552)

• `to_sql` now infers datatypes of non-NA values for columns that contain NA values and have dtype `object` (GH8778).

### 1.4.3 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)

1.4.4 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 paramaters 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; buggy with `np.array_split` (GH8846)

1.4. v0.15.2 (December 12, 2014) 37
• 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)

1.5 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

1.5.1 API changes

• `s.dt.hour` and other `.dt` accessors will now return `np.nan` for missing values (rather than previously -1), (GH8689)
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:
In [6]: s.dt.hour
Out[6]:
0 0
1 0
2 -1
3 0
4 0
dtype: int64

current behavior:
In [4]: s.dt.hour
Out[4]:
0 0
1 0
2 NaN
3 0
4 0
dtype: float64

• groupby with as_index=False will not add erroneous extra columns to result (GH8582):

In [5]: np.random.seed(2718281)

In [6]: df = pd.DataFrame(np.random.randint(0, 100, (10, 2)),
...: columns=['jim', 'joe'])
...

In [7]: df.head()
Out[7]:
   jim  joe
0    61   81
1    96   49
2    55   65
3    72   51
4    77   12

In [8]: ts = pd.Series(5 * np.random.randint(0, 3, 10))

previous behavior:
In [4]: df.groupby(ts, as_index=False).max()
Out[4]:
      NaN  jim  joe
current behavior:

```python
In [9]: df.groupby(ts, as_index=False).max()
Out[9]:
          jim   joe
0       72    83
1       77    84
2       96    65
```

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

```python
In [10]: df = pd.DataFrame({'jim': range(5), 'joe': range(5, 10)})
In [11]: df
Out[11]:
     jim   joe
0      0    5
1      1    6
2      2    7
3      3    8
4      4    9
```

```python
In [12]: gr = df.groupby(df['jim'] < 2)
```

previous behavior (excludes 1st column from output):

```python
In [4]: gr.apply(sum)
Out[4]:
    joe
   jim
False  24
True   11
```

current behavior:

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

- Support for slicing with monotonic decreasing indexes, even if `start` or `stop` is not found in the index (GH7860):

```python
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 `callsMMMYYYY` and `putsMMMYYYY`. The next expiry is saved as `calls` and `puts`.

**New features:**

- The expiry parameter can now be a single date or a list-like object containing dates.
- A new property `expiry_dates` was added, which returns all available expiry dates.

**Current behavior:**

In [17]: from pandas.io.data import Options

In [18]: aapl = Options('aapl','yahoo')

In [19]: aapl.get_call_data().iloc[0:5,0:1]
Out[19]:
<table>
<thead>
<tr>
<th>Last</th>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>65.25</td>
<td>65</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00065000</td>
</tr>
<tr>
<td>70.72</td>
<td>70</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00070000</td>
</tr>
<tr>
<td>57.21</td>
<td>75</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00075000</td>
</tr>
<tr>
<td>47.50</td>
<td>80</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00080000</td>
</tr>
<tr>
<td>42.50</td>
<td>85</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00085000</td>
</tr>
</tbody>
</table>

In [20]: aapl.expiry_dates
Out[20]:
```
[datetime.date(2015, 6, 19),
datetime.date(2015, 6, 26),
datetime.date(2015, 7, 2),
datetime.date(2015, 7, 10),
datetime.date(2015, 7, 17),
datetime.date(2015, 7, 24),
datetime.date(2015, 7, 31),
datetime.date(2015, 8, 21),
datetime.date(2015, 10, 16),
datetime.date(2015, 12, 18),
datetime.date(2016, 1, 15),
```
In [21]: aapl.get_near_stock_price(expiry=aapl.expiry_dates[0:3]).iloc[0:5,0:1]
Out[21]:

<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>2015-06-26</td>
<td>call</td>
<td>AAPL150626C00127000</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>2015-07-02</td>
<td>call</td>
<td>AAPL150702C00127000</td>
<td>2.36</td>
</tr>
<tr>
<td>128</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00128000</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>2015-06-26</td>
<td>call</td>
<td>AAPL150626C00128000</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>2015-07-02</td>
<td>call</td>
<td>AAPL150702C00128000</td>
<td>1.88</td>
</tr>
</tbody>
</table>

See the Options documentation in Remote Data

• pandas now also registers the datetime64 dtype in matplotlib’s units registry to plot such values as dates. 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).

1.5.2 Enhancements

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

  In [22]: from collections import deque
  
  In [23]: df1 = pd.DataFrame([1, 2, 3])
  
  In [24]: 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 [25]: pd.concat(deque((df1, df2)))
  
  Out[25]:
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

• 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 [26]: dfi = DataFrame(1,index=pd.MultiIndex.from_product([['a'],range(1000)]),columns=['A'])

  previous behavior:
  
  In [1]: dfi.memory_usage(index=True)

  # this was underreported in prior versions
  In [1]: dfi.memory_usage(index=True)
Out[1]:
Index 8000 # took about 2400 bytes in < 0.15.1
A 8000
dtype: int64

current behavior:

In [27]: dfi.memory_usage(index=True)
Out[27]:
Index 11020
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)

1.5.3 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)

1.5. v0.15.1 (November 9, 2014)
• 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).

1.6  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 funtions, 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](https://pandas.pydata.org/pandas-docs/stable/categorical.html).

### 1.6.1 New features

#### Categoricals in Series/DataFrame

*Categorical* can now be included in *Series* and *DataFrames* and gained new methods to manipulate. Thanks to Jan Schulz for much of this API/implementation. (GH3943, GH5313, GH5314, GH7444, GH7839, GH7848, GH7864, GH7914, GH7768, GH8006, GH3678, GH8075, GH8076, GH8143, GH8453, GH8518).

For full docs, see the *categorical introduction* and the *API documentation*.

```python
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

```python
In [4]: df['grade'].cat.categories = ['very good', 'good', 'very bad']
```

# Reorder the categories and simultaneously add the missing categories

```python
In [5]: df['grade'] = df['grade'].cat.set_categories(['very bad', 'bad', 'medium', 'good', 'very good'])
```

```python
In [6]: df['grade']
```

```
Out[6]:
0   very good
1     good
```
pandas: powerful Python data analysis toolkit, Release 0.16.2

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("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()
Out[8]:
grade
very bad 1
bad   NaN
medium  NaN
good  2
very good 3
dtype: float64

• pandas.core.group_agg and pandas.core.factor_agg were removed. As an alternative, construct a dataframe and use df.groupby(<group>).agg(<func>).

• Supplying “codes/labels and levels” to the Categorical constructor is not supported anymore. Supplying two arguments to the constructor is now interpreted as “values and levels (now called ‘categories’)”. Please change your code to use the from_codes() constructor.

• The Categorical.labels attribute was renamed to Categorical.codes and is read only. If you want to manipulate codes, please use one of the API methods on Categoricals.

• The Categorical.levels attribute is renamed to Categorical.categories.

TimedeltaIndex/Scalar

We introduce a new scalar type Timedelta, which is a subclass of datetime.timedelta, and behaves in a similar manner, but allows compatibility with np.timedelta64 types as well as a host of custom representation, parsing, and attributes. This type is very similar to how Timestamp works for datetimes. It is a nice-API box for the type. See the docs. (GH3009, GH4533, GH8209, GH8187, GH8190, GH7869, GH7661, GH8345, GH8471)
Warning: Timedelta scalars (and TimedeltaIndex) component fields are not the same as the component fields on a datetime.timedelta object. For example, .seconds on a datetime.timedelta object returns the total number of seconds combined between hours, minutes and seconds. In contrast, the pandas Timedelta breaks out hours, minutes, microseconds and nanoseconds separately.

# 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.
The arguments to pd.to_timedelta are now (arg,unit='ns',box=True,coerce=False). previously were (arg,box=True,unit='ns') as these are more logical.

Construct a scalar

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

In [14]: td = Timedelta('1 hour 3m 15.5us')

In [15]: td.seconds
Out[15]: 3780L

In [16]: td.microseconds
Out[16]: 15L
In [17]: td.nanoseconds
Out[17]: 500L

Construct a TimedeltaIndex

In [18]: TimedeltaIndex(['1 days', '1 days, 00:00:05',
                   np.timedelta64(2, 'D'), timedelta(days=2, seconds=2)])
Out[18]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
                         '2 days 00:00:02'], dtype='timedelta64[ns]', freq=None)

Constructing a TimedeltaIndex with a regular range

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]' )

In [20]: timedelta_range(start='1 days', end='2 days', freq='30T')
Out[20]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                         '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                         '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                         '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                         '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                         '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                         '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                         '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: int32

You can select with partial string selections

In [23]: s['1 day 00:00:02']
Out[23]: 2
Finally, the combination of TimedeltaIndex with DatetimeIndex allow certain combination operations that are NaT preserving:

```python
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', offset='D'),
           Timestamp('2013-01-02 00:00:00', offset='D'),
           Timestamp('2013-01-03 00:00:00', offset='D')]
In [29]: (dti + tdi).tolist()
Out[29]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]
In [30]: (dti - tdi).tolist()
Out[30]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

- iteration of a Series e.g. list(Series(...)) of timedelta64[ns] would prior to v0.15.0 return np.timedelta64 for each element. These will now be wrapped in Timedelta.

**Memory Usage**

Implemented methods to find memory usage of a DataFrame. See the FAQ for more. (GH6852).

A new display option `display.memory_usage` (see Options and Settings) sets the default behavior of the `memory_usage` argument in the `df.info()` method. By default `display.memory_usage` is True.

```python
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'>
Int64Index: 5000 entries, 0 to 4999
Data columns (total 8 columns):
bool      5000 non-null bool
```
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   40000
bool    5000
complex128  80000
datetime64[ns]  40000
float64  40000
int64    40000
object   20000
timedelta64[ns]  40000
categorical  5800
dtype: int64

.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

# 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
Out[40]:
0   9
1   9
2   9
3   9
dtype: int64

In [41]: s.dt.second
Out[41]:
0   12
1   12
2   12
3   12
dtype: int64
In [42]: s.dt.day
Out[42]:
0    1
1    2
2    3
3    4
dtype: int64

In [43]: s.dt.freq
Out[43]: <Day>

This enables nice expressions like this:

In [44]: s[s.dt.day==2]
Out[44]:
0  2013-01-02 09:10:12
dtype: datetime64[ns]

You can easily produce tz aware transformations:

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

In [46]: stz
Out[46]:
0  2013-01-01 09:10:12-05:00
1  2013-01-02 09:10:12-05:00
2  2013-01-03 09:10:12-05:00
3  2013-01-04 09:10:12-05:00
dtype: object

In [47]: stz.dt.tz
Out[47]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>

You can also chain these types of operations:

In [48]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[48]:
0  2013-01-01 04:10:12-05:00
1  2013-01-02 04:10:12-05:00
2  2013-01-03 04:10:12-05:00
3  2013-01-04 04:10:12-05:00
dtype: object

The .dt accessor works for period and timedelta dtypes.

# period
In [49]: s = Series(period_range('20130101', periods=4, freq='D'))

In [50]: s
Out[50]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: object

In [51]: s.dt.year
Out[51]:
0  2013
In [52]: s.dt.day
Out[52]:
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

Timezone handling improvements

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

In [58]: ts = 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]', freq='H', tz='US/Eastern')

**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=None)

• tz_localize now accepts the ambiguous keyword which allows for passing an array of bools indicating whether the date belongs in DST or not, ‘NaT’ for setting transition times to NaT, ‘infer’ for inferring DST/non-DST, and ‘raise’ (default) for an AmbiguousTimeError to be raised. See the docs for more details (GH7943)

• DataFrame.tz_localize and DataFrame.tz_convert now accepts an optional level argument for localizing a specific level of a MultiIndex (GH7846)

• Timestamp.tz_localize and Timestamp.tz_convert now raise TypeError in error cases, rather than Exception (GH8025)

• a timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone (rather than being a naive datetime64[ns]) as object dtype (GH8411)

• Timestamp.__repr__ displays dateutil.tz.tzoffset info (GH7907)

**Rolling/Expanding Moments improvements**

• rolling_min(), rolling_max(), rolling_cov(), and rolling_corr() now return objects with all NaN when 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** [65]: rolling_min(s, window=10, min_periods=5)
**Out**[65]:
0 NaN
1 NaN
2 NaN
pandas: powerful Python data analysis toolkit, Release 0.16.2

```python
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):

```python
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])):

```python
In [66]: rolling_sum(Series(range(4)), window=3, min_periods=0, center=True)
Out[66]:
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)

```python
In [67]: s = Series([10.5, 8.8, 11.4, 9.7, 9.3])
```

Behavior prior to 0.15.0:

```python
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

```python
In [68]: rolling_window(s, window=3, win_type='triang', center=True)
Out[68]:
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)):

```python
In [69]: s = Series([1, None, None, None, 2, 3])
In [51]: ewma(s, com=3., min_periods=2)
Out[51]:
   0    NaN
   1    NaN
   2 1.00000
   3 1.00000
   4 1.57143
   5 2.18919
   dtype: float64
```

New behavior (note values start at index 4, the location of the 2nd (since `min_periods=2`) non-empty value):

```python
In [70]: ewma(s, com=3., min_periods=2)
Out[70]:
   0    NaN
   1    NaN
   2    NaN
   3    NaN
   4 1.75964
   5 2.38378
   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)

```python
In [71]: ewma(Series([None, 1., 8.]), com=2.)
Out[71]:
   0    NaN
   1  1.0
   2  5.2
   dtype: float64

In [72]: ewma(Series([1., None, 8.]), com=2., ignore_na=True)  # pre-0.15.0 behavior
Out[72]:
   0    NaN
   1  1.0
   2  5.2
   dtype: float64
```
```python
In [73]: ewma(Series([1., None, 8.]), com=2., ignore_na=False)  # new default
Out[73]:
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:

```python
In [74]: 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):

```python
In [75]: ewmvar(s, com=2., bias=False)
Out[75]:
   0    NaN
   1  0.500000
   2  1.210526
   3  4.089069
dtype: float64
```

```python
In [76]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
Out[76]:
   0    NaN
   1  2.083333
   2  1.583333
   3  1.425439
dtype: float64
```

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

**Improvements in the sql io module**

- Added support for a chunksize parameter to `to_sql` function. This allows DataFrame to be written in chunks and avoid packet-size overflow errors (GH8062).
- Added support for a chunksize parameter to `read_sql` function. Specifying this argument will return an iterator through chunks of the query result (GH2908).
- Added support for writing `datetime.date` and `datetime.time` object columns with `to_sql` (GH6932).
- Added support for specifying a schema to read from/write to with `read_sql_table` and `to_sql` (GH7441, GH7952). For example:
  ```python
df.to_sql('table', engine, schema='other_schema')
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).

### 1.6.2 Backwards incompatible API changes

**Breaking changes**

API changes related to `Categorical` (see [here](#) for more details):

- The `Categorical` constructor with two arguments changed from “codes/labels and levels” to “values and levels (now called ‘categories’)”. This can lead to subtle bugs. If you use `Categorical` directly, please audit your code by changing it to use the `from_codes()` constructor.

An old function call like (prior to 0.15.0):

```python
pd.Categorical([0,1,0,2,1], levels=['a', 'b', 'c'])
```

will have to adapted to the following to keep the same behaviour:
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.

```python
In [77]: df = DataFrame([['a'], ['b']], index=[1,2])
```

```python
In [78]: df
Out[78]:
   0
 0  a
 1  b
```

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 [79]: df.loc[[1,3]]
Out[79]:
   0  a
  1  b
```

```python
In [80]: df.loc[[1,3],:]
Out[80]:
   0  a
  1  b
```

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

```python
In [81]: 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'])
```

```python
In [82]: p
```
Out[82]:
<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 [83]: p.loc[['ItemA','ItemD'],:,'D']
Out[83]:
          ItemA  ItemD
   1      3.0  NaN
   2      7.0  NaN
   3     11.0  NaN

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

In [84]: s = Series(np.arange(3,dtype='int64'),
   ....:   index=MultiIndex.from_product([[4],['A','bar','baz']],
   ....:   names=['one','two'])
   ....:   ).sortlevel()
   ....:
In [85]: s
Out[85]:
one   two
   A bar  1
   baz  2
   foo  0
dtype: int64
In [86]: try:
   ....:   s.loc[['D']]
   ....:   except KeyError as e:
   ....:     print("KeyError: " + str(e))
   ....:
KeyError: 'cannot index a multi-index axis with these keys'

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

Previously, assigning to None in numeric containers changed the dtype to object (or errored, depending on the call). It now uses NaN:

In [87]: s = Series([1, 2, 3])
In [88]: s.loc[0] = None
In [89]: s
Out[89]:
   0  NaN
   1   2
   2   3
dtype: float64

NaT is now used similarly for datetime containers.

For object containers, we now preserve None values (previously these were converted to NaN values).
In [90]: s = Series(["a", "b", "c")
In [91]: s.loc[0] = None
In [92]: s
Out[92]:
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 [93]: s = Series([1, 2, 3])
In [94]: s2 = s
In [95]: s += 1.5

Behavior prior to v0.15.0

# the original object
In [5]: s
Out[5]:
0   2.5
1   3.5
2   4.5
dtype: float64

# a reference to the original object
In [7]: s2
Out[7]:
0   1
1   2
2   3
dtype: int64

This is now the correct behavior

# the original object
In [96]: s
Out[96]:
0   2.5
1   3.5
2   4.5
dtype: float64

# a reference to the original object
In [97]: s2
Out[97]:
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 [98]: i = date_range('1/1/2011', periods=3, freq='10s', tz = 'US/Eastern')
In [99]: i
Out[99]: 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]', freq='10S', tz='US/Eastern')
```

```python
In [100]: df = DataFrame( {'a' : i } )
In [101]: df
Out[101]:
   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
```

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
```

- merge, DataFrame.merge, and ordered_merge now return the same type as the left argument (GH7737).
• Previously an enlargement with a mixed-dtype frame would act unlike `.append` which will preserve dtypes (related GH2578, GH8176):

```
In [103]: df = DataFrame([[True, 1],[False, 2]],
                 columns=['female','fitness']
                 ....:

In [104]: df
Out[104]:
      female  fitness
0      True        1
1     False        2

In [105]: df.dtypes
Out[105]:
female    bool
fitness   int64
dtype: object

# dtypes are now preserved
```

• Series.to_csv() now returns a string when path=None, matching the behaviour of DataFrame.to_csv() (GH8215).

• `read_hdf` now raises IOError when a file that doesn’t exist is passed in. Previously, a new, empty file was created, and a KeyError raised (GH7715).

• DataFrame.info() now ends its output with a newline character (GH8114)

• Concatenating no objects will now raise a ValueError rather than a bare Exception.

• Merge errors will now be sub-classes of ValueError rather than raw Exception (GH8501)

• DataFrame.plot and Series.plot keywords are now have consistent orders (GH8037)

**Internal Refactoring**

In 0.15.0 Index has internally been refactored to no longer sub-class ndarray but instead subclass PandasObject, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (GH5080, GH7439, GH7796, GH8024, GH8367, GH7997, GH8522):

• you may need to unpickle pandas version < 0.15.0 pickles using `pd.read_pickle` rather than `pickle.load`. See `pickle docs`
• when plotting with a PeriodIndex, the matplotlib internal axes will now be arrays of Period rather than a PeriodIndex (this is similar to how a DatetimeIndex passes arrays of datetimes now)

• MultiIndex will now raise similarly to other pandas objects w.r.t. truth testing, see here (GH7897).

• When plotting a DatetimeIndex directly with matplotlib’s plot function, the axis labels will no longer be formatted as dates but as integers (the internal representation of a datetime64). UPDATE This is fixed in 0.15.1, see here.

Deprecations

• The attributes Categorical labels and levels attributes are deprecated and renamed to codes and categories.

• The outtype argument to pd.DataFrame.to_dict has been deprecated in favor of orient. (GH7840)

• The convert_dummies method has been deprecated in favor of get_dummies (GH8140)

• The infer_dst argument in tz_localize will be deprecated in favor of ambiguous to allow for more flexibility in dealing with DST transitions. Replace infer_dst=True with ambiguous=’infer’ for the same behavior (GH7943). See the docs for more details.

• The top-level pd.value_range has been deprecated and can be replaced by .describe() (GH8481)

• The Index set operations + and – were deprecated in order to provide these for numeric type operations on certain index types. + can be replaced by .union() or |, and – by .difference(). Further the method name Index.diff() is deprecated and can be replaced by Index.difference() (GH8226)

```python
# +
Index(["a","b","c"]) + Index(["b","c","d"])
# should be replaced by
Index(["a","b","c"]).union(Index(["b","c","d"]))

# -
Index(["a","b","c"]) - Index(["b","c","d"])
# should be replaced by
Index(["a","b","c"]).difference(Index(["b","c","d"]))
```

• The infer_types argument to read_html() now has no effect and is deprecated (GH7762, GH7032).

Removal of prior version deprecations/changes

• Remove DataFrame.delevel method in favor of DataFrame.reset_index

1.6.3 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 [109]: df = DataFrame({'catA': ['foo', 'foo', 'bar'] * 8,
                   'catB': ['a', 'b', 'c', 'd'] * 6,
                   'numC': np.arange(24),
                   'numD': np.arange(24.) + .5})

In [110]: df.describe(include=['object'])
Out[110]:
   catA  catB
count 24    24
unique 2     4
top    foo    d
freq   16    6

In [111]: df.describe(include=['number', 'object'], exclude=['float'])
Out[111]:
   catA  catB  numC
count 24    24   24.000000
unique 2     4   NaN
top    foo    d   NaN
freq   16    6   NaN
mean  NaN   NaN  11.500000
std   NaN   NaN  7.071068
min   NaN   NaN  0.000000
25%   NaN   NaN  5.750000
50%   NaN   NaN  11.500000
75%   NaN   NaN  17.250000
max   NaN   NaN  23.000000

Requesting all columns is possible with the shorthand ‘all’

In [112]: df.describe(include='all')
Out[112]:
   catA  catB  numC  numD
count 24    24 24.000000 23.000000
unique 2     4    NaN      NaN
top    foo    d    NaN      NaN
freq   16    6    NaN      NaN
mean  NaN   NaN  11.500000  23.000000
std   NaN   NaN  7.071068  7.071068
min   NaN   NaN  0.000000  0.000000
25%   NaN   NaN  5.750000  5.750000
50%   NaN   NaN  11.500000 11.500000
75%   NaN   NaN  17.250000 17.250000
max   NaN   NaN  23.000000 23.000000
```
pandas: powerful Python data analysis toolkit, Release 0.16.2

count  24  24  24.000000  24.000000
unique  2  4  NaN  NaN
top  foo  d  NaN  NaN
tfreq  16  6  NaN  NaN
mean  NaN  NaN  11.500000  12.000000
std  NaN  NaN  7.071068  7.071068
min  NaN  NaN  0.000000  0.500000
25%  NaN  NaN  5.750000  6.250000
50%  NaN  NaN  11.500000  12.000000
75%  NaN  NaN  17.250000  17.750000
max  NaN  NaN  23.000000  23.500000

Without those arguments, 'describe' will behave as before, including only numerical columns or, if none are, only categorical columns. See also the docs

• **Added** split as an option to the orient argument in pd.DataFrame.to_dict. (GH7840)

• The get_dummies method can now be used on DataFrames. By default only categorical columns are encoded as 0’s and 1’s, while other columns are left untouched.

```
In [113]: df = DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
                        'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
                        'C': [1, 2, 3]})

In [114]: pd.get_dummies(df)
Out[114]:
     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

• **PeriodIndex** supports resolution as the same as DatetimeIndex (GH7708)

• pandas.tseries.holiday has added support for additional holidays and ways to observe holidays (GH7070)

• pandas.tseries.holiday.Holiday now supports a list of offsets in Python3 (GH7070)

• pandas.tseries.holiday.Holiday now supports a days_of_week parameter (GH7070)

• GroupBy.nth() now supports selecting multiple nth values (GH7910)

```
In [115]: business_dates = date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [116]: df = DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [117]: df.groupby((df.index.year, df.index.month)).nth([0, 3, -1])
Out[117]:
     a  b
2014-04-01  1  1
2014-04-04  1  1
2014-04-30  1  1
2014-05-01  1  1
2014-05-06  1  1
2014-05-30  1  1
2014-06-02  1  1
2014-06-05  1  1
2014-06-30  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.

```python
In [118]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [119]: idx
Out[119]:
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='int64', freq='H')
```

```python
In [120]: idx + pd.offsets.Hour(2)
Out[120]:
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='int64', freq='H')
```

```python
In [121]: idx + Timedelta('120m')
Out[121]:
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='int64', freq='H')
```

```python
In [122]: idx = pd.period_range('2014-07', periods=5, freq='M')

In [123]: idx
Out[123]:
dtype='int64', freq='M')
```

```python
In [124]: idx + pd.offsets.MonthEnd(3)
Out[124]:
dtype='int64', freq='M')
```

- Added experimental compatibility with `openpyxl` for versions >= 2.0. The `DataFrame.to_excel` method engine keyword now recognizes `openpyxl1` and `openpyxl2` which will explicitly require openpyxl v1 and v2 respectively, failing if the requested version is not available. The `openpyxl` engine is now a meta-engine that automatically uses whichever version of openpyxl is installed. (GH7177)

- `DataFrame.fillna` can now accept a `DataFrame` as a fill value (GH8377)

- Passing multiple levels to `stack()` will now work when multiple level numbers are passed (GH7660). See `Reshaping by stacking and unstacking`.

- `set_names()`, `set_labels()`, and `set_levels()` methods now take an optional level keyword argument to all modification of specific level(s) of a MultiIndex. Additionally `set_names()` now accepts a scalar string value when operating on an Index or on a specific level of a MultiIndex (GH7792)

```python
In [125]: idx = MultiIndex.from_product([['a'], range(3), list("pqr")], names=['foo', 'bar', 'baz'])

In [126]: idx.set_names('qux', level=0)
Out[126]:
MultiIndex(levels=[['u'a'], [0, 1, 2], [u'p', u'q', u'r']],
labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2], [u'qux', u'bar', u'baz'])
```

```python
In [127]: idx.set_names(['qux','baz'], level=[0,1])
Out[127]:
MultiIndex(levels=[['u'a'], [0, 1, 2], [u'p', u'q', u'r']],
labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2], [u'qux', u'baz', u'baz'])
```

```python
In [128]: idx.set_levels([['a','b','c'], level='bar'])
```
Out[128]:
MultiIndex(levels=[[u'a'], [u'a', u'b', u'c'], [u'p', u'q', u'r']],
       labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2],
       names=[u'foo', u'bar', u'baz'])

In [129]: idx.set_levels([['a','b','c'],[1,2,3]], level=[1,2])
Out[129]:
MultiIndex(levels=[[u'a'], [u'a', u'b', u'c'], [1, 2, 3]],
       labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2],
       names=[u'foo', u'bar', u'baz'])

- Index.isin now supports a level argument to specify which index level to use for membership tests (GH7892, GH7890)

In [1]: idx = MultiIndex.from_product([[0, 1], ['a', 'b', 'c']])

In [2]: idx.values
Out[2]: array([(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')], dtype=object)

In [3]: idx.isin(['a', 'c', 'e'], level=1)
Out[3]: array([ True, False, True, True, False, True], dtype=bool)

- Index now supports duplicated and drop_duplicates. (GH4060)

In [130]: idx = Index([1, 2, 3, 4, 1, 2])

In [131]: idx
Out[131]: Int64Index([1, 2, 3, 4, 1, 2], dtype='int64')

In [132]: idx.duplicated()
Out[132]: array([False, False, False, False, True, True], dtype=bool)

In [133]: idx.drop_duplicates()
Out[133]: 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)

1.6.4 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)

1.6.5 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 reffered 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).

1.6. v0.15.0 (October 18, 2014)
• Bug in to_excel() where a negative sign was being prepended to positive infinity and was absent for negative infinity (GH7949)
• Bug in area plot draws legend with incorrect alpha when stacked=True (GH8027)
• Period and PeriodIndex addition/subtraction with np.timedelta64 results in incorrect internal representations (GH7740)
• Bug in Holiday with no offset or observance (GH7987)
• Bug in DataFrame.to_latex formatting when columns or index is a MultiIndex (GH7982).
• Bug in DateOffset around Daylight Savings Time produces unexpected results (GH5175).
• Bug in DataFrame.shift where empty columns would throw ZeroDivisionError on numpy 1.7 (GH8019)
• Bug in installation where html_encoding/*.html wasn’t installed and therefore some tests were not running correctly (GH7927).
• Bug in read_html where bytes objects were not tested for in _read (GH7927).
• Bug in DataFrame.stack() when one of the column levels was a datelike (GH8039)
• Bug in broadcasting numpy scalars with DataFrame (GH8116)
• Bug in pivot_table performed with nameless index and columns raises KeyError (GH8103)
• Bug in DataFrame.plot(kind='scatter') draws points and errorbars with different colors when the color is specified by c keyword (GH8081)
• Bug in Float64Index where iat and at were not testing and were failing (GH8092).
• Bug in DataFrame.boxplot() where y-limits were not set correctly when producing multiple axes (GH7528, GH5517).
• Bug in read_csv where line comments were not handled correctly given a custom line terminator or delim_whitespace=True (GH8122).
• Bug in read_html where empty tables caused a StopIteration (GH7575)
• Bug in casting when setting a column in a same-dtype block (GH7704)
• Bug in accessing groups from a GroupBy when the original grouper was a tuple (GH8121).
• Bug in .at that would accept integer indexers on a non-integer index and do fallback (GH7814)
• Bug with kde plot and NaNs (GH8182)
• Bug in GroupBy.count with float32 data type were nan values were not excluded (GH8169).
• Bug with stacked barplots and NaNs (GH8175).
• Bug in resample with non evenly divisible offsets (e.g. ‘7s’) (GH8371)
• Bug in interpolation methods with the limit keyword when no values needed interpolating (GH7173).
• Bug where col_space was ignored in DataFrame.to_string() when header=False (GH8230).
• Bug with DatetimeIndex.asof incorrectly matching partial strings and returning the wrong date (GH8245).
• Bug in plotting methods modifying the global matplotlib rcParams (GH8242).
• Bug in DataFrame.__setitem__ that caused errors when setting a dataframe column to a sparse array (GH8131)
• Bug where DataFrame.boxplot() failed when entire column was empty (GH8181).
• Bug with messed variables in radviz visualization (GH8199).
• Bug in interpolation methods with the limit keyword when no values needed interpolating (GH7173).
• Bug where col_space was ignored in DataFrame.to_string() when header=False (GH8230).
• Bug in to_clipboard that would clip long column data (GH8305)
• Bug in DataFrame terminal display: Setting max_column/max_rows to zero did not trigger auto-resizing of dfs to fit terminal width/height (GH7180).
• Bug in OLS where running with “cluster” and “nw_lags” parameters did not work correctly, but also did not throw an error (GH5884).
• Bug in DataFrame.dropna that interpreted non-existent columns in the subset argument as the ‘last column’ (GH8303)
• Bug in Index.intersection on non-monotonic non-unique indexes (GH8362).
• Bug in masked series assignment where mismatching types would break alignment (GH8387)
• Bug in NDFrame.equals gives false negatives with dtype=object (GH8437)
• Bug in assignment with indexer where type diversity would break alignment (GH8258)
• Bug in NDFrame.loc indexing when row/column names were lost when target was a list/ndarray (GH6552)
• Regression in NDFrame.loc indexing when rows/columns were converted to Float64Index if target was an empty list/ndarray (GH7774)
• Bug in Series that allows it to be indexed by a DataFrame which has unexpected results. Such indexing is no longer permitted (GH8444)
• Bug in item assignment of a DataFrame with 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.

1.7 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
1.7.1 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):

  ```python
  In [6]: from pandas.tseries import offsets
  In [7]: d = pd.Timestamp('2014-01-01 09:00')
  # old behaviour < 0.14.1
  In [8]: d + offsets.MonthEnd()
  Out[8]: Timestamp('2014-01-31 00:00:00')
  
  Starting from 0.14.1 all offsets preserve time by default. The old behaviour can be obtained with `normalize=True`

  # new behaviour
  In [1]: d + offsets.MonthEnd()
  Out[1]: Timestamp('2014-01-31 09:00:00')
  
  In [2]: d + offsets.MonthEnd(normalize=True)
  Out[2]: Timestamp('2014-01-31 00:00:00')
  
  Note that for the other offsets the default behaviour did not change.

- Add back `#N/A  N/A` as a default NA value in text parsing, (regression from 0.12) (GH5521)

- Raise a TypeError on inplace-setting with a `.where` and a non np.nan value as this is inconsistent with a set-item expression like `df[mask] = None` (GH7656)

1.7.2 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):

  ```python
  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')

- 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)

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)

1.7.3 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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)

1.7.4 Experimental

- pandas.io.data.Options has a new method, get_all_data method, and now consistently returns a multi-indexed DataFrame, see the docs. (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).

1.7.5 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_timestrings 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 (GH7313).
• Bug in `isnull()` when `mode.use_inf_as_null == True` where `isnull` wouldn’t test `True` when it encountered an `inf/-inf` (GH7315).
• Bug in inferred_freq results in None for eastern hemisphere timezones (GH7310)
• Bug in `Easter` returns incorrect date when offset is negative (GH7195)
• Bug in broadcasting with `.div`, integer dtypes and divide-by-zero (GH7325)
• Bug in `CustomBusinessDay.apply` raises `NameError` when np.datetime64 object is passed (GH7196)
• Bug in `MultiIndex.append`, `concat` and `pivot_table` don’t preserve timezone (GH6606)
• Bug in `.loc` with a list of indexers on a single-multi index level (that is not nested) (GH7349)
• Bug in `Series.map` when mapping a dict with tuple keys of different lengths (GH7333)
• Bug all `StringMethods` now work on empty Series (GH7242)
• Fix delegation of `read_sql` to `read_sql_query` when query does not contain ‘select’ (GH7324).
• Bug where a string column name assignment to a `DataFrame` with a `Float64Index` raised a `TypeError` during a call to `np.isnan` (GH7366).
• Bug where `NDFrame.replace()` didn’t correctly replace objects with `Period` values (GH7379).
• Bug in `.ix` getitem should always return a Series (GH7150)
• Bug in 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 (GH7464).
• Bug in DataFrame.reset_index loses tz (GH3950)
• Bug in DatetimeIndex.freqstr raises AttributeError when freq is None (GH7606)
• Bug in GroupBy.size created by TimeGrouper raises AttributeError (GH7453)
• Bug in single column bar plot is misaligned (GH7498).
• Bug in area plot with tz-aware time series raises ValueError (GH7471)
• Bug in non-monotonic Index.union may preserve name incorrectly (GH7458)
• Bug in DatetimeIndex.intersection doesn’t preserve timezone (GH4690)
• Bug in rolling_var where a window larger than the array would raise an error (GH7297)
• Bug with last plotted timeseries dictating xlim (GH2960)
• Bug with secondary_y axis not being considered for timeseries xlim (GH3490)
• Bug in Float64Index assignment with a non scalar indexer (GH7586)
• Bug in pandas.core.strings.str_contains does not properly match in a case insensitive fashion when regex=False and case=False (GH7505)
• Bug in expanding_cov, expanding_corr, rolling_cov, and rolling_corr for two arguments with mismatched index (GH7512)
• Bug in to_sql taking the boolean column as text column (GH7678)
• Bug in grouped hist doesn’t handle rot kw and sharex kw properly (GH7234)
• Bug in .loc performing fallback integer indexing with object dtype indices (GH7496)
• Bug (regression) in PeriodIndex constructor when passed Series objects (GH7701).
1.8  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.
  - 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)

1.8.1  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]: df.iloc[:,2:3]
Out[3]:
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [4]: df.iloc[:,1:3]
Out[4]:
   B
0 -0.438313
1 -0.780572
2  0.542241
3 -0.251642
4  1.666563

In [5]: df.iloc[4:6]
Out[5]:
   A   B
4  0.787484  1.666563

These are out-of-bounds selections

df.iloc[[4,5,6]]
IndexError: positional indexers are out-of-bounds

df.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)::−1] now enumerates all elements in reverse

• The DataFrame.interpolate() keyword downcast default has been changed from infer to None. This is to preserve the original dtype unless explicitly requested otherwise (GH6290).

• When converting a dataframe to HTML it used to return Empty DataFrame. This special case has been removed, instead a header with the column names is returned (GH6062).

• Series and Index now internally share more common operations, e.g. factorize(),nunique(),value_counts() are now supported on Index types as well. The Series.weekday property from is removed from Series for API consistency. Using a DatetimeIndex/PeriodIndex method on a Series will now raise a TypeError. (GH4551, GH4056, GH5519, GH6380, GH7206).

• Add  is_month_start, is_month_end, is_quarter_start, is_quarter_end, is_year_start, is_year_end accessors for DateTimeIndex / Timestamp which return a boolean array of whether the timestamp(s) are at the start/end of the month/quarter/year defined by the frequency of the DateTimeIndex / Timestamp (GH4565, GH6998).

• Local variable usage has changed in pandas.eval()/DataFrame.eval()/DataFrame.query() (GH5987). For the DataFrame methods, two things have changed
– Column names are now given precedence over locals
– Local variables must be referred to explicitly. This means that even if you have a local variable that is not a column you must still refer to it with the `@` prefix.
– You can have an expression like `df.query('@a < a')` with no complaints from pandas about ambiguity of the name `a`.
– The top-level `pandas.eval()` function does not allow you use the `@` prefix and provides you with an error message telling you so.
– `NameResolutionError` was removed because it isn’t necessary anymore.

• Define and document the order of column vs index names in query/eval (GH6676)
• `concat` will now concatenate mixed Series and DataFrames using the Series name or numbering columns as needed (GH2385). See the docs
• Slicing and advanced/boolean indexing operations on `Index` classes as well as `Index.delete()` and `Index.drop()` methods will no longer change the type of the resulting index (GH6440, GH7040)

```
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]: Int64Index([1, 2, 3], dtype='int64')
```

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='int32')
```

• `set_index` no longer converts MultiIndexes to an Index of tuples. For example, the old behavior returned an Index in this case (GH6459):

```python
# Old behavior, casted MultiIndex to an Index
In [10]: tuple_ind
Out[10]: Index([(u'a', u'c'), (u'a', u'd'), (u'b', u'c'), (u'b', u'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)
Out[13]:
   0  1
a c  0.471435 -1.190976
d  1.432707 -0.312652
```
This also applies when passing multiple indices to `set_index`:

```
# Old output, 2-level MultiIndex of tuples
In [14]: df_multi.set_index([df_multi.index, df_multi.index])
Out[14]:
   0  1
(a, c) (a, c)  0.471435 -1.190976
(a, d) (a, d)  1.432707 -0.312652
(b, c) (b, c) -0.720589  0.887163
(b, d) (b, d)  0.859588 -0.636524

# New output, 4-level MultiIndex
In [15]: df_multi.set_index([df_multi.index, df_multi.index])
Out[15]:
   0  1
 a c  a c  0.471435 -1.190976
 d a  d d  1.432707 -0.312652
 b c  b c -0.720589  0.887163
 d b  d d  0.859588 -0.636524
```

- `pairwise` keyword was added to the statistical moment functions `rolling_cov`, `rolling_corr`, `ewmcov`, `ewmcorr`, `expanding_cov`, `expanding_corr` to allow the calculation of moving window covariance and correlation matrices (GH4950). See `Computing rolling pairwise covariances and correlations` in the docs.

```
In [16]: df = DataFrame(np.random.randn(10,4),columns=list('ABCD'))

In [17]: covs = rolling_cov(df[['A','B','C']], df[['B','C','D']], 5, pairwise=True)

In [18]: covs[df.index[-1]]
Out[18]:
   B   C   D
A  0.128104  0.183628 -0.047358
B  0.856265  0.058945  0.145447
C  0.058945  0.335350  0.390637
```

- `Series.iteritems()` is now lazy (returns an iterator rather than a list). This was the documented behavior prior to 0.14. (GH6760)

- `Added nunique and value_counts functions to Index for counting unique elements. (GH6734)`

- `stack and unstack` now raise a `ValueError` when the `level` keyword refers to a non-unique item in the `Index` (previously raised a `KeyError`). (GH6738)

- `drop unused order argument from Series.sort; args now are in the same order as Series.order; add na_position arg to conform to Series.order (GH6847)`

- `default sorting algorithm for Series.order is now quicksort, to conform with Series.sort (and numpy defaults)`

- `add inplace keyword to Series.order/sort to make them inverses (GH6859)`

- `DataFrame.sort now places NaNs at the beginning or end of the sort according to the na_position parameter. (GH3917)`

- `accept TextFileReader in concat, which was affecting a common user idiom (GH6583), this was a regression from 0.13.1`
• Added `factorize` functions to `Index` and `Series` to get indexer and unique values (GH7090)
• `describe` on a DataFrame with a mix of `Timestamp` and string like objects returns a different `Index` (GH7088). Previously the index was unintentionally sorted.
• Arithmetic operations with only `bool` dtypes now give a warning indicating that they are evaluated in Python space for `+`, `-`, and `*` operations and raise for all others (GH7011, GH6762, GH7015, GH7210)

```python
x = pd.Series(np.random.rand(10) > 0.5)
y = True
x + y  # warning generated: should do x | y instead
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)
• `AssertionError` to `TypeError` for invalid types passed to `concat` (GH6583)
• Raise a `TypeError` when `DataFrame` is passed an iterator as the `data` argument (GH5357)

1.8.2 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
0  0  1  2  3  4  5
1  6  7  8  9 10 11
2 12 13 14 15 16 17
3 18 19 20 21 22 23
4 24 25 26 27 28 29
5 30 31 32 33 34 35
6 36 37 38 39 40 41
7 42 43 44 45 46 47
8 48 49 50 51 52 53
9 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  5
0  0  1  2  3  4  5
1  6  7  8  9 10 11
2 12 13 14 15 16 17
3 18 19 20 21 22 23
4 24 25 26 27 28 29
5 30 31 32 33 34 35
6 36 37 38 39 40 41
7 42 43 44 45 46 47
8 48 49 50 51 52 53
9 54 55 56 57 58 59

[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 [19]: 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 [20]: with pd.option_context('display.max_rows', 2, 'display.max_columns', 2, 'display.show_dimensions', 'truncate'):
   ....:       print(dfd)
   ....: 0 ... 4
0  0 ... 4
.. ... ... ...
4  20 ...  24

[5 rows x 5 columns]

# will not show dimensions since it is not truncated

In [21]: with pd.option_context('display.max_rows', 10, 'display.max_columns', 40,
....:   'display.show_dimensions', 'truncate'):
....:   print(dfd)
....:
  0   1   2   3   4
 0  0   1   2   3   4
 1  5   6   7   8   9
 2 10  11  12  13  14
 3 15  16  17  18  19
 4 20  21  22  23  24

• Regression in the display of a MultiIndexed Series with display.max_rows is less than the length of the series (GH7101)

• Fixed a bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the large_repr set to 'info' (GH7105)

• The verbose keyword in DataFrame.info(), which controls whether to shorten the info representation, is now None by default. This will follow the global setting in display.max_info_columns. The global setting can be overriden with verbose=True or verbose=False.

• Fixed a bug with the info repr not honoring the display.max_info_columns setting (GH6939)

• Offset/freq info now in Timestamp __repr__ (GH4553)

1.8.3 Text Parsing API Changes

read_csv()/read_table() will now be noiser 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)

1.8.4 Groupby API Changes

More consistent behaviour for some groupby methods:

• groupby head and tail now act more like filter rather than an aggregation:

In [22]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [23]: g = df.groupby('A')
In [24]: g.head(1)  # filters DataFrame
Out[24]:
    A  B
0  1  2
2  5  6

In [25]: g.apply(lambda x: x.head(1))  # used to simply fall-through
Out[25]:
        A  B
   A
0  1  0  1  2
   5  2  5  6

• groupby head and tail respect column selection:

In [26]: g[['B']].head(1)
Out[26]:
    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 [27]: df = DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
In [28]: g = df.groupby('A')
In [29]: g.nth(0)
Out[29]:
   A  B
  A
1  NaN
5  6

# this is equivalent to g.first()
In [30]: g.nth(0, dropna='any')
Out[30]:
   A  B
  A
1  4
5  6

# this is equivalent to g.last()
In [31]: g.nth(-1, dropna='any')
Out[31]:
   A  B
  A
1  4
5  6

Filtering

In [32]: gf = df.groupby('A', as_index=False)
In [33]: gf.nth(0)
Out[33]:
   A  B
In [34]: gf.nth(0, dropna='any')
Out[34]:
   B
0 1
1 5
5 6

- `groupby` will now not return the grouped column for non-cython functions (GH5610, GH5614, GH6732), as it's already the index.

In [35]: df = DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])
In [36]: g = df.groupby('A')
In [37]: g.count()
Out[37]:
   B
A   
  1 1
  5 2
In [38]: g.describe()
Out[38]:
   B
A   
  1 count 1.000000
     mean 4.000000
      std NaN
      min 4.000000
     25% 4.000000
      50% 4.000000
     75% 4.000000
       ... ...
  5 mean 7.000000
     std 1.414214
      min 6.000000
     25% 6.500000
      50% 7.000000
     75% 7.500000
   max 8.000000
[16 rows x 1 columns]

- `passing as_index` will leave the grouped column in-place (this is not change in 0.14.0)

In [39]: df = DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])
In [40]: g = df.groupby('A', as_index=False)
In [41]: g.count()
Out[41]:
   A B
0 1 1
1 5 2
In [42]: g.describe()
Out[42]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>count</td>
<td>2.000000</td>
</tr>
<tr>
<td></td>
<td>mean</td>
<td>4.000000</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>min</td>
<td>4.000000</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>4.000000</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>4.000000</td>
</tr>
<tr>
<td></td>
<td>75%</td>
<td>4.000000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>mean</td>
<td>7.000000</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>1.414214</td>
</tr>
<tr>
<td></td>
<td>min</td>
<td>6.000000</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>6.500000</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>7.000000</td>
</tr>
<tr>
<td></td>
<td>75%</td>
<td>7.500000</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>8.000000</td>
</tr>
</tbody>
</table>

[16 rows x 2 columns]

- Allow specification of a more complex groupby via `pd.Grouper`, such as grouping by a Time and a string field simultaneously. See the docs. (GH3794)

- Better propagation/preservation of Series names when performing groupby operations:
  - `SeriesGroupBy.agg` will ensure that the name attribute of the original series is propagated to the result (GH6265).
  - If the function provided to `GroupBy.apply` returns a named series, the name of the series will be kept as the name of the column index of the DataFrame returned by `GroupBy.apply` (GH6124). This facilitates `DataFrame.stack` operations where the name of the column index is used as the name of the inserted column containing the pivoted data.

### 1.8.5 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 sqlite3 database:

In [43]: `from sqlalchemy import create_engine`

# Create your connection.
In [44]: `engine = create_engine('sqlite:///:memory:');`

This engine can then be used to write or read data to/from this database:
In [45]: df = pd.DataFrame({'A': [1,2,3], 'B': ['a', 'b', 'c']})

In [46]: df.to_sql('db_table', engine, index=False)

You can read data from a database by specifying the table name:

In [47]: pd.read_sql_table('db_table', engine)
Out[47]:
   A  B
0  1  a
1  2  b
2  3  c

or by specifying a sql query:

In [48]: pd.read_sql_query('SELECT * FROM db_table', engine)
Out[48]:
   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).

### 1.8.6 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:

```python
df.loc[('A1', 'A3'), :]
```
rather than this:

```python
df.loc[('A1', 'A3')]
```

Warning: You will need to make sure that the selection axes are fully lexsorted!

```
In [49]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:

In [50]: index = MultiIndex.from_product([mklbl('A', 4),
   ....:     mklbl('B', 2),
   ....:     mklbl('C', 4),
   ....:     mklbl('D', 2)])
   ....:

In [51]: columns = MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'),
   ....:     ('b', 'foo'), ('b', 'bah')],
   ....:     names=['lvl0', 'lvl1'])
   ....:

In [52]: df = DataFrame(np.arange(len(index) * len(columns)).reshape((len(index), len(columns))),
   ....:     index=index,
   ....:     columns=columns).sortlevel().sortlevel(axis=1)
   ....:

In [53]: df
Out[53]:
   lvl0   a   b
  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
   D1 ... ... ... ...
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
```

Basic multi-index slicing using slices, lists, and labels.
In [54]: df.loc[(slice('A1','A3'),slice(None), ['C1','C3']),:]
Out[54]:
   lvl0  a  b
  lvl1  bar  foo  bah  foo
 A1  B0  C1  D0  73  72  75  74
     D1  77  76  79  78
     C3  D0  89  88  91  90
     D1  93  92  95  94
 B1  C1  D0  105 104 107 106
     D1  109 108 111 110
     C3  D0  121 120 123 122
     D1  125 124 128 127
... ... ... ... ...
 B3  B0  C1  D1  205 204 207 206
     C3  D0  249 248 251 250
     D1  253 252 255 254
[24 rows x 4 columns]

You can use a pd.IndexSlice to shortcut the creation of these slices

In [55]: idx = pd.IndexSlice

In [56]: df.loc[idx[:,:,['C1','C3']],idx[:,,'foo']]
Out[56]:
   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
     D1  60  62
... ... ... ... ...
 B3  B0  C1  D1  204 206
     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 [57]: df.loc['A1',(slice(None),'foo')]
In [58]: df.loc[idx[:,:,['C1','C3']],idx[:,'foo']]

Out[58]:
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 [59]: mask = df[('a','foo')]>200

In [60]: df.loc[idx[mask,:,['C1','C3']],idx[:,'foo']]

Out[60]:
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 [61]: df.loc(axis=0)[:,:,['C1','C3']]
Furthermore you can set the values using these methods:

```python
In [62]: df2 = df.copy()

In [63]: df2.loc(axis=0)[::, ['C1', 'C3']] = -10

In [64]: df2
Out[64]:
    lvl0  a   b
      lvl1    bar  foo  bah  foo
   A0 B0 C0 D0  1   0   3   2
      D1   5   4   7   6
   C1 D0 -10 -10 -10 -10
      D1 -10 -10 -10 -10
   C2 D0 17  16  19  18
      D1  21  20  23  22
   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.

```python
In [65]: df2 = df.copy()

In [66]: df2.loc[idx[::, ['C1', 'C3']], :] = df2*1000

In [67]: df2
Out[67]:
    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
      D1 -10 -10 -10 -10
   A3 B1 C0 D1 229 228 231 230
      D1 245 244 247 246
    C2 D0 -10 -10 -10 -10
      D1 -10 -10 -10 -10
```

[64 rows x 4 columns]
1.8.7 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.

1.8.8 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)

1.8.9 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 take 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)

```
# 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
Out[1]: 1
```

1.8. v0.14.0 (May 31, 2014)
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
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
Out[3]:
   3  1
   dtype: int64
# these are Float64Indexes, so integer or floating point is acceptable
In [4]: Series(1, np.arange(5.))[3]
Out[4]: 1
In [5]: Series(1, np.arange(5.))[3.0]
Out[6]: 1

• Numpy 1.9 compat w.r.t. deprecation warnings (GH6960)
  • Panel.shift() now has a function signature that matches DataFrame.shift(). The old positional argument lags has been changed to a keyword argument periods with a default value of 1. A FutureWarning is raised if the old argument lags is used by name. (GH6910)
  • The order keyword argument of factorize() will be removed. (GH6926).
  • Remove the copy keyword from DataFrame.xs(), Panel.major_xs(), Panel.minor_xs(). A view will be returned if possible, otherwise a copy will be made. Previously the user could think that copy=False would ALWAYS return a view. (GH6894)
  • 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 matpltolib Axes in a future release. You can use the future behavior now by passing return_type='axes' to boxplot.

1.8.10 Known Issues

• OpenPyXL 2.0.0 breaks backwards compatibility (GH7169)

1.8.11 Enhancements

• DataFrame and Series will create a MultiIndex object if passed a tuples dict, See the docs (GH3323)
In [68]: Series({('a', 'b'): 1, ('a', 'a'): 0, 
    ....:         ('a', 'c'): 2, ('b', 'a'): 3, ('b', 'b'): 4})

Out[68]:
   a   b
a  0
b  1
c  2
b a  3
b  4
dtype: int64

In [69]: 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[69]:
   a b
   a b c a b
A 4 1 5 8 10
C 3 2 6 7 NaN
D NaN NaN NaN NaN 9

• 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 [70]: 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 [71]: household

Out[71]:

   male  wealth
household_id
1     0       196087.3
2     1       316478.7
3     0       294750.0

In [72]: portfolio = DataFrame(dict(household_id = [1,2,2,3,3,3,4],
    ....:         asset_id = ["nl0000301109","nl0000289783","gb00b03m1x29", 
    ....:             "gb00b03m1x29","lu0197800237","nl0000289965",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 [73]: portfolio
Out[73]:

<table>
<thead>
<tr>
<th>name</th>
<th>share</th>
</tr>
</thead>
<tbody>
<tr>
<td>household_id asset_id</td>
<td></td>
</tr>
<tr>
<td>nl0000301109 1</td>
<td>1.00</td>
</tr>
<tr>
<td>n10000289783 2</td>
<td>0.40</td>
</tr>
<tr>
<td>gb00b03mlx29 3</td>
<td>0.60</td>
</tr>
<tr>
<td>gb00b03mlx29 3</td>
<td>0.15</td>
</tr>
<tr>
<td>lu0197800237 3</td>
<td>0.60</td>
</tr>
<tr>
<td>n10000289965 4</td>
<td>0.25</td>
</tr>
<tr>
<td>NaN 4</td>
<td>1.00</td>
</tr>
</tbody>
</table>

In [74]: household.join(portfolio, how='inner')
Out[74]:

<table>
<thead>
<tr>
<th>male</th>
<th>wealth</th>
<th>name</th>
<th>share</th>
</tr>
</thead>
<tbody>
<tr>
<td>household_id</td>
<td>asset_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n10000301109 1</td>
<td>196087.3</td>
<td>ABN Amro</td>
</tr>
<tr>
<td>2</td>
<td>n10000289783 1</td>
<td>316478.7</td>
<td>Robeco</td>
</tr>
<tr>
<td>3</td>
<td>gb00b03mlx29 3</td>
<td>316478.7</td>
<td>Royal Dutch Shell</td>
</tr>
<tr>
<td>3</td>
<td>gb00b03mlx29 3</td>
<td>294750.0</td>
<td>Royal Dutch Shell</td>
</tr>
<tr>
<td>4</td>
<td>lu0197800237 3</td>
<td>294750.0</td>
<td>AAB Eastern Europe Equity Fund</td>
</tr>
<tr>
<td>5</td>
<td>n10000289965 4</td>
<td>294750.0</td>
<td>Postbank BioTech Fonds</td>
</tr>
</tbody>
</table>

- `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 [75]: import datetime

In [76]: 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(2013,10,15,13,5),
                   datetime.datetime(2013,9,5,20,0), datetime.datetime(2013,11,2,10,0),
                   datetime.datetime(2013,10,7,20,0), datetime.datetime(2013,9,5,10,0)]})

In [77]: df
Out[77]:
   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 [78]: pivot_table(df, index=Grouper(freq='M', key='Date'),
                   columns=Grouper(freq='M', key='PayDay'),
                   values='Quantity', aggfunc=np.sum)

Out[78]:
   Date        NaN          1          9
   2013-09-30     3          NaN        NaN
   2013-10-31     1          NaN        NaN
   2013-11-30     1          NaN        NaN

- 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 [79]: prng = period_range('2013-01-01 09:00', periods=100, freq='H')

In [80]: ps = Series(np.random.randn(len(prng)), index=prng)

In [81]: ps
Out[81]:
   2013-01-01 09:00  0.755414
   2013-01-01 10:00  0.215269
   2013-01-01 11:00  0.841009
   2013-01-01 12:00 -1.445810
   2013-01-01 13:00 -1.401973
   2013-01-01 14:00 -0.100918

1.8. v0.14.0 (May 31, 2014)
2013-01-01 15:00  -0.548242
... 
2013-01-05 06:00  -0.379811 
2013-01-05 07:00   0.702562 
2013-01-05 08:00  -0.850346 
2013-01-05 09:00   1.176812 
2013-01-05 10:00  -0.524336 
2013-01-05 11:00  0.700908 
2013-01-05 12:00  0.984188 
Freq: H, dtype: float64

In [82]: ps['2013-01-02']
Out[82]:
2013-01-02 00:00  -0.208499
2013-01-02 01:00   1.033801
2013-01-02 02:00  -2.400454
2013-01-02 03:00   2.030604
2013-01-02 04:00  -1.142631
2013-01-02 05:00   0.211883
2013-01-02 06:00   0.704721
... 
2013-01-02 17:00   0.464392
2013-01-02 18:00  -3.563517
2013-01-02 19:00   1.321106
2013-01-02 20:00   0.152631
2013-01-02 21:00   0.164530
2013-01-02 22:00  -0.430096
2013-01-02 23:00   0.767369
Freq: H, 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)

1.8.12 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).

1.8.13 Experimental

There are no experimental changes in 0.14.0

1.8.14 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 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 no tests -A disabled) (GH6048).
• Bug in DataFrame.replace() when passing a nested dict that contained keys not in the values to be replaced (GH6342)
• str.match ignored the na flag (GH6609).
• Bug in take with duplicate columns that were not consolidated (GH6240)
• Bug in interpolate changing dtypes (GH6290)
• Bug in Series.get, was using a buggy access method (GH6383)
• Bug in hdfstore queries of the form where=[(‘date’, ‘>=’, datetime(2013,1,1)), (‘date’, ‘<=’, datetime(2014,1,1))] (GH6313)
• Bug in DataFrame.dropna with duplicate indices (GH6355)
• Regression in chained getitem indexing with embedded list-like from 0.12 (GH6394)
• Float64Index with nans not comparing correctly (GH6401)
• eval/query expressions with strings containing the @ character will now work (GH6366).
• Bug in Series.reindex when specifying a method with some nan values was inconsistent (noted on a resample) (GH6418)
• Bug in DataFrame.replace() where nested dicts were erroneously depending on the order of dictionary keys and values (GH5338).
• 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 (GH6678)
• Internal tests for patching __finalize__ / bug in merge not finalizing (GH6923, GH6927)
• accept TextFileReader in concat, which was affecting a common user idiom (GH6583)
• Bug in C parser with leading 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 stringndarray 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)

1.9 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:

```
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.ix[0,'A'] = np.nan

In [6]: df
Out[6]:
   A
0  NaN
1  bar
2  bah
3  foo
4  bar
```
### 1.9.1 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)

```python
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'>
Int64Index: 10 entries, 0 to 9
Data columns (total 3 columns):
A float64
B float64
C 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'>
Int64Index: 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.

```python
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
```
• 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>0</td>
<td>2001-01-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
</tr>
<tr>
<td>1</td>
<td>2004-06-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
</tr>
</tbody>
</table>
```

Now the output looks like:

```python
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
```

### 1.9.2 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:

```python
In [23]: s = Series(['a', 'a|b', np.nan, 'a|c'])
In [24]: s.str.get_dummies(sep='|')
Out[24]:
   a  b  c
 0  1  0  0
 1  1  1  0
 2  0  0  0
 3  1  0  1
```

- Added the `NDFrame.equals()` method to compare if two `NDFrames` are equal have equal axes, dtypes, and values. Added the `array_equivalent` function to compare if two ndarrays are equal. NaNs in identical locations are treated as equal. (GH5283) See also the docs for a motivating example.

```python
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())
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]))
Out[31]: False

• `DataFrame.apply` will use the `reduce` argument to determine whether a `Series` or a `DataFrame` should be returned when the `DataFrame` is empty (GH6007).

Previously, calling `DataFrame.apply` an empty `DataFrame` would return either a `DataFrame` if there were no columns, or the function being applied would be called with an empty `Series` to guess whether a `Series` or `DataFrame` should be returned:

```python
In [32]: def applied_func(col):
    ....:     print("Apply function being called with: ", col)
    ....:     return col.sum()
    ....:

In [33]: empty = DataFrame(columns=['a', 'b'])
In [34]: empty.apply(applied_func)
"Apply function being called with: ", Series([], dtype: float64)
Out[34]:
    a  NaN
    b  NaN
dtype: float64
```

Now, when `apply` is called on an empty `DataFrame`: if the `reduce` argument is `True` a `Series` will returned, if it is `False` a `DataFrame` will be returned, and if it is `None` (the default) the function being applied will be called with an empty series to try and guess the return type.

```python
In [35]: empty.apply(applied_func, reduce=True)
Out[35]:
    a  NaN
    b  NaN
dtype: float64

In [36]: empty.apply(applied_func, reduce=False)
Out[36]: Empty DataFrame
Columns: [a, b]
Index: []
[0 rows x 2 columns]
```

### 1.9.3 Prior Version Deprecations/Changes

There are no announced changes in 0.13 or prior that are taking effect as of 0.13.1
1.9.4 Deprecations

There are no deprecations of prior behavior in 0.13.1

1.9.5 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 @lexual for suggesting and @danbirken for rapidly implementing. (GH5490, GH6021)

  If `parse_dates` is enabled and this flag is set, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

  ```python
  # Try to infer the format for the index column
  df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
                   infer_datetime_format=True)
  ```

- `date_format` and `datetime_format` keywords can now be specified when writing to `excel` files (GH4133)

- `MultiIndex.from_product` convenience function for creating a MultiIndex from the cartesian product of a set of iterables (GH6055):

  ```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.

  ```python
  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']
  Out[43]:
   A   B    C    D
  2000-01-03  0.952 478 -1.239072 -1.409432 -0.014752
  2000-01-04  0.988138 0.139683 1.422986 1.272395
  2000-01-05 -0.072608 -0.223019 -2.147855 -1.449567
  2000-01-06 -0.550603 2.123692 -1.347533 -1.195524
  2000-01-07 -0.938153 0.122273 0.363565 -0.591863
  [5 rows x 4 columns]
Specifying an apply that operates on a Series (to return a single element)

```python
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

```python
In [45]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[45]:
       ItemA   ItemB   ItemC
A  0.379252 -3.696907  3.709335
B  0.923558  0.504242  4.656781
C -3.118269 -1.545718  3.188329
D -1.979310 -0.758060 -1.436483
[4 rows x 3 columns]
```

This is equivalent to

```python
In [46]: panel.sum('major_axis')
Out[46]:
       ItemA   ItemB   ItemC
A  0.379252 -3.696907  3.709335
B  0.923558  0.504242  4.656781
C -3.118269 -1.545718  3.188329
D -1.979310 -0.758060 -1.436483
[4 rows x 3 columns]
```

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 axis: A to D

In [49]: result['ItemA']
Out[49]:
        A       B       C       D
2000-01-03  1.004994 -1.166509 -0.535027  0.350970
2000-01-04  1.045875 -0.036892  1.393532  1.536326
2000-01-05 -0.718186  1.588611 -0.492880 -0.736422
2000-01-06 -1.162486 -0.051156  0.672185 -0.180500
[5 rows x 4 columns]

- 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 axis: ItemA to ItemC
```

```python
In [53]: result.loc[:,:,'ItemA']
```

Out[53]:
```
A   B   C   D
2000-01-03 0.116579 -0.667845 -1.151538 -0.157547
2000-01-04 0.650448 -1.114910 0.841527 0.760706
2000-01-05 -0.987433 -0.438897 -1.154468 -0.015033
2000-01-06 0.494000 1.060450 -0.775993 -1.140165
2000-01-07 -0.363770 0.013169 0.392036 -1.123913
```

[5 rows x 4 columns]

This is equivalent to the following

```python
In [54]: result = Panel(dict((ax,f(panel.loc[:,:ax]))
    ....:     for ax in panel.minor_axis ))

In [55]: result
```

Out[55]:
```
<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
```

```python
In [56]: result.loc[:,:,'ItemA']
```

Out[56]:
```
A   B   C   D
2000-01-03 0.116579 -0.667845 -1.151538 -0.157547
2000-01-04 0.650448 -1.114910 0.841527 0.760706
2000-01-05 -0.987433 -0.438897 -1.154468 -0.015033
2000-01-06 0.494000 1.060450 -0.775993 -1.140165
2000-01-07 -0.363770 0.013169 0.392036 -1.123913
```

[5 rows x 4 columns]

1.9.6 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)

1.9.7 Experimental

There are no experimental changes in 0.13.1

1.9.8 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.

1.10 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

 Their 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*

### 1.10.1 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 *truedivision*, 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
  ```
pandas: powerful Python data analysis toolkit, Release 0.16.2

2  1
dtype: int64

True Division

In [7]: pd.Series(arr) / pd.Series(arr2)  # no future import required
Out[7]:
0  0.200000
1  0.666667
2  1.500000
3  4.000000
dtype: float64

• Infer and downcast dtype if downcast='infer' is passed tofillna/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.

    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

1.10. v0.13.0 (January 3, 2014)
Traceback (most recent call last)
...
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_index,col_indexer] = value instead

Here is the correct method of assignment.

In [8]: dfc.loc[0, 'A'] = 11

In [9]: dfc
Out[9]:
   A  B
0  11  1
1  bbb  2
2  ccc  3
[3 rows x 2 columns]

- 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)

### 1.10.2 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)

### 1.10.3 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.
1.10.4 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
dtype: int64


In [13]: s
Out[13]:
0    1
1    2
2    3
5    5
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
```
A Panel setting operation on an arbitrary axis aligns the input to the Panel

```python
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']
Out[24]:
     Item1  Item2
2001-01-12    30    32
2001-01-13    30    32
2001-01-14    30    32
2001-01-15    30    32
[4 rows x 2 columns]
```

1.10.5 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.

```python
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
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.ix[3]
Out[30]: 2

In [31]: s.loc[3]
Out[31]: 2

The only positional indexing is via iloc

In [32]: s.iloc[3]
Out[32]: 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 [33]: s[2:4]
Out[33]:
  2   1
  3   2
dtype: int64

In [34]: s.ix[2:4]
Out[34]:
  2   1
  3   2
dtype: int64

In [35]: s.loc[2:4]
Out[35]:
  2   1
  3   2
dtype: int64

In [36]: s.iloc[2:4]
Out[36]:
  3.0  2
  4.5  3
dtype: int64

In float indexes, slicing using floats are allowed

In [37]: s[2.1:4.6]
Out[37]:
  3.0  2
  4.5  3
In [38]: s.loc[2.1:4.6]
Out[38]:
3.0  2
4.5  3
dtype: int64

• Indexing on other index types are preserved (and positional fallback for [], ix), with the exception, that floating
point slicing on indexes on non Float64Index will now raise a TypeError.

In [1]: Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type (Int64Index)

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

In [3]: Series(range(5))[3.0]
Out[3]: 3

1.10.6 HDFStore API Changes

• Query Format Changes. A much more string-like query format is now supported. See the docs.

In [39]: path = 'test.h5'

In [40]: dfq = DataFrame(randn(10, 4),
        columns=list('ABCD'),
        index=date_range('20130101', periods=10))

In [41]: dfq.to_hdf(path,'dfq',format='table',data_columns=True)

Use boolean expressions, with in-line function evaluation.

In [42]: read_hdf(path,'dfq',
        where="index>Timestamp('20130104') & columns=['A', 'B']")

Out[42]:
   A    B
0  2013-01-05  -1.392054  1.153922
1  2013-01-06   -0.881047  0.295080
2  2013-01-07    -1.407085  0.126781
3  2013-01-08    -0.838843  0.553921
4  2013-01-09    1.529401  0.205455
5  2013-01-10    0.299071  1.076541

[6 rows x 2 columns]

Use an inline column reference

In [43]: read_hdf(path,'dfq',
        where="A>0 or C>0")

Out[43]:
   A    B    C    D
2013-01-01  1.126386  0.247112  0.121172  0.298984
2013-01-03  0.581073  2.763844  0.399325  0.668488
2013-01-04 -0.275774  0.500483  0.863065 -1.051628
2013-01-05 -1.392054  1.153922  1.181944  0.391371
2013-01-06 -0.881047  0.295080  1.863801 -1.712274
2013-01-07 -1.407085  0.126781  0.003760 -1.268994
2013-01-09  1.529401  0.205455  0.313013  0.866521
2013-01-10  0.299071  1.076541  0.363177  1.893680

[8 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 [44]: path = 'test.h5'

In [45]: df = DataFrame(randn(10, 2))

In [46]: df.to_hdf(path, 'df_table', format='table')

In [47]: df.to_hdf(path, 'df_table2', append=True)

In [48]: df.to_hdf(path, 'df_fixed')

In [49]: with get_store(path) as store:
   ....:   print(store)
   ....:
<class 'pandas.io.pytables.HDFStore'>
/df_fixed frame (shape->[10,2])
/df_table frame_table (typ->appendable,nrows->10,ncols->2,indexers->[index])
/df_table2 frame_table (typ->appendable,nrows->10,ncols->2,indexers->[index])

• 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 [50]: path = 'test.h5'

In [51]: df = DataFrame(randn(10, 2))

In [52]: store1 = HDFStore(path)

In [53]: store2 = HDFStore(path)

In [54]: store1.append('df', df)

In [55]: store2.append('df2', df)
In [56]: store1
Out[56]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
/df    frame_table  (typ->appendable,nrows->10,ncols->2,indexers->[index])

In [57]: store2
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
/df    frame_table  (typ->appendable,nrows->10,ncols->2,indexers->[index])
/df2   frame_table  (typ->appendable,nrows->10,ncols->2,indexers->[index])

In [58]: store1.close()

In [59]: store2
Out[59]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
/df    frame_table  (typ->appendable,nrows->10,ncols->2,indexers->[index])
/df2   frame_table  (typ->appendable,nrows->10,ncols->2,indexers->[index])

In [60]: store2.close()

In [61]: store2
Out[61]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
File is CLOSED

• 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

1.10.7 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.
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')`.

### 1.10.8 Enhancements

- `df.to_clipboard()` learned a new keyword `excel` that lets 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 handling in `get_dummies` (GH4446) with `dummy_na`

```python
# previously, nan was erroneously counted as 2 here
# now it is not counted at all
In [62]: get_dummies([1, 2, np.nan])
Out[62]:
   1  2
0  1  0
1  0  1
2  0  0

[3 rows x 2 columns]
```

```python
# unless requested
In [63]: get_dummies([1, 2, np.nan], dummy_na=True)
Out[63]:
   1  2  NaN
0  1  0  0
1  0  1  0
2  0  0  1

[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 [64]: to_timedelta('1 days 06:05:01.00003')
Out[64]: Timedelta('1 days 06:05:01.000030')

In [65]: to_timedelta('15.5us')
Out[65]: Timedelta('0 days 00:00:00.000015')

In [66]: to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
Out[66]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT], dtype='timedelta64[ns]', freq=None)

In [67]: to_timedelta(np.arange(5), unit='s')
Out[67]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'], dtype='timedelta64[ns]', freq=None)

In [68]: to_timedelta(np.arange(5), unit='d')
Out[68]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype='timedelta64[ns]', freq=None)

A Series of dtype `timedelta64[ns]` can now be divided by another `timedelta64[ns]` object, or astyped to yield a `float64` dtyped Series. This is frequency conversion. See the docs for the docs.

In [69]: from datetime import timedelta

In [70]: td = Series(date_range('20130101', periods=4))-Series(date_range('20121201', periods=4))

In [71]: td[2] += timedelta(minutes=5, seconds=3)

In [72]: td[3] = np.nan

In [73]: td
Out[73]:
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 [74]: td / np.timedelta64(1, 'D')
Out[74]:
0  31.000000
1  31.000000
2  31.003507
3   NaN
dtype: float64

In [75]: td.astype('timedelta64[D]')
Out[75]:
0   31
1   31
2   31
3    NaN
dtype: float64

# to seconds
In [76]: td / np.timedelta64(1, 's')
Out[76]:
0 2678400
1 2678400
2 2678703
3 NaN
dtype: float64

In [77]: td.astype('timedelta64[s]')
Out[77]:
0 2678400
1 2678400
2 2678703
3 NaN
dtype: float64

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series

In [78]: td * -1
Out[78]:
0 -31 days +00:00:00
1 -31 days +00:00:00
2 -32 days +23:54:57
3 NaT
dtype: timedelta64[ns]

In [79]: td * Series([1,2,3,4])
Out[79]:
0 31 days 00:00:00
1 62 days 00:00:00
2 93 days 00:15:09
3 NaT
dtype: timedelta64[ns]

Absolute DateOffset objects can act equivalently to `timedeltas`

In [80]: from pandas import offsets
In [81]: td + offsets.Minute(5) + offsets.Milli(5)
Out[81]:
0 31 days 00:05:00.005000
1 31 days 00:05:00.005000
2 31 days 00:10:03.005000
3 NaT
dtype: timedelta64[ns]

Fillna is now supported for `timedeltas`

In [82]: td.fillna(0)
Out[82]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 0 days 00:00:00
dtype: timedelta64[ns]

In [83]: td.fillna(timedelta(days=1,seconds=5))
Out[83]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 1 days 00:00:05
dtype: timedelta64[ns]
You can do numeric reduction operations on timedeltas.

```
In [84]: td.mean()
Out[84]: Timedelta('31 days 00:01:41')
```

```
In [85]: td.quantile(.1)
Out[85]: Timedelta('31 days 00:00:00')
```

- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to `scipy.stats.gaussian_kde()` (for scipy >= 0.11.0) to set the bandwidth, and to `gkde.evaluate()` to specify the indices at which it is evaluated, respectively. See scipy docs. (GH4298)

- DataFrame constructor now accepts a numpy masked record array (GH3478)

- The new vectorized string method `extract` return regular expression matches more conveniently.

```
In [86]: Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)')
Out[86]:
   0  1
  0  a  1
  1  b  2
  2  NaN NaN
   dtype: object
```

Elements that do not match return NaN. Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [87]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
Out[87]:
   0  1
  0  a  1
  1  b  2
  2  NaN NaN
[3 rows x 2 columns]
```

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

```
In [88]: Series(['a1', 'b2', 'c3']).str.extract(
       ....:     '(?P<letter>[ab])(?P<digit>\d)')
       ....:
Out[88]:
   letter  digit
  0     a      1
  1     b      2
  2  NaN  NaN
[3 rows x 2 columns]
```

and optional groups can also be used.

```
In [89]: Series(['a1', 'b2', '3']).str.extract(
       ....:     '(?P<letter>[ab])?(?P<digit>\d)')
       ....:
Out[89]:
   letter  digit
  0     a      1
  1     b      2
  2  NaN  NaN
[3 rows x 2 columns]
```
• `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.

```python
In [90]: date_range('2013-01-01', periods=5, freq='5N')
Out[90]:
               '2013-01-01'],
dtype='datetime64[ns]', freq='5N', tz=None)
```

or with frequency as offset

```python
In [91]: date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
Out[91]:
               '2013-01-01'],
dtype='datetime64[ns]', freq='5N', tz=None)
```

Timestamps can be modified in the nanosecond range

```python
In [92]: t = Timestamp('20130101 09:01:02')
In [93]: t + pd.datetools.Nano(123)
Out[93]: 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:

```python
In [94]: dfi = DataFrame({'A': [1, 2, 3, 4], 'B': ['a', 'b', 'f', 'n']})
In [95]: dfi
Out[95]:
     A     B
0    1    a
1    2    b
2    3    f
3    4    n
```

```python
In [96]: other = DataFrame({'A': [1, 3, 3, 7], 'B': ['e', 'f', 'f', 'e']})
In [97]: mask = dfi.isin(other)
In [98]: mask
```
Out[98]:
   A   B
0  True False
1  False False
2  True  True
3  False False
[4 rows x 2 columns]

In [99]: dfi[mask.any(1)]
Out[99]:
   A   B
0 1  a
2 3  f
[2 rows x 2 columns]

• 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

    # 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)

In [100]: 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 [101]: df.interpolate()
Out[101]:
   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
[6 rows x 2 columns]
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 [102]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])
In [103]: ser.interpolate(limit=2)
Out[103]:
0   1
1   3
2   5
3   7
4  NaN
5  11
dtype: float64
```

• Added wide_to_long panel data convenience function. See the docs.

```python
In [107]: df
Out[107]:
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]
```

```python
In [108]: wide_to_long(df, ["A", "B"], i="id", j="year")
Out[108]:
X   A   B
id year
0  1970 -1.085631  a   2.5
1  1970  0.997345  b   1.2
2  1970  0.282978  c   0.7
0  1980 -1.085631  d   3.2
1  1980  0.997345  e   1.3
2  1980  0.282978  f   0.1

[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)

### 1.10.9 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,

  ```python
  In [109]: nrows, ncols = 20000, 100
  In [110]: df1, df2, df3, df4 = [DataFrame(randn(nrows, ncols))
                        for _ in range(4)]
  # eval with NumExpr backend
  In [111]: %timeit pd.eval('df1 + df2 + df3 + df4')
  100 loops, best of 3: 14 ms per loop
  # pure Python evaluation
  In [112]: %timeit df1 + df2 + df3 + df4
  10 loops, best of 3: 24.1 ms per loop
  
  For more details, see the [docs](https://pandas.pydata.org/docs/dev/index.html).

- Similar to `pandas.eval`, `DataFrame` has a new `DataFrame.eval` method that evaluates an expression in the context of the DataFrame. For example,

  ```python
  In [113]: df = DataFrame(randn(10, 2), columns=['a', 'b'])
  In [114]: df.eval('a + b')
  Out[114]:
   a   b
  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
dtype: float64
  
  • `query()` method has been added that allows you to select elements of a DataFrame using a natural query syntax nearly identical to Python syntax. For example,

    ```python
    In [115]: n = 20
    In [116]: df = DataFrame(np.random.randint(n, size=(n, 3)), columns=['a', 'b', 'c'])
    In [117]: df.query('a < b < c')
    Out[117]:
     a  b  c
    11 1  5  8
    15 8 16 19
    [2 rows x 3 columns]
    ```
selects all the rows of \( \text{df} \) 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.

```python
In [118]: df = DataFrame(np.random.rand(5,2),columns=list('AB'))

In [119]: df.to_msgpack('foo.msg')

In [120]: pd.read_msgpack('foo.msg')
Out[120]:
   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 [121]: s = Series(np.random.rand(5),index=date_range('20130101',periods=5))

In [122]: pd.to_msgpack('foo.msg', df, s)

In [123]: pd.read_msgpack('foo.msg')
Out[123]:
[  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, dtype: float64]

You can pass `iterator=True` to iterator over the unpacked results

```python
In [124]: 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, dtype: float64]
```
pandas: powerful Python data analysis toolkit, Release 0.16.2

2013-01-04  0.193225
2013-01-05  0.110977
Freq: D, dtype: float64

- pandas.io.gbq provides a simple way to extract from, and load data into, Google’s BigQuery DataSets 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

    from pandas.io import gbq

    # A query to select the average monthly temperatures in the
    # in the year 2000 across the USA. The dataset,
    # publicata:samples.gsod, is available on all BigQuery accounts,
    # and is based on NOAA gsod data.

    query = """SELECT station_number as STATION,
    month as MONTH, AVG(mean_temp) as MEAN_TEMP
    FROM publicdata:samples.gsod
    WHERE YEAR = 2000
    GROUP BY STATION, MONTH
    ORDER BY STATION, MONTH ASC"""

    # Fetch the result set for this query

    # Your Google BigQuery Project ID
    # To find this, see your dashboard:
    # https://code.google.com/apis/console/b/0/?noredirect
    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:

    > df3

    | MONTH | Min Tem | Mean Temp | Max Temp |
    |-------|---------|-----------|---------|
    | 1     | -53.336667 | 39.827892 | 89.770968 |
    | 2     | -49.837500 | 43.685219 | 93.437932 |
    | 3     | -77.926087 | 48.708355 | 96.099998 |
    | 4     | -82.892858 | 55.070087 | 97.317240 |
    | 5     | -92.378261 | 61.428117 | 102.042856 |
    | 6     | -77.703334 | 65.858888 | 102.900000 |
    | 7     | -87.821428 | 68.169663 | 106.510714 |
    | 8     | -89.431999 | 68.614215 | 105.500000 |
    | 9     | -86.611112 | 63.436935 | 107.142856 |
    | 10    | -78.209677 | 56.880838 | 92.103333 |
    | 11    | -50.125000 | 48.861228 | 94.996428 |
    | 12    | -50.332258 | 42.286879 | 94.396774 |
Warning: To use this module, you will need a BigQuery account. See <https://cloud.google.com/products/big-query> for details.
As of 10/10/13, there is a bug in Google’s API preventing result sets from being larger than 100,000 rows.
A patch is scheduled for the week of 10/14/13.

1.10.10 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)
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 [125]: s = Series([1, 2, 3, 4])
```

**Numpy Usage**

```python
In [126]: np.ones_like(s)
Out[126]: array([1, 1, 1, 1], dtype=int64)
In [127]: np.diff(s)
Out[127]: array([1, 1, 1], dtype=int64)
In [128]: np.where(s>1,s,np.nan)
Out[128]: array([ nan,  2.,  3.,  4.])
```

**Pandonic Usage**

```python
In [129]: Series(1, index=s.index)
Out[129]:
0 1
1 1
2 1
3 1
dtype: int64
In [130]: s.diff()
Out[130]:
0  NaN
1   1
2   1
3   1
dtype: float64
In [131]: s.where(s>1)
Out[131]:
0  NaN
1   2
2   3
3   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)
• __getattr__, __setattr__
• _indexed_same, reindex_like, align, where, mask
• fillna, replace (Series replace is now consistent with DataFrame)
• filter (also added axis argument to selectively filter on a different axis)
• reindex, reindex_axis, take
• truncate (moved to become part of NDFrame)

• These are API changes which make Panel more consistent with DataFrame
  – swapaxes on a Panel with the same axes specified now return a copy
  – support attribute access for setting
  – filter supports the same API as the original DataFrame filter
• Reindex called with no arguments will now return a copy of the input object
• TimeSeries is now an alias for Series. the property is_time_series can be used to distinguish (if desired)
• Refactor of Sparse objects to use BlockManager
  – Created a new block type in internals, SparseBlock, which can hold multi-dtypes and is non-consolidatable. SparseSeries and SparseDataFrame now inherit more methods from their hierarchy (Series/DataFrame), and no longer inherit from SparseArray (which instead is the object of the SparseBlock)
  – Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
  – Operations on sparse structures within DataFrames should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
  – enable setitem on SparseSeries for boolean/integer/slices
  – SparsePanels implementation is unchanged (e.g. not using BlockManager, needs work)
• added ftypes method to Series/DataFrame, similar to dtypes, but indicates if the underlying is sparse/dense (as well as the dtype)
• All NDFrame objects can now use __finalize__() to specify various values to propagate to new objects from an existing one (e.g. name in Series will follow more automatically now)
• Internal type checking is now done via a suite of generated classes, allowing isinstance(value, klass) without having to directly import the klass, courtesy of @jtratner
• Bug in Series update where the parent frame is not updating its cache based on changes (GH4080) or types (GH3217), fillna (GH3386)
• Indexing with dtype conversions fixed (GH4463, GH4204)
• Refactor Series.reindex to core/generic.py (GH4604, GH4618), allow method= in reindexing on a Series to work
• Series.copy no longer accepts the order parameter and is now consistent with NDFrame copy
• Refactor rename methods to core/generic.py; fixes Series.rename for (GH4605), and adds rename with the same signature for Panel
• Refactor clip methods to core/generic.py (GH4798)
• Refactor of _get_numeric_data/_get_bool_data to core/generic.py, allowing Series/Panel functionality
• Series (for index)/Panel (for items) now allow attribute access to its elements (GH1903)

In [132]: s = Series([1,2,3],index=list('abc'))
In [133]: s.b
Out[133]: 2
In [134]: s.a = 5
In [135]: s
Out[135]:
a  5
b  2
c  3
dtype: int64

1.10.11 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.

1.11 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.

1.11.1 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 correct a numpy bug that treats integer and float dtypes differently.

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
Out[3]:
   first  second
0     0      NaN
1     0      NaN
2     0      0

[3 rows x 2 columns]

In [4]: p / p
Out[4]:
   first  second
0     1      NaN
1     1      NaN
2     1      1

[3 rows x 2 columns]

In [5]: p / 0
Out[5]:
   first  second
0   inf      NaN
1   inf      NaN
2   inf      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).

```python
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, dtype: float64

# no squeezing (the default, and behavior in 0.10.1)
In [9]: df2.groupby("val1").apply(func)
Out[9]:
   val2  0  1  2  3
   val1
   1  0.5 -0.5  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 plenty of alternatives. This preserves the iloc API to be purely positional based.

```python
In [10]: df = DataFrame(lrange(5), list('ABCDE'), columns=['a'])
In [11]: mask = (df.a%2 == 0)
In [12]: mask
Out[12]:
   A  True
   B  False
   C  True
   D  False
   E  True
Name: a, dtype: bool

# this is what you should use
In [13]: df.loc[mask]
Out[13]:
   a
  A  0
  C  2
  E  4
[3 rows x 1 columns]
```
# this will work as well
In [14]: df.iloc[mask.values]
Out[14]:
   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

```python
from pandas.io.parsers import ExcelFile
xls = ExcelFile('path_to_file.xls')
xls.parse('Sheet1', index_col=None, na_values=['NA'])
```

With

```python
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

```python
from pandas.io.sql import read_frame
read_frame(...)
```

- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Do not allow 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
**1.11.2 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()
  In [18]: alist = pd.read_html(html, infer_types=True, index_col=0)
  In [19]: print(df == alist[0])
  a  b
  0 True True
  1 True True
  2 True True
  [3 rows x 2 columns]
  
  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*
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.

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',tupleize_cols=False)

In [23]: print(open('mi.csv').read())

In [24]: pd.read_csv('mi.csv',header=[0,1,2,3],index_col=[0,1],tupleize_cols=False)

- 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 1.392665 -0.123497
read_csv will now throw a more informative error message when a file contains no columns, e.g., all newline characters

1.11.3 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 [28]: df = DataFrame({'a': list('ab..'), 'b': [1, 2, 3, 4]})

In [29]: df.replace(regex=r'^\s*\.$', value=np.nan)
Out[29]:
    a  b
0  a  1
1  b  2
2  NaN 3
3  NaN 4

[4 rows x 2 columns]

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 [30]: df.replace('.', np.nan)
Out[30]:
    a  b
0  a  1
1  b  2
2  NaN 3
3  NaN 4

[4 rows x 2 columns]

to replace all occurrences of the string ‘.’ with NaN.
• **pd.melt()** now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame.

• **pd.set_option()** now allows N option, value pairs (GH3667).

  Let’s say that we had an option ‘a.b’ and another option ‘b.c’. We can set them at the same time:

  ```
  In [31]: pd.get_option('a.b')
  Out[31]: 2

  In [32]: pd.get_option('b.c')
  Out[32]: 3

  In [33]: pd.set_option('a.b', 1, 'b.c', 4)

  In [34]: pd.get_option('a.b')
  Out[34]: 1

  In [35]: pd.get_option('b.c')
  Out[35]: 4
  ```

• The **filter** method for group objects returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

  ```
  In [36]: sf = Series([1, 1, 2, 3, 3, 3])

  In [37]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
  Out[37]:
  3 3
  4 3
  5 3
  dtype: int64
  ```

  The argument of **filter** must a function that, applied to the group as a whole, returns True or False.

  Another useful operation is filtering out elements that belong to groups with only a couple members.

  ```
  In [38]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

  In [39]: dff.groupby('B').filter(lambda x: len(x) > 2)
  Out[39]:
   A B
  2 2 b
  3 3 b
  4 4 b
  5 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.

  ```
  In [40]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
  Out[40]:
   A  B
  0 NaN NaN
  1 NaN NaN
  2 2 b
  3 3 b
  4 4 b
  ```

1.11. v0.12.0 (July 24, 2013)
5 5 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 datetime.min and datetime.max (respectively), thanks @SleepingPills
- read_html now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

### 1.11.4 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 [41]: from pandas.tseries.offsets import CustomBusinessDay
In [42]: from datetime import datetime

# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [43]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
# add that for a couple of years
In [44]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]
In [45]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)
In [46]: dt = datetime(2013, 4, 30)
In [47]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00
In [48]: dts = date_range(dt, periods=5, freq=bday_egypt)
In [49]: 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, dtype: object
```

### 1.11.5 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,

```python
In [50]: strs = 'go', 'bow', 'joe', 'slow'
In [51]: ds = Series(strs)
In [52]: for s in ds.str:
   ....:     print(s)
   ....:
0  g
1  b
2  j
3  s
dtype: object
0  o
1  o
2  o
3  l
dtype: object
0  NaN
1  w
2  e
3  o
dtype: object
0  NaN
1  NaN
2  NaN
3  w
dtype: object
```

```python
In [53]: s
Out[53]:
0  NaN
1  NaN
2  NaN
3  w
dtype: object
```

```python
In [54]: s.dropna().values.item() == 'w'
Out[54]: True
```

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.iterables() now works with frames with duplicate column names (GH3873)
  - Bug in non-unique indexing via iloc (GH4017); added takeable argument to reindex for location-based taking
  - Allow non-unique indexing in series via .ix/.loc and __getitem__ (GH4246)
  - Fixed non-unique indexing memory allocation issue with .ix/.loc (GH4280)
- DataFrame.from_records did not accept empty recarrays (GH3682)
- read_html now correctly skips tests (GH3741)
- Fixed a bug where DataFrame.replace with a compiled regular expression in the to_replace argument
  wasn’t working (GH3907)
- Improved network test decorator to catch IOError (and therefore URLError as well). Added
  with_connectivity_check decorator to allow explicitly checking a website as a proxy for seeing if there
  is network connectivity. Plus, new optional_args decorator factory for decorators. (GH3910, GH3914)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982,
  GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed
  (GH3982, GH3985, GH4028, GH4054)
- Series.hist will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of tox under python3 where the pickle import was getting rewritten in an incompatible way
  (GH4062, GH4063)
- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fixed bug in DataFrame.replace where a nested dict wasn’t being iterated over when regex=False
  (GH4115)
- Fixed bug in the parsing of microseconds when using the format argument in to_datetime (GH4152)
pandas: powerful Python data analysis toolkit, Release 0.16.2

• 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.

1.12 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

1.12.1 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 indices are out of bounds. Allowed inputs are:
  – An integer e.g. 5
  – A list or array of integers [4, 3, 0]
  – A slice object with ints 1:7
  – A boolean array

See more at Selection by Position
.ix supports mixed integer and label based access. It is primarily label based, but will fallback to integer positional access. .ix is the most general and will support any of the inputs to .loc and .iloc, as well as support for floating point label schemes. .ix is especially useful when dealing with mixed positional and label based hierarchial 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.

1.12.2 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.

1.12.3 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  0.245972
1  0.319442
2  1.378512
3  0.292502
4  0.329791
5  1.392047
6  0.769914
7 -2.472300

[8 rows x 1 columns]

In [3]: df1.dtypes
Out[3]:
   A   float32
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 -0.611328 -0.270630  255
In 

```
In [6]: df2.dtypes
Out[6]:
A float16
B float64
C uint8
dtype: object
```

# here you get some upcasting

```
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```

In 

```
In [8]: df3
Out[8]:
   A       B       C
0  0.365356  0.270630  255
1  1.364364  1.685677    0
2  2.882418  0.440747    0
3  1.035623  0.115070    1
4  1.354205  0.632102    0
5  2.052203  0.585977    0
6  2.006243  1.444787    0
7  4.642221  0.201135    0
```

```
In [9]: df3.dtypes
Out[9]:
A float32
B float32
C float32
dtype: object
```

### 1.12.4 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')
```

Conversion

```
In [11]: df3.astype('float32').dtypes
Out[11]:
A float32
B float32
C float32
dtype: object
```

Mixed Conversion
In [12]: df3['D'] = '1.'

In [13]: df3['E'] = '1'

In [14]: df3.convert_objects(convert_numeric=True).dtypes
   Out[14]:
          A float32
          B float64
          C float64
          D float64
          E int64
dtype: object

# same, but specific dtype conversion
In [15]: df3['D'] = df3['D'].astype('float16')
In [16]: df3['E'] = df3['E'].astype('int32')

In [17]: df3.dtypes
   Out[17]:
          A float32
          B float64
          C float64
          D float16
          E int32
dtype: object

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
dtype: datetime64[ns]

1.12.5 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

148 Chapter 1. What’s New
dtype: object

In [22]: DataFrame({'a' : [1,2]}).dtypes
Out[22]:
da int64
dtype: object

In [23]: DataFrame({'a' : 1}, index=range(2)).dtypes
Out[23]:
a int64
dtype: object

Keep in mind that DataFrame(np.array([1,2])) WILL result in int32 on 32-bit platforms!

Upcasting Gotchas

Performing indexing operations on integer type data can easily upcast the data. The dtype of the input data will be preserved in cases where nans are not introduced.

In [24]: dfi = df3.astype('int32')

In [25]: dfi['D'] = dfi['D'].astype('int64')

In [26]: dfi
Out[26]:
   A  B  C  D  E
0  0  0  255  1  1
1  1 -1   0  1  1
2  2  0   0  1  1
3 -1  0  1  1  1
4  1  0   0  1  1
5  2  0   0  1  1
6  2 -1   0  1  1
7 -4  0   0  1  1

[8 rows x 5 columns]

In [27]: dfi.dtypes
Out[27]:
A     int32
B     int32
C     int32
D     int64
E     int32
dtype: object

In [28]: casted = dfi[dfi>0]

In [29]: casted
Out[29]:
   A  B  C  D  E
0  NaN NaN  255  1  1
1  NaN NaN   1  1  1
2  NaN NaN   1  1  1
3  NaN NaN   1  1  1
4  NaN NaN   1  1  1
5  NaN NaN   1  1  1
6  NaN NaN   1  1  1
7  NaN NaN   1  1  1
[8 rows x 5 columns]

In [30]: casted.dtypes
Out[30]:
A  float64
B  float64
C  float64
D  int64
E  int32

dtype: object

While float dtypes are unchanged.

In [31]: df4 = df3.copy()

In [32]: df4['A'] = df4['A'].astype('float32')

In [33]: df4.dtypes
Out[33]:
A  float32
B  float64
C  float64
D  float16
E  int32

dtype: object

In [34]: casted = df4[df4>0]

In [35]: casted
Out[35]:
     A   B    C    D   E
0  NaN  NaN  255   1   1
1  1.364364  NaN  NaN   1   1
2  2.882418  NaN  NaN   1   1
3  NaN   NaN   1   1   1
4  1.354205  NaN  NaN   1   1
5  2.052203  NaN  NaN   1   1
6  2.006243  NaN  NaN   1   1
7  NaN   NaN   NaN   1   1
[8 rows x 5 columns]

In [36]: casted.dtypes
Out[36]:
A  float32
B  float64
C  float64
D  float16
E  int32

dtype: object

1.12.6 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
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  2001-01-02 -1.448835  0.153437  2001-01-03
1  2001-01-03 -1.123570 -0.791498  2001-01-03
2  2001-01-04  0.105400  1.262401  2001-01-03
3  2001-01-05 -0.721844 -0.647645  2001-01-03
4  2001-01-06 -0.830631  0.761823  2001-01-03
5  2001-01-07  0.597819  1.045558  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
dtype: int64

# use the traditional nan, which is mapped to NaT internally
In [41]: df.ix[2:4,['A','timestamp']] = np.nan

In [42]: df
Out[42]:
   A       B    timestamp
0  2001-01-02 -1.448835  0.153437  2001-01-03
1  2001-01-03 -1.123570 -0.791498  2001-01-03
2  2001-01-04 NaN    1.262401        NaT
3  2001-01-05 NaN    -0.647645        NaT
4  2001-01-06 -0.830631  0.761823  2001-01-03
5  2001-01-07  0.597819  1.045558  2001-01-03

[6 rows x 3 columns]

Astype conversion on datetime64[ns] to object, implicity 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
dtype: datetime64[ns]

In [48]: s.dtype
Out[48]: dtype('<M8[ns]')

In [49]: s = s.astype('O')

In [50]: s
Out[50]:
0  2001-01-02 00:00:00
1  NaN
2  2001-01-02 00:00:00
   dtype: object

In [51]: s.dtype
Out[51]: dtype('O')

1.12.7 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

1.12.8 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

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 support iteration on select and select_as_multiple (GH3076)

- You can now select timestamps from an unordered timeseries similarly to an ordered timeseries (GH2437)
- You can now select with a string from a DataFrame with a datelike index, in a similar way to a Series (GH3070)
In [55]: idx = date_range("2001-10-1", periods=5, freq='M')

In [56]: ts = Series(np.random.rand(len(idx)),index=idx)

In [57]: ts['2001']
Out[57]:
2001-10-31  0.483450
2001-11-30  0.407530
2001-12-31  0.965096
Freq: M, dtype: float64

In [58]: df = DataFrame(dict(A = ts))

In [59]: df['2001']
Out[59]:
    A
2001-10-31  0.483450  
2001-11-30  0.407530  
2001-12-31  0.965096  

[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
Out[61]:
<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 [62]: p.reindex(items=['ItemA']).squeeze()
Out[62]:
           A   B   C   D
2001-01-02  0.396537  0.534880 -0.488797 -1.539385
2001-01-03  0.829037  0.306681  0.331032  1.544977
2001-01-04 -0.621754  1.026208  0.413106 -1.490869
2001-01-05 -1.253235 -0.538879 -1.487449 -1.426475

[4 rows x 4 columns]

In [63]: p.reindex(items=['ItemA'],minor=['B']).squeeze()
Out[63]:
     A   B
2001-01-02  0.534880   
2001-01-03  0.306681   
2001-01-04  1.026208   
2001-01-05 -0.538879   
Freq: D, Name: B, dtype: float64

- In `pd.io.data.Options`,
  - Fix bug when trying to fetch data for the current month when already past expiry.
— Now using lxml to scrape html instead of BeautifulSoup (lxml was faster).

— New instance variables for calls and puts are automatically created when a method that creates them is called. This works for current month where the instance variables are simply `calls` and `puts`. Also works for future expiry months and save the instance variable as `callsMMYY` or `putsMMYY`, where `MMYY` are, respectively, the month and year of the option’s expiry.

— `Options.get_near_stock_price` now allows the user to specify the month for which to get relevant options data.

— `Options.get_forward_data` now has optional kwargs `near` and `above_below`. This allows the user to specify if they would like to only return forward looking data for options near the current stock price. This just obtains the data from `Options.get_near_stock_price` instead of `Options.get_xxx_data()` (GH2758).

- Cursor coordinate information is now displayed in time-series plots.
- added option `display.max_seq_items` to control the number of elements printed per sequence pprinting it. (GH2979)
- added option `display.chop_threshold` to control display of small numerical values. (GH2739)
- added option `display.max_info_rows` to prevent verbose_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- `value_counts()` now accepts a “normalize” argument, for normalized histograms. (GH2710).
- `DataFrame.from_records` now accepts not only dicts but any instance of the collections.Mapping ABC.
- added option `display.mpl_style` providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- `to_html()` now accepts an optional “escape” argument to control reserved HTML character escaping (enabled by default) and escapes &, in addition to `< and >`. (GH2919)

See the full release notes or issue tracker on GitHub for a complete list.

### 1.13 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.

#### 1.13.1 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)
1.13.2 New features

- MySQL support for database (contribution from Dan Allan)

1.13.3 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.ix[4:6,'string'] = np.nan

In [5]: df.ix[7:9,'string'] = 'bar'

In [6]: df['string2'] = 'cool'

In [7]: df
Out[7]:
A    B    C  string  string2
2000-01-01 1.601262 0.256718 0.239369    foo       cool
2000-01-02 -0.174122 -1.131794 -1.948006    foo       cool
2000-01-03  0.980347 -0.674429 -0.361633    foo       cool
2000-01-04 -0.761218  1.768215  0.152288    foo       cool
2000-01-05 -0.862613 -0.210968 -0.859278    NaN       cool
2000-01-06  1.498195  0.462413 -0.647604    NaN       cool
2000-01-07  1.511487 -0.727189 -0.342928    foo       cool
2000-01-08 -0.007364  1.427674  0.104020    bar       cool
[8 rows x 5 columns]
```

# on-disk operations

```python
In [8]: store.append('df', df, data_columns=['B','C','string','string2'])

In [9]: store.select('df', ['B > 0', 'string == foo'])
Out[9]:
A    B    C  string  string2
2000-01-04 -0.761218  1.768215  0.152288    foo       cool
[1 rows x 5 columns]
```

# this is in-memory version of this type of selection

```python
In [10]: df[(df.B > 0) & (df.string == 'foo')]  
Out[10]:
A    B    C  string  string2
2000-01-04 -0.761218  1.768215  0.152288    foo       cool
[1 rows x 5 columns]
```

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

```python
In [11]: df_mixed = df.copy()
In [12]: df_mixed['datetime64'] = Timestamp('20010102')
In [13]: df_mixed.ix[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 -1.601262 -0.256718    foo      cool   2001-01-02
1 2000-01-02  0.174122 -1.131794    foo      cool   2001-01-02
2 2000-01-03  0.980347 -0.674429    foo      cool   2001-01-02
3 2000-01-04   NaN        NaN       NaN       NaN   2001-01-02
4 2000-01-05 -0.862613 -0.210968    NaN      cool   2001-01-02
5 2000-01-06  1.498195  0.462413   -0.647604    NaN       NaN   2001-01-02
6 2000-01-07  1.511487 -0.727189   -0.342928    NaN       NaN   2001-01-02
7 2000-01-08 -0.007364  1.427674  0.104020    bar      cool   2001-01-02
[8 rows x 6 columns]
```

```python
In [17]: df_mixed1.get_dtype_counts()
Out[17]:
 datetime64[ns]    1
    float64           3
      object           2
dtype: int64
```

You can pass `columns` keyword to select to filter a list of the return columns, this is equivalent to passing a `Term('columns', list_of_columns_to_filter)`

```python
In [18]: store.select('df', columns = ['A', 'B'])
Out[18]:
   A        B
0 2000-01-01 -1.601262 -0.256718
1 2000-01-02  0.174122 -1.131794
2 2000-01-03  0.980347 -0.674429
3 2000-01-04 -0.761218  1.768215
4 2000-01-05 -0.862613 -0.210968
5 2000-01-06  1.498195  0.462413
6 2000-01-07  1.511487 -0.727189
7 2000-01-08 -0.007364  1.427674
[8 rows x 2 columns]
```

HDFStore now serializes multi-index dataframes when appending tables.

```python
In [19]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'gux'],
                                   ['one', 'two', 'three']],
                      index.names=['first', 'second'])
```
In [20]: df = DataFrame(np.random.randn(10, 3), index=index,
                 columns=['A', 'B', 'C'])

In [21]: df
Out[21]:
   A       B      C
foo one  2.052171 -1.230963 -0.019240
two    -1.713238  0.838912  -0.637855
three   0.215109 -1.515362   1.586924
bar one -0.447974 -1.573998   0.630925
two    -0.071659 -1.277640  -0.102206
baz two  0.870302  1.275280  -1.199212
three   1.060780  1.673018   1.249874
qux one  1.458210 -0.710542   0.825392
two     1.557329  1.993441  -0.616293
three   0.150468  0.132104   0.580923

[10 rows x 3 columns]

In [22]: store.append('mi', df)

In [23]: store.select('mi')
Out[23]:
   A       B      C
foo one  2.052171 -1.230963 -0.019240
two    -1.713238  0.838912  -0.637855
three   0.215109 -1.515362   1.586924
bar one -0.447974 -1.573998   0.630925
two    -0.071659 -1.277640  -0.102206
baz two  0.870302  1.275280  -1.199212
three   1.060780  1.673018   1.249874
qux one  1.458210 -0.710542   0.825392
two     1.557329  1.993441  -0.616293
three   0.150468  0.132104   0.580923

[10 rows x 3 columns]

# the levels are automatically included as data columns

In [24]: store.select('mi', Term('foo=bar'))
Out[24]:
   A       B      C
foo one  2.052171 -1.230963 -0.019240
two    -1.713238  0.838912  -0.637855
three   0.215109 -1.515362   1.586924
bar one -0.447974 -1.573998   0.630925
two    -0.071659 -1.277640  -0.102206
baz two  0.870302  1.275280  -1.199212
three   1.060780  1.673018   1.249874
qux one  1.458210 -0.710542   0.825392
two     1.557329  1.993441  -0.616293
three   0.150468  0.132104   0.580923

[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.
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
/df frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,string,string2])
/df1_mt frame_table (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_mixed frame_table (typ->appendable,nrows->8,ncols->6,indexers->[index])
/mi frame_table (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->[bar,foo])

# individual tables were created
In [29]: store.select('df1_mt')
Out[29]:
A   B
2000-01-01 -0.128750 1.445964
2000-01-02 -0.688741 0.228006
2000-01-03 0.932498 -2.200069
2000-01-04 1.298390 1.662964
2000-01-05 -0.462446 -0.112019
2000-01-06 -1.626124 0.982041
2000-01-07 0.942864 2.502156
2000-01-08 0.268766 -1.225092
[8 rows x 2 columns]

In [30]: store.select('df2_mt')
Out[30]:
C   D   E   F   foo
2000-01-01 -0.431163 0.016640 0.904578 -1.645852 bar
2000-01-02 0.800353 -0.451572 0.831767 0.228760 bar
2000-01-03 1.239198 0.185437 -0.540770 -0.370038 bar
2000-01-04 -0.040863 0.290110 -0.096145 1.717830 bar
2000-01-05 -0.134024 -0.205969 1.348944 -1.198246 bar
2000-01-06 0.059493 -0.460111 -1.565401 -0.025706 bar
2000-01-07 -0.302741 0.261551 -0.066342 0.897097 bar
2000-01-08 0.582752 -1.490764 -0.639757 -0.952750 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')
Out[31]:
A   B   C   D   E   F   foo
2000-01-04 1.298390 1.662964 -0.040863 0.290110 -0.096145 1.717830 bar
2000-01-07 0.942864 2.502156 -0.302741 0.261551 -0.066342 0.897097 bar
[2 rows x 7 columns]

Enhancements
• **HDFStore** now can read native PyTables table format tables

• You can pass `nan_rep = 'my_nan_rep'` to append, to change the default nan representation on disk (which converts to/from np.nan), this defaults to `nan`.

• You can pass `index` to append. This defaults to `True`. This will automagically create indicies 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 expected. This will optimize read/write performance.

• **Select** now supports passing `start` and `stop` to provide selection space limiting in selection.

• Greatly improved ISO8601 (e.g., yyyy-mm-dd) date parsing for file parsers (GH2698)

• Allow DataFrame.merge to handle combinatorial sizes too large for 64-bit integer (GH2690)

• Series now has unary negation (`-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.

1.14 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.

1.14.1 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

1.14.2 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:

```python
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 -0.892402 0.505987 -0.681624 0.850162
2000-01-02  0.586586 1.175843 -0.160391 0.481679
2000-01-03  0.408279 1.641246  0.383888 -1.495227
```

Chapter 1. What’s New
pandas: powerful Python data analysis toolkit, Release 0.16.2

2000-01-04  1.166096 -0.802272 -0.275253  0.517938
2000-01-05  -0.750872  1.216537 -0.910343 -0.606534
2000-01-06  -0.410659  0.264024 -0.069315 -1.814768

[6 rows x 4 columns]

# deprecated now
In [4]: df - df[0]
Out[4]:
0   1   2   3
2000-01-01  0  1.398389  0.210778  1.742564
2000-01-02  0  0.589256 -0.746978 -0.104908
2000-01-03  0  1.232968 -0.024391 -1.903505
2000-01-04  0 -1.968368 -1.441350 -0.648158
2000-01-05  0  1.967410 -0.159471  0.144338
2000-01-06  0  0.674682  0.341344 -1.404109

[6 rows x 4 columns]

# Change your code to
In [5]: df.sub(df[0], axis=0) # align on axis 0 (rows)
Out[5]:
0   1   2   3
2000-01-01  0  1.398389  0.210778  1.742564
2000-01-02  0  0.589256 -0.746978 -0.104908
2000-01-03  0  1.232968 -0.024391 -1.903505
2000-01-04  0 -1.968368 -1.441350 -0.648158
2000-01-05  0  1.967410 -0.159471  0.144338
2000-01-06  0  0.674682  0.341344 -1.404109

[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).

Note:
In [6]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')
In [7]: series = Series(np.arange(len(dates)), index=dates)
In [8]: series
Out[8]:
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
...  2000-01-04  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: int32

In [9]: series.resample('D', how='sum')
Out[9]:
2000-01-01 15
2000-01-02 51
2000-01-03 87
2000-01-04 123
2000-01-05 24
Freq: D, dtype: int32

# old behavior
In [10]: series.resample('D', how='sum', closed='right', label='right')
Out[10]:
2000-01-01 0
2000-01-02 21
2000-01-03 57
2000-01-04 93
2000-01-05 129
Freq: D, dtype: int32

- Infinity and negative infinity are no longer treated as NA by isnull and notnull. That they every were was a relic of early pandas. This behavior can be re-enabled globally by the mode.use_inf_as_null option:

In [11]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])

In [12]: pd.isnull(s)
Out[12]:
0 False
1 False
2 False
3 False
dtype: bool

In [13]: s.fillna(0)
Out[13]:
0  1.500000
1   inf
2  3.400000
3  -inf
dtype: float64

In [14]: pd.set_option('use_inf_as_null', True)

In [15]: pd.isnull(s)
Out[15]:
0  False
1  True
2  False
3  True
dtype: bool
In [16]: s.fillna(0)
Out[16]:
  0 1.5
  1 0.0
  2 3.4
  3 0.0
dtype: float64

In [17]: pd.reset_option('use_inf_as_null')

- Methods with the `inplace` option now all return None instead of the calling object. E.g. code written like `df = df.fillna(0, inplace=True)` may stop working. To fix, simply delete the unnecessary variable assignment.

- `pandas.merge` no longer sorts the group keys (sort=False) by default. This was done for performance reasons: the group-key sorting is often one of the more expensive parts of the computation and is often unnecessary.

- The default column names for a file with no header have been changed to the integers 0 through N - 1. This is to create consistency with the DataFrame constructor with no columns specified. The v0.9.0 behavior (names X0, X1, ...) can be reproduced by specifying `prefix='X'`:

In [18]: data = 'a,b,c
   ...: 1,Yes,2
   ...: 3,No,4'

In [19]: print(data)
a,b,c
1,Yes,2
3,No,4

In [20]: pd.read_csv(StringIO(data), header=None)
Out[20]:
     0  1  2
 0   a  b  c
 1   1  Yes 2
 2   3   No 4
[3 rows x 3 columns]

In [21]: pd.read_csv(StringIO(data), header=None, prefix='X')
Out[21]:
    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 [22]: print(data)
a,b,c
1,Yes,2
3,No,4

In [23]: pd.read_csv(StringIO(data))
Out[23]:
     a  b  c
 0   1  Yes  2
In [24]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[24]:
   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 [25]: s = Series([np.nan, 1., 2., np.nan, 4])

In [26]: s
Out[26]:
0   NaN
1    1
2    2
3   NaN
4    4
dtype: float64

In [27]: s.fillna(0)
Out[27]:
0    0
1    1
2    2
3    0
4    4
dtype: float64

In [28]: s.fillna(method='pad')
Out[28]:
0   NaN
1    1
2    2
3    2
4    4
dtype: float64

Convenience methods `ffill` and `bfill` have been added:

In [29]: s.ffill()
Out[29]:
0   NaN
1    1
2    2
3    2
4    4
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

```python
In [30]: def f(x):
    ....:     return Series([ x, x**2 ], index = ['x', 'x^2'])
    ....:

In [31]: s = Series(np.random.rand(5))

In [32]: s
Out[32]:
0   0.013135
1   0.909855
2   0.098093
3   0.023540
4   0.141354
dtype: float64

In [33]: s.apply(f)
Out[33]:
     x  x^2
0  0.013135  0.000173
1  0.909855  0.827836
2  0.098093  0.009622
3  0.023540  0.000554
4  0.141354  0.019981
```

• 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 [34]: get_option("display.max_rows")
Out[34]: 15
```

• to_string() methods now always return unicode strings (GH2224).

### 1.14.3 New features

#### 1.14.4 Wide DataFrame Printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```python
In [35]: wide_frame = DataFrame(randn(5, 16))

In [36]: wide_frame
Out[36]:
   0      1      2      3      4      5      6
0 2.520045 1.570114 -0.360875 -0.880096 0.235532 0.207232 -1.983857
1 0.422194 0.288403 -0.487393 -0.777639 0.055865 1.383381 0.085638
2 0.585174 -0.568825 -0.719412 1.191340 -0.456362 0.089931 0.776079
```
The old behavior of printing out summary information can be achieved via the 'expand_frame_repr' print option:

In [37]: pd.set_option('expand_frame_repr', False)

In [38]: wide_frame

Out[38]:

[5 rows x 16 columns]

The width of each line can be changed via ‘line_width’ (80 by default):

In [39]: pd.set_option('line_width', 40)

line_width has been deprecated, use display.width instead (currently both are identical)

In [40]: wide_frame

Out[40]:

[5 rows x 16 columns]
1.14.5 Updated PyTables Support

Docs for PyTables Table format & several enhancements to the api. Here is a taste of what to expect.

In [41]: store = HDFStore('store.h5')

In [42]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
                           columns=['A', 'B', 'C'])

In [43]: df

Out[43]:
     A          B          C
2000-01-01 -2.036047  0.000830 -0.955697
2000-01-02 -0.898872 -0.725411  0.059904
2000-01-03 -0.449644  1.082900 -1.221265
2000-01-04  0.361078  1.330704  0.855932
2000-01-05 -1.216718  1.488887  0.018993
2000-01-06 -0.877046  0.045976  0.437274
2000-01-07 -0.567182 -0.886657 -0.556383
2000-01-08  0.655457  1.117949 -2.782376

[8 rows x 3 columns]

# appending data frames
In [44]: df1 = df[0:4]

In [45]: df2 = df[4:]
In [46]: store.append('df', df1)

In [47]: store.append('df', df2)

In [48]: store
Out[48]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable, nrows->8, ncols->3, indexers->[index])

# selecting the entire store
In [49]: store.select('df')
Out[49]:
   A    B    C
2000-01-01 -2.036047 0.000830 -0.955697
2000-01-02 -0.898872 -0.725411 0.059904
2000-01-03 -0.449644 1.082900 -1.221265
2000-01-04 0.361078 1.330704 0.855932
2000-01-05 -1.216718 1.488887 0.018993
2000-01-06 -0.877046 0.045976 0.437274
2000-01-07 -0.567182 -0.888657 -0.556383
2000-01-08 0.655457 1.117949 -2.782376
[8 rows x 3 columns]

In [50]: 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 [51]: wp
Out[51]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

# storing a panel
In [52]: store.append('wp', wp)

# selecting via A QUERY
In [53]: store.select('wp',
                    [ Term('major_axis>20000102'), Term('minor_axis', '=', ['A','B']) ]
                )
Out[53]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B

# removing data from tables
In [54]: store.remove('wp', Term('major_axis>20000103'))
Out[54]:

In [55]: store.select('wp')
Out[55]:

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to D

# deleting a store
In [56]: del store['df']

In [57]: store
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp  wide_table  (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])

Enhancements
• added ability to hierarchical keys

In [58]: store.put('foo/bar/bah', df)

In [59]: store.append('food/orange', df)

In [60]: store.append('food/apple', df)

In [61]: store
Out[61]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/foo/bar/bah       frame       (shape->[8,3])
/food/apple        frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange       frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/wp                wide_table  (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])

# remove all nodes under this level
In [62]: store.remove('food')

In [63]: store
Out[63]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/foo/bar/bah       frame       (shape->[8,3])
/wp                wide_table  (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])

• added mixed-dtype support!

In [64]: df['string'] = 'string'

In [65]: df['int'] = 1

In [66]: store.append('df',df)

In [67]: df1 = store.select('df')

In [68]: df1
Out[68]:
    A    B    C  string  int
2000-01-01 -2.036047 0.000830 -0.955697  string  1
2000-01-02 -0.898872 -0.725411 0.059904  string  1
2000-01-03 -0.449644 1.082900 -1.221265 string 1
2000-01-04 0.361078 1.330704 0.855932 string 1
2000-01-05 -1.216718 1.488887 0.018993 string 1
2000-01-06 -0.877046 0.045976 0.437274 string 1
2000-01-07 -0.567182 -0.888657 -0.556383 string 1
2000-01-08 0.655457 1.117949 -2.782376 string 1

[8 rows x 5 columns]

In [69]: df1.get_dtype_counts()
Out[69]:
float64 3
int64 1
object 1
dtype: int64

- performance improvements on table writing
- support for arbitrarily indexed dimensions
- SparseSeries now has a density property (GH2384)
- enable Series.str.strip/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.

1.14.6 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 [71]: p4d
Out[71]:
<class 'pandas.core.panel.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

See the full release notes or issue tracker on GitHub for a complete list.

1.15 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.

1.15.1 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 [1]: df = DataFrame(np.random.randint(0, 2, (6, 3)), columns=['A', 'B', 'C'])
  
  In [2]: df.sort(['A', 'B'], ascending=[1, 0])

  Out[2]:
           A  B  C
    0  2  1  1
    1  1  0  1
    2  1  0  1
    3  1  0  1
    4  1  0  1
    5  1  0  1

[6 rows x 3 columns]

- `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)

  In [3]: df = DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])
  
  In [4]: df.ix[2:4] = np.nan
  
  In [5]: df.rank()

  Out[5]:
           A  B  C
    0  3  1  2
    1  2  1  3
    2 NaN NaN NaN
    3 NaN NaN NaN
    4 NaN NaN NaN
    5  1  3  2
In [6]: df.rank(na_option='top')
Out[6]:
    A  B  C
0  6  5  4
1  5  4  6
2  2  2  2
3  2  2  2
4  2  2  2
5  4  6  5

In [7]: df.rank(na_option='bottom')
Out[7]:
    A  B  C
0  3  2  1
1  2  1  3
2  5  5  5
3  5  5  5
4  5  5  5
5  1  3  2

- DataFrame has new `where` and `mask` methods to select values according to a given boolean mask (GH2109, GH2151)

  DataFrame currently supports slicing via a boolean vector the same length as the DataFrame (inside the []). The returned DataFrame has the same number of columns as the original, but is sliced on its index.

In [8]: df = DataFrame(np.random.randn(5, 3), columns = ['A','B','C'])

In [9]: df
Out[9]:
    A  B  C
0  0.706220 -1.130744 -0.690308
1 -0.885387  0.246004  1.986687
2  0.212595 -1.189832 -0.344258
3  0.816335 -1.514102  1.298184
4  0.089527  0.576687 -0.737750

[5 rows x 3 columns]

In [10]: df[df['A'] > 0]
Out[10]:
    A  B  C
0  0.706220 -1.130744 -0.690308
2  0.212595 -1.189832 -0.344258
3  0.816335 -1.514102  1.298184
4  0.089527  0.576687 -0.737750

[4 rows x 3 columns]

If a DataFrame is sliced with a DataFrame based boolean condition (with the same size as the original DataFrame), then a DataFrame the same size (index and columns) as the original is returned, with
elements that do not meet the boolean condition as *NaN*. This is accomplished via the new method `DataFrame.where`. In addition, `where` takes an optional `other` argument for replacement.

```python
In [11]: df[df>0]
Out[11]:
   A    B    C
0  0.706220 NaN NaN
1    NaN  0.246004  1.986687
2  0.212595 NaN NaN
3  0.816335 NaN  1.298184
4  0.089527  0.576687 NaN

[5 rows x 3 columns]
```

```python
In [12]: df.where(df>0)
Out[12]:
   A    B    C
0  0.706220 NaN NaN
1    NaN  0.246004  1.986687
2  0.212595 NaN NaN
3  0.816335 NaN  1.298184
4  0.089527  0.576687 NaN

[5 rows x 3 columns]
```

```python
In [13]: df.where(df>0,-df)
Out[13]:
   A    B    C
0  0.706220  1.130744  0.690308
1  0.885387  3.000000  3.000000
2  3.000000  1.189832  0.344258
3  3.000000  1.514102  3.000000
4  0.089527  0.576687  0.737750

[5 rows x 3 columns]
```

Furthermore, `where` now aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via `.ix` (but on the contents rather than the axis labels).

```python
In [14]: df2 = df.copy()

In [15]: df2[ df2[1:4] > 0 ] = 3

In [16]: df2
Out[16]:
   A    B    C
0  0.706220 -1.130744 -0.690308
1 -0.885387  3.000000  3.000000
2  3.000000 -1.189832 -0.344258
3  3.000000 -1.514102  3.000000
4  0.089527  0.576687 -0.737750

[5 rows x 3 columns]
```

`DataFrame.mask` is the inverse boolean operation of `where`.

```python
In [17]: df.mask(df<=0)
Out[17]:
   A    B    C
0  0.706220 -1.130744 -0.690308
1 -0.885387  3.000000  3.000000
2  3.000000 -1.189832 -0.344258
3  3.000000 -1.514102  3.000000
4  0.089527  0.576687 -0.737750
```
A  B  C
0  0.706220  NaN  NaN
1  NaN  0.246004  1.986687
2  0.212595  NaN  NaN
3  0.816335  NaN  1.298184
4  0.089527  0.576687  NaN

[5 rows x 3 columns]

- Enable referencing of Excel columns by their column names (GH1936)

In [18]: xl = ExcelFile('data/test.xls')

In [19]: xl.parse('Sheet1', index_col=0, parse_dates=True,
   ....:         parse_cols='A:D')
   ....:
Out[19]:
A  B  C
2000-01-03  0.980269  3.685731  -0.364217
2000-01-04  1.047916  -0.041232  -0.161812
2000-01-05  0.498581  0.731168  -0.537677
2000-01-06  1.120202  1.567621  0.003641
2000-01-07  -0.487094  0.571455  -1.611639
2000-01-10  0.836649  0.246462  0.588543
2000-01-11 -0.157161  1.340307  1.195778

[7 rows x 3 columns]

- 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)

### 1.15.2 API changes

- Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

In [20]: prng = period_range('2012Q1', periods=2, freq='Q')

In [21]: s = Series(np.random.randn(len(prng)), prng)

In [22]: s.resample('M')
Out[22]:
2012-01  0.194513
2012-02  NaN
2012-03  NaN
2012-04 -0.854246
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)
  
  ```python
  In [23]: p = Period('2012')
  
  In [24]: p.end_time
  Out[24]: 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)
  
  ```python
  In [25]: data = 'A,B,C
  00001,001,5
  00002,002,6'
  
  In [26]: read_csv(StringIO(data), converters={'A': lambda x: x.strip()})
  Out[26]:
  A  B  C
  0  1  5
  1  2  6
  [2 rows x 3 columns]
  ```

See the full release notes or issue tracker on GitHub for a complete list.

### 1.16 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.

#### 1.16.1 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)

#### 1.16.2 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:
  
  ```python
  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  
dtype: int64  

In [6]: s2 = Series(s1, index=['foo', 'bar', 'baz'])  
In [7]: s2  
Out[7]:  
foo  NaN  
bar  NaN  
baz  NaN  
dtype: float64  

- Deprecated `day_of_year` API removed from `PeriodIndex`, use `dayofyear` (GH1723)  
- Don’t modify NumPy suppress printoption to True at import time  
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by `HDFStore` (GH1834, GH1824)  
- Legacy cruft removed: `pandas.stats.misc.quantileTS`  
- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)  
- Empty DataFrame columns are now created as object dtype. This will prevent a class of `TypeError`s that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)  
- Setting parts of DataFrame/Panel using `ix` now aligns input Series/DataFrame (GH1630)  
- `first` and `last` methods in `GroupBy` no longer drop non-numeric columns (GH1809)  
- Resolved inconsistencies in specifying custom NA values in text parser. `na_values` of type dict no longer override default NAs unless `keep_default_na` is set to false explicitly (GH1657)  
- `DataFrame.dot` will not do data alignment, and also work with Series (GH1915)  

See the [full release notes](https://github.com/pandas-dev/pandas) or issue tracker on GitHub for a complete list.
1.17 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.

1.17.1 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)

1.17.2 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

1.18 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.

1.18.1 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)
1.18.2 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.

1.18.3 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

1.18.4 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

1.18. v0.8.0 (June 29, 2012)
• Add **update** methods to Series/DataFrame for updating values in place
• Add **any** and **all** method to DataFrame

### 1.18.5 New plotting methods

Series.plot now supports a secondary_y option:

```python
In [1]: plt.figure()
Out[1]: <matplotlib.figure.Figure at 0xa9d450ec>

In [2]: fx['FR'].plot(style='g')
Out[2]: <matplotlib.axes._subplots.AxesSubplot at 0xa9d450cc>

In [3]: fx['IT'].plot(style='k--', secondary_y=True)
Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0xa9d45c6c>
```

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 0xac3c2e8c>

In [6]: s.hist(normed=True, alpha=0.2)
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0xa9f4066c>

In [7]: s.plot(kind='kde')
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0xa9f4066c>
```

See the plotting page for much more.

### 1.18.6 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.
1.18.7 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:

```
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', offset='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:

```
In [15]: stamp_array = rng.asobject

In [16]: stamp_array
Out[16]:
Index([2000-01-01 00:00:00, 2000-01-02 00:00:00, 2000-01-03 00:00:00,
      2000-01-04 00:00:00, 2000-01-05 00:00:00, 2000-01-06 00:00:00,
      2000-01-07 00:00:00, 2000-01-08 00:00:00, 2000-01-09 00:00:00,
      2000-01-10 00:00:00],
dtype='object')
```

```
In [17]: stamp_array[5]
Out[17]: Timestamp('2000-01-06 00:00:00', offset='D')
```

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='datetime64[ns]')
```
datetime.datetime(2000, 1, 8, 0, 0),
datetime.datetime(2000, 1, 9, 0, 0),
datetime.datetime(2000, 1, 10, 0, 0)), dtype=object)

In [20]: dt_array[5]
Out[20]: datetime.datetime(2000, 1, 6, 0, 0)

matplotlib knows how to handle `datetime.datetime` but not `Timestamp` objects. While I recommend that you plot time series using `TimeSeries.plot`, you can either use `to_pydatetime` or register a converter for the `Timestamp` type. See matplotlib documentation for more on this.

```
Warning: There are bugs in the user-facing API with the nanosecond `datetime64` unit in NumPy 1.6. In particular, the string version of the array shows garbage values, and conversion to `dtype=object` is similarly broken.
```

```
In [21]: rng = date_range('1/1/2000', periods=10)
In [22]: rng
Out[22]:
              '2000-01-09', '2000-01-10'],
             dtype='datetime64[ns]', freq='D', tz=None)
```

```
In [23]: np.asarray(rng)
Out[23]:
array(['2000-01-01T01:00:00.000000000+0100','2000-01-02T01:00:00.000000000+0100','2000-01-03T01:00:00.000000000+0100','2000-01-04T01:00:00.000000000+0100','2000-01-05T01:00:00.000000000+0100','2000-01-06T01:00:00.000000000+0100','2000-01-07T01:00:00.000000000+0100','2000-01-08T01:00:00.000000000+0100','2000-01-09T01:00:00.000000000+0100','2000-01-10T01:00:00.000000000+0100'], dtype='datetime64[ns]')
```

```
In [24]: converted = np.asarray(rng, dtype=object)
In [25]: converted[5]
Out[25]: 947116800000000000L
```

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.

### 1.19 v0.7.3 (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.

1.19.1 New features

- New fixed width file reader, read_fwf
- New scatter_matrix function for making a scatter plot matrix

```python
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

1.19.2 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
In comparisons, NA / NaN will always come through as False except with != which is True. Be very careful with boolean arithmetic, especially negation, in the presence of NA data. You may wish to add an explicit NA filter into boolean array operations if you are worried about this:

```
In [4]: mask = series == 'Steve'
```

```
In [5]: series[mask & series.notnull()]
```

While propagating NA in comparisons may seem like the right behavior to some users (and you could argue on purely technical grounds that this is the right thing to do), the evaluation was made that propagating NA everywhere, including in numerical arrays, would cause a large amount of problems for users. Thus, a “practicality beats purity” approach was taken. This issue may be revisited at some point in the future.

1.19.3 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:

```
In [1]: 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 [2]: df
```

```
Out[2]:
```

```
<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>foo</td>
<td>one</td>
<td>0.144909</td>
<td>1.387310</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>one</td>
<td>-1.033812</td>
<td>0.063490</td>
</tr>
<tr>
<td>2</td>
<td>foo</td>
<td>two</td>
<td>0.197333</td>
<td>1.437656</td>
</tr>
<tr>
<td>3</td>
<td>bar</td>
<td>three</td>
<td>-0.059730</td>
<td>-0.814844</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>two</td>
<td>0.087205</td>
<td>-0.482060</td>
</tr>
<tr>
<td>5</td>
<td>bar</td>
<td>two</td>
<td>-1.607906</td>
<td>1.521442</td>
</tr>
<tr>
<td>6</td>
<td>foo</td>
<td>one</td>
<td>-1.275249</td>
<td>0.882182</td>
</tr>
<tr>
<td>7</td>
<td>foo</td>
<td>three</td>
<td>-0.054460</td>
<td>-0.108020</td>
</tr>
</tbody>
</table>

[8 rows x 4 columns]

```
In [3]: grouped = df.groupby('A')['C']
```

```
In [4]: grouped.describe()
```

```
Out[4]:
```
A
bar  count 3.000000
  mean  -0.900483
  std    0.782652
  min   -1.607906
  25%   -1.320859
  50%   -1.033812
  75%   -0.546771
  ...  
foo  mean  -0.180052
  std    0.619410
  min   -1.275249
  25%   -0.054460
  50%    0.087205
  75%    0.144909
  max    0.197333
dtype: float64

In [5]: grouped.apply(lambda x: x.order()[-2:])  # top 2 values
Out[5]:
A
bar 1 -1.033812
  3 -0.059730
foo 0  0.144909
  2  0.197333
dtype: float64

1.20 v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

1.20.1 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)

1.20.2 Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)
1.21 v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

1.21.1 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 (GH877)
- 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

1.21.2 Performance improvements

- Improve performance and memory usage of `fillna` on DataFrame
- Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH877)

1.22 v.0.7.0 (February 9, 2012)

1.22.1 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)

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  0.119046  0.455043 -0.093701 -0.330828
std   0.814006  0.972606  0.948124  0.814913
```
min     -0.964456  -0.790943  -1.921164  -1.578003
25%     -0.512550  -0.462622  -0.683389  -0.934434
50%     0.013691   0.415879   -0.061961  -0.343709
75%     0.616168   1.351857   0.671847   0.150746
max     1.507974   1.755240   1.183075   1.051356
[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`. 
1.22.2 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:

```
In [3]: s = Series(randn(10), index=range(0, 20, 2))
In [4]: s
Out[4]:
0   -0.392051
2   -0.189537
4       0.886170
6   -1.125894
8       0.319635
10    0.998222
12    0.091743
14  -2.032047
16  -0.448560
18    0.730510
dtype: float64
```

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:
1.22.3 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:

```python
In [8]: s = Series(randn(6), index=list('gmkaec'))

In [9]: s
Out[9]:
g    1.269713
m    1.209524
k    2.160843
a    0.533532
e   -2.371548
c    0.562726
dtype: float64
```

Then this is OK:

```python
In [10]: s.ix['k':'e']
Out[10]:
k    2.160843
a    0.533532
e   -2.371548
dtype: float64
```

But this is not:

```python
In [12]: s.ix['b':'h']
KeyError 'b'
```

If the index had been sorted, the “range selection” would have been possible:

```python
In [11]: s2 = s.sort_index()

In [12]: s2
Out[12]:
a    0.533532
c    0.562726
e   -2.371548
g    1.269713
k    2.160843
m    1.209524
dtype: float64
```

```python
In [13]: s2.ix['b':'h']
Out[13]:
c    0.562726
e   -2.371548
g    1.269713
dtype: float64
```
1.22.4 Changes to Series [] operator

As as notational convenience, you can pass a sequence of labels or a label slice to a Series when getting and setting values via [] (i.e. the __getitem__ and __setitem__ methods). The behavior will be the same as passing similar input to ix except in the case of integer indexing:

```
In [14]: s = Series(randn(6), index=list('acegkm'))

In [15]: s
Out[15]:
          a    2.031757
          c    0.851077
          e    0.660056
          g   -1.662471
          k    0.571380
          m    0.945588
       dtype: float64

In [16]: s[['m', 'a', 'c', 'e']]
Out[16]:
          m    0.945588
          a    2.031757
          c    0.851077
          e    0.660056
       dtype: float64

In [17]: s['b':'l']
Out[17]:
          c    0.851077
          e    0.660056
         g   -1.662471
          k    0.571380
       dtype: float64

In [18]: s['c':'k']
Out[18]:
          c    0.851077
          e    0.660056
         g   -1.662471
          k    0.571380
       dtype: float64
```

In the case of integer indexes, the behavior will be exactly as before (shadowing ndarray):

```
In [19]: s = Series(randn(6), index=range(0, 12, 2))

In [20]: s[[4, 0, 2]]
Out[20]:
          4    -1.263534
          0    -0.414691
          2     2.108285
       dtype: float64

In [21]: s[1:5]
Out[21]:
          2     2.108285
          4    -1.263534
          6     2.617801
          8     1.967592
```
If you wish to do indexing with sequences and slicing on an integer index with label semantics, use ix.

### 1.22.5 Other API Changes

- The deprecated LongPanel class has been completely removed
- If Series.sort is called on a column of a DataFrame, an exception will now be raised. Before it was possible to accidentally mutate a DataFrame’s column by doing df[col].sort() instead of the side-effect free method df[col].order() (GH316)
- Miscellaneous renames and deprecations which will (harmlessly) raise FutureWarning
- drop added as an optional parameter to DataFrame.reset_index (GH699)

### 1.22.6 Performance improvements

- Cythonized GroupBy aggregations no longer presort the data, thus achieving a significant speedup (GH93). GroupBy aggregations with Python functions significantly sped up by clever manipulation of the ndarray data type in Cython (GH496).
- Better error message in DataFrame constructor when passed column labels don’t match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don’t print length by default in Series.to_string, add length option (GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of Series.__getitem__ for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in setup.py if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also (GH536)
- Default name assignment when calling reset_index on DataFrame with a regular (non-hierarchical) index (GH476)
- Use Cythonized groupers when possible in Series/DataFrame stat ops with level parameter passed (GH545)
- Ported skiplist data structure to C to speed up rolling_median by about 5-10x in most typical use cases (GH374)

### 1.23 v.0.6.1 (December 13, 2011)

#### 1.23.1 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

1.23.2 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)

1.24 v.0.6.0 (November 25, 2011)

1.24.1 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 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)

1.24.2 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)

1.25 v.0.5.0 (October 24, 2011)

1.25.1 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)
• Implemented `&` and `|` to intersect / union `Index` objects, respectively (GH261)
• Added `pivot_table` convenience function to pandas namespace (GH234)
• Implemented `Panel.rename_axis` function (GH243)
• `DataFrame` will show index level names in console output (GH334)
• Implemented `Panel.take`
• Added `set_eng_float_format` for alternate `DataFrame` floating point string formatting (ENH61)
• Added convenience `set_index` function for creating a `DataFrame` index from its existing columns
• Implemented `groupby` hierarchical index level name (GH223)
• Added support for different delimiters in `DataFrame.to_csv` (GH244)
• TODO: DOCS ABOUT TAKE METHODS

1.25.2 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

1.26 v.0.4.3 through v0.4.1 (September 25 - October 9, 2011)

1.26.1 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_dtype_counts and property dtypes (ENHdc)
• Added ignore_index option to DataFrame.append to stack DataFrames (ENH1b)
• read_csv tries to sniff delimiters using csv.Sniffer (GH146)
• read_csv can read multiple columns into a MultiIndex; DataFrame's to_csv method writes out a corresponding MultiIndex (GH151)
• DataFrame.rename has a new copy parameter to rename a DataFrame in place (ENHed)
• Enable unstacking by name (GH142)
• Enable sortlevel to work by level (GH141)

1.26.2 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 2.6, 2.7, 3.2, 3.3, and 3.4.

2.2 Installing pandas

2.2.1 Trying out pandas, no installation required!

The easiest way to start experimenting with pandas doesn’t involve installing pandas at all.

Wakari is a free service that provides a hosted IPython Notebook service in the cloud.

Simply create an account, and have access to pandas from within your browser via an IPython Notebook in a few minutes.

2.2.2 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).
2.2.3 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:

```bash
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:

```bash
source activate name_of_my_env
```

On Windows the command is:

```bash
activate name_of_my_env
```

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

```bash
conda install pandas
```

To install a specific pandas version:

```bash
conda install pandas=0.13.1
```

To install other packages, IPython for example:

```bash
conda install ipython
```

To install the full Anaconda distribution:

```bash
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:

```bash
conda install pip
pip install django
```

2.2.4 Installing from PyPI

Pandas can be installed via pip from PyPI.

```bash
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.
### 2.2.5 Installing using your Linux distribution’s package manager.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Status</th>
<th>Download / Repository Link</th>
<th>Install method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian</td>
<td>stable</td>
<td>official Debian repository</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Debian &amp; Ubuntu</td>
<td>unstable (latest packages)</td>
<td>NeuroDebian</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>stable</td>
<td>official Ubuntu repository</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td></td>
<td>unstable (daily builds)</td>
<td>PythonXY PPA; activate by: sudo add-apt-repository ppa:pythonxy/pythonxy-devel &amp; sudo apt-get update</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>OpenSuse &amp; Fedora</td>
<td>stable</td>
<td>OpenSuse Repository</td>
<td>zypper in python-pandas</td>
</tr>
</tbody>
</table>

### 2.2.6 Installing from source

See the [contributing documentation](#) for complete instructions on building from the git source tree. Further, see [creating a development environment](#) if you wish to create a pandas development environment.

### 2.2.7 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 nose and run:

```bash
$ nosetests pandas
..........................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................

Ran 818 tests in 21.631s

OK (SKIP=2)
```
2.3 Dependencies

- **NumPy**: 1.7.0 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 2.1 or higher.
- **bottleneck**: for accelerating certain types of **nan** evaluations. **bottleneck** uses specialized cython routines to achieve large speedups.

**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 0.19.1 or higher.
- **SciPy**: miscellaneous statistical functions
- **PyTables**: necessary for HDF5-based storage. Version 3.0.0 or higher required, Version 3.2.0 or higher highly recommended.
- **SQLAlchemy**: for SQL database support. Version 0.8.1 or higher recommended.
- **matplotlib**: for plotting
- **statsmodels**
  - Needed for parts of **pandas.stats**
- **openpyxl, xlrd/xlw**
  - openpyxl version 1.6.1 or higher, but lower than 2.0.0
  - Needed for Excel I/O
- **XlsxWriter**
  - Alternative Excel writer.
- **boto**: necessary for Amazon S3 access.
- **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.
- **Google’s python-gflags and google-api-python-client**
  - Needed for **gbq**
- **setuptools**
  - Needed for **gbq** (specifically, it utilizes **pkg_resources**)

pandas: powerful Python data analysis toolkit, Release 0.16.2

Chapter 2. Installation
• **httplib2**
  
  - Needed for gbq

- 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 reading gotchas* for reasons as to why you should probably not take this approach.

<table>
<thead>
<tr>
<th>Warning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if you install BeautifulSoup4 you must install either lxml or html5lib or both. <code>read_html()</code> will not work with only BeautifulSoup4 installed.</td>
</tr>
<tr>
<td>- You are highly encouraged to read <em>HTML reading gotchas</em>. It explains issues surrounding the installation and usage of the above three libraries</td>
</tr>
<tr>
<td>- <strong>You may need to install an older version of BeautifulSoup4:</strong></td>
</tr>
<tr>
<td>* Versions 4.2.1, 4.1.3 and 4.0.2 have been confirmed for 64 and 32-bit Ubuntu/Debian</td>
</tr>
<tr>
<td>- Additionally, if you’re using Anaconda you should definitely read <em>the gotchas about HTML parsing libraries</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if you’re on a system with apt-get you can do</td>
</tr>
</tbody>
</table>

```
    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 Enthought Canopy may be worth considering.
### 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 mailing list

### 3.2 Bug Reports/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. Since many versions of *pandas* are supported, knowing version information will also identify improvements made since previous versions. Often trying the bug-producing code out on the *master* branch is 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 have the code formatted nicely by using GitHub Flavored Markdown:

   ```python
   >>> from pandas import DataFrame
   >>> df = DataFrame(...) 
   ...
   ```

2. Include the full version string of *pandas* and its dependencies. In recent (>0.12) versions of *pandas* you can use a built in function:

   ```python
   >>> from pandas.util.print_versions import show_versions
   >>> show_versions()
   ```

   and in 0.13.1 onwards:

   ```python
   >>> 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 make the process straightforward and will work without much trouble. 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 working seamlessly with 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:

git clone git@github.com:your-user-name/pandas.git pandas-yourname
cd pandas-yourname
git remote add upstream git://github.com/pydata/pandas.git

This creates the directory pandas-yourname and connects your repository to the upstream (main project) pandas repository.

The testing suite will run automatically on Travis-CI 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 Travis-CI needs to be hooked up to your GitHub repository. Instructions are for doing so are here.

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:

git branch shiny-new-feature
git checkout shiny-new-feature

The above can be simplified to:

git checkout -b shiny-new-feature

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to pandas. You can have many shiny-new-features and switch in between them using the git checkout command.

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 lastest 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 Install Anaconda or Install 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 name you would like for this environment by running:

```bash
conda create -n pandas_dev --file ci/requirements_dev.txt
```

For a python 3 environment:

```bash
conda create -n pandas_dev python=3 --file ci/requirements_dev.txt
```

If you are on windows, then you will need to install the compiler linkages:

```bash
conda install -n pandas_dev libpython
```

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:

```bash
conda install -n pandas_dev -c pandas pytables scipy
```

To install all pandas dependencies you can do the following:

```bash
conda install -n pandas_dev -c pandas --file ci/requirements_all.txt
```

To work in this environment, activate it as follows:

```bash
activate pandas_dev
```

At which point, the prompt will change to indicate you are in the new development environment.

**Note:** The above syntax is for windows environments. To work on macosx/linux, use:

```bash
source activate pandas_dev
```

To view your environments:

```bash
conda info -e
```

To return to you home root environment:

```bash
dec activate
```

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 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:
python setup.py develop

This makes a symbolic link that tells the Python interpreter to import pandas from your development directory. Thus, you can always be using the development version on your system without being inside the clone directory.

### 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!

Actually, there are sections of the docs that are worse off by 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.

#### Documentation:

- About the pandas documentation
- How to build the pandas documentation
  - Requirements
  - Building the documentation
  - Built Master Branch Documentation

#### 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
     :cache: True

  x = 2
  x**3
  ```
  
  will be rendered as

  In [1]: x = 2

3.4. Contributing to the documentation
In [2]: x**3
Out[2]: 8

This means that almost all code examples in the docs are always run (and the output saved) during the doc build. This way, they will always be up to date, but it makes the doc building a bit more complex.

### 3.4.2 How to build the pandas documentation

#### Requirements

To build the pandas 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.

It is easiest to create a development environment, then install:

```bash
conda install -n pandas_dev sphinx ipython
```

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. Because all the code in the documentation is executed during the doc build, the examples using this optional dependencies will generate errors. Run `pd.show_versions()` to get an overview of the installed version of all dependencies.

**Warning:** Sphinx version >= 1.2.2 or the older 1.1.3 is required.

#### Building the documentation

So how do you build the docs? Navigate to your local the folder `pandas/doc/` directory in the console and run:

```bash
python make.py html
```

And then you can find the html output in the folder `pandas/doc/build/html/`.

The first time it will take quite a while, because it has to run all the code examples in the documentation and build all generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```bash
python make.py clean
python make.py build
```

Starting with 0.13.1 you can tell `make.py` to compile only a single section of the docs, greatly reducing the turnaround time for checking your changes. You will be prompted to delete `.rst` files that aren’t required. This is okay since the prior version can be checked out from git, but make sure to not commit the file deletions.

```
# 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. However, subsequent builds only process portions you changed. Now, 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!

**Built 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.

### 3.5 Contributing to the code base

**Code Base:**

- Code Standards
- Test-driven Development/Writing Code
  - Writing tests
  - Running the test suite
  - Running the performance test suite
- Documenting your code

#### 3.5.1 Code Standards

*pandas* uses the PEP8 standard. There are several tools to ensure you abide by this standard.

We’ve written a tool to check that your commits are PEP8 great, `pip install pep8radius`. Look at PEP8 fixes in your branch vs master with:

```
pep8radius master --diff
```

and make these changes with `pep8radius master --diff --in-place`

Alternatively, use `flake8` tool for checking the style of your code. Additional standards are outlined on the code style wiki page.

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 Test-driven Development/Writing Code

*Pandas* is serious about testing and strongly encourages individuals 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*. It is worth getting in the habit of writing tests ahead of time so this is never an issue.
Like many packages, pandas uses the Nose testing system and the convenient extensions in numpy.testing.

Writing tests

All tests should go into the tests subdirectory of the specific package. There are probably many examples already there and looking to these for inspiration is suggested. If you 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)
```

Running the test suite

The tests can then be run directly inside your git clone (without having to install pandas) by typing:

```
nosetests 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. This is done using one of the following constructs:

```
nosetests pandas/tests/[test-module].py
nosetests pandas/tests/[test-module].py:[TestClass]
nosetests pandas/tests/[test-module].py:[TestClass].[test_method]
```

Running the performance test suite

Performance matters and it is worth considering that your code has not introduced performance regressions. Currently pandas uses the vbench library to enable easy monitoring of the performance of critical pandas operations. These benchmarks are all found in the pandas/vb_suite directory. vbench currently only works on python2.

To install vbench:

```
pip install git+https://github.com/pydata/vbench
```

Vbench also requires sqalchemy, gitpython, and psutil which can all be installed using pip. If you need to run a benchmark, change your directory to the pandas root and run:
./test_perf.sh -b master -t HEAD

This will checkout the master revision and run the suite on both master and your commit. Running the full test suite can take up to one hour and use up to 3GB of RAM. Usually it is sufficient to past a subset of the results in to the Pull Request to show that the committed changes do not cause unexpected performance regressions.

You can run specific benchmarks using the -r flag which takes a regular expression.

See the performance testing wiki for information on how to write a benchmark.

### 3.5.3 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.

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.

### 3.6 Contributing your changes to pandas

#### 3.6.1 Committing your code

Keep style fixes to a separate commit to make your PR more readable.

Once you’ve made changes, you can see them by typing:

`git status`

If you’ve created a new file, it is not being tracked by git. Add it by typing

`git add path/to/file-to-be-added.py`

Doing ‘git status’ again should give something like

```
# On branch shiny-new-feature
#
#     modified:   /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message. Pandas uses a convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- ENH: Enhancement, new functionality
- BUG: Bug fix
- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts
- PERF: Performance improvement
- CLN: Code cleanup
The following defines how a commit message should be structured. Please reference the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < 80 chars.
- One blank line.
- Optionally, a commit message body.

Now you can commit your changes in your local repository:

```
git commit -m
```

If you have multiple commits, it is common to 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.

### 3.6.2 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/pydata/pandas.git (fetch)
upstream git://github.com/pydata/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.

### 3.6.3 Review your code

When you’re ready to ask for a code review, you will file a Pull Request. Before you do, again make sure you’ve 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 off of:

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.
3.6.4 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 appears 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:

```bash
git push -f origin shiny-new-feature
```

This will automatically update your Pull Request with the latest code and restart the Travis-CI tests.

3.6.5 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

```bash
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can just do:

```bash
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`, or else git won’t warn you if your feature branch has not actually been merged.

The branch will still exist on GitHub, so to delete it there do

```bash
git push origin --delete shiny-new-feature
```
FREQUENTLY ASKED QUESTIONS (FAQ)

4.1 DataFrame memory usage

As of pandas version 0.15.0, 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()`:

```python
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 = DataFrame(data)
In [5]: df['categorical'] = df['object'].astype('category')
In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 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: 303.5+ 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`.

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 [7]: df.memory_usage()
Out[7]:
bool      5000
complex128  80000
datetime64[ns]  40000
float64    40000
int64      40000
object     20000
timedelta64[ns]  40000
categorical  5800

dtype: int64
```

# total memory usage of dataframe
```python
In [8]: df.memory_usage().sum()
Out[8]: 270800
```

By default the memory usage of the dataframe’s index is not shown in the returned Series, the memory usage of the index can be shown by passing the `index=True` argument:

```python
In [9]: df.memory_usage(index=True)
Out[9]:
Index     40000
bool      5000
complex128  80000
datetime64[ns]  40000
float64    40000
int64      40000
object     20000
timedelta64[ns]  40000
categorical  5800

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`.

### 4.2 Migrating from scikits.timeseries to pandas >= 0.8.0

Starting with pandas 0.8.0, users of scikits.timeseries should have all of the features that they need to migrate their code to use pandas. Portions of the scikits.timeseries codebase for implementing calendar logic and timespan frequency conversions (but not resampling, that has all been implemented from scratch from the ground up) have been ported to the pandas codebase.

The scikits.timeseries notions of `Date` and `DateArray` are responsible for implementing calendar logic:

```python
In [16]: dt = ts.Date('Q', '1984Q3')

# sic
In [17]: dt
Out[17]: <Q-DEC : 1984Q1>
```
In [18]: dt.asfreq('D', 'start')
Out[18]: <D : 01-Jan-1984>

In [19]: dt.asfreq('D', 'end')
Out[19]: <D : 31-Mar-1984>

In [20]: dt + 3
Out[20]: <Q-DEC : 1984Q4>

Date and DateArray from scikits.timeseries have been reincarnated in pandas Period and PeriodIndex:

In [10]: pnow('D')  # scikits.timeseries.now()
Out[10]: Period('2015-06-13', 'D')

In [11]: Period(year=2007, month=3, day=15, freq='D')
Out[11]: Period('2007-03-15', 'D')

In [12]: p = Period('1984Q3')

In [13]: p
Out[13]: Period('1984Q3', 'Q-DEC')

In [14]: p.asfreq('D', 'start')
Out[14]: Period('1984-07-01', 'D')

In [15]: p.asfreq('D', 'end')
Out[15]: Period('1984-09-30', 'D')

In [16]: (p + 3).asfreq('T') + 6 * 60 + 30
Out[16]: Period('1985-07-01 06:29', 'T')

In [17]: rng = period_range('1990', '2010', freq='A')

In [18]: rng
Out[18]:
dtype='int64', freq='A-DEC')

In [19]: rng.asfreq('B', 'end') - 3
Out[19]:
dtype='int64', freq='B')

<table>
<thead>
<tr>
<th>scikits.timeseries</th>
<th>pandas</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Date</td>
<td>A span of time, from yearly through to secondly</td>
</tr>
<tr>
<td>DateArray</td>
<td>DateArray</td>
<td>An array of timespans</td>
</tr>
<tr>
<td>convert</td>
<td>convert</td>
<td>Frequency conversion in scikits.timeseries</td>
</tr>
<tr>
<td>convert_to_annual</td>
<td>convert_to_annual</td>
<td>currently supports up to daily frequency, see GH736</td>
</tr>
</tbody>
</table>
4.2.1 PeriodIndex / DateArray properties and functions

The scikits.timeseries DateArray had a number of information properties. Here are the pandas equivalents:

<table>
<thead>
<tr>
<th>scikits.timeseries</th>
<th>pandas</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_steps</td>
<td>np.diff(idx.values)</td>
<td></td>
</tr>
<tr>
<td>has_missing_dates</td>
<td>not idx.is_full</td>
<td></td>
</tr>
<tr>
<td>is_full</td>
<td>idx.is_full</td>
<td></td>
</tr>
<tr>
<td>is_valid</td>
<td>idx.is_monotonic and idx.is_unique</td>
<td></td>
</tr>
<tr>
<td>is_chronological</td>
<td>idx.is_monotonic</td>
<td></td>
</tr>
<tr>
<td>arr.sort_chronologically</td>
<td>idx.order()</td>
<td></td>
</tr>
</tbody>
</table>

4.2.2 Frequency conversion

Frequency conversion is implemented using the resample method on Series and DataFrame objects with a DateTimeIndex or PeriodIndex. resample also works on panels (3D). Here is some code that resamples daily data to monthly:

In [20]: rng = period_range('Jan-2000', periods=50, freq='M')
In [21]: data = Series(np.random.randn(50), index=rng)
In [22]: data
Out[22]:
2000-01  1.544821
2000-02 -1.708552
2000-03  1.545458
2000-04 -0.735738
2000-05 -0.649091
2000-06 -0.403878
2000-07 -2.474932
...  
2003-08  1.034493
2003-09  1.269838
2003-10  0.606166
2003-11 -0.827409
2003-12 -0.943863
2004-01  1.041569
2004-02  0.701815
Freq: M, dtype: float64
In [23]: data.resample('A', how=np.mean)
Out[23]:
2000    0.102447
2001   -0.204847
2002    0.210840
2003    0.300564
2004    0.871692
Freq: A-DEC, dtype: float64

4.2.3 Plotting

Much of the plotting functionality of scikits.timeseries has been ported and adopted to pandas’s data structures. For example:
4.2.4 Converting to and from period format

Use the `to_timestamp` and `to_period` instance methods.

4.2.5 Treatment of missing data

Unlike scikits.timeseries, pandas data structures are not based on NumPy’s `MaskedArray` object. Missing data is represented as `NaN` in numerical arrays and either as `None` or `NaN` in non-numerical arrays. Implementing a version of pandas’s data structures that use `MaskedArray` is possible but would require the involvement of a dedicated maintainer. Active pandas developers are not interested in this.
4.2.6 Resampling with timestamps and periods

resample has a kind argument which allows you to resample time series with a DatetimeIndex to PeriodIndex:

```python
In [27]: rng = date_range('1/1/2000', periods=200, freq='D')
```

```python
In [28]: data = Series(np.random.randn(200), index=rng)
```

```python
In [29]: data[:10]
Out[29]:
2000-01-01 -0.197661
2000-01-02  0.507155
2000-01-03 -0.493913
2000-01-04 -0.994339
2000-01-05 -0.581662
2000-01-06 -0.855251
2000-01-07 -0.256469
2000-01-08 -0.454868
2000-01-09  0.519612
2000-01-10  0.764490
Freq: D, dtype: float64
```

```python
In [30]: data.index
Out[30]:
'2000-01-09', '2000-01-10',
'2000-07-17', '2000-07-18'],
dtype='datetime64[ns]', length=200, freq='D', tz=None)
```

```python
In [31]: data.resample('M', kind='period')
Out[31]:
2000-01 -0.226155
2000-02  0.056704
2000-03 -0.132553
2000-04 -0.064003
2000-05  0.233736
2000-06 -0.301008
2000-07 -0.584631
Freq: M, dtype: float64
```

Similarly, resampling from periods to timestamps is possible with an optional interval (‘start’ or ‘end’) convention:

```python
In [32]: rng = period_range('Jan-2000', periods=50, freq='M')
```

```python
In [33]: data = Series(np.random.randn(50), index=rng)
```

```python
In [34]: resampled = data.resample('A', kind='timestamp', convention='end')
```

```python
In [35]: resampled.index
Out[35]:
DatetimeIndex(['2000-12-31', '2001-12-31', '2002-12-31', '2003-12-31',
'2004-12-31'],
dtype='datetime64[ns]', freq='A-DEC', tz=None)
```
4.3 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. 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 [36]: x = np.array(list(range(10)), '>i4')  # big endian
In [37]: newx = x.byteswap().newbyteorder()  # force native byteorder
In [38]: s = Series(newx)
```

See the NumPy documentation on byte order for more details.

4.4 Visualizing Data in Qt applications

```
import numpy as np
import pandas as pd
from pandas.sandbox.qtpandas import DataFrameModel, DataFrameWidget
from PySide import QtGui, QtCore

class MainWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        super(MainWidget, self).__init__(parent)

        # Create two DataFrames
        self.df1 = pd.DataFrame(np.arange(9).reshape(3, 3),
                                columns=['foo', 'bar', 'baz'])
        self.df2 = pd.DataFrame(
            {'int': [1, 2, 3],
             'float': [1.5, 2.5, 3.5],
             'string': ['a', 'b', 'c'],
             'nan': [np.nan, np.nan, np.nan]},
            index=['AAA', 'BBB', 'CCC'],
            columns=['int', 'float', 'string', 'nan'])

        # Create the widget and set the first DataFrame
        self.widget = DataFrameWidget(self.df1)

        # Create the buttons for changing DataFrames
        self.button_first = QtGui.QPushButton('First')
```

Warning: The qt support is deprecated and will be removed in a future version. We refer users to the external package pandas-qt.
self.button_first.clicked.connect(self.on_first_click)
self.button_second = QtGui.QPushButton('Second')
self.button_second.clicked.connect(self.on_second_click)

# Set the layout
vbox = QtGui.QVBoxLayout()
vbox.addWidget(self.widget)
hbox = QtGui.QHBoxLayout()
hbox.addWidget(self.button_first)
hbox.addWidget(self.button_second)
vbox.addLayout(hbox)
self.setLayout(vbox)

def on_first_click(self):
    '''Sets the first DataFrame'''
    self.widget.setDataFrame(self.df1)

def on_second_click(self):
    '''Sets the second DataFrame'''
    self.widget.setDataFrame(self.df2)

if __name__ == '__main__':
    import sys

    # Initialize the application
    app = QtGui.QApplication(sys.argv)
mw = MainWidget()
mw.show()
    app.exec_()
Pandas consists of the following things:

- 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.)
- Static and moving window linear and panel regression

### 5.1 Data structures at a glance

<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></td>
<td>DataFrame</td>
<td>General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns</td>
</tr>
<tr>
<td></td>
<td>Panel</td>
<td>General 3D labeled, also size-mutable array</td>
</tr>
</tbody>
</table>

#### 5.1.1 Why more than 1 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 Panel is a container for DataFrame objects. 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
```

## 5.2 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.

## 5.3 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.

Longer discussions occur on the developer mailing list, and commercial support inquiries for Lambda Foundry should be sent to: support@lambdafoundry.com

## 5.4 Credits

Pandas development began at AQR Capital Management in April 2008. It was open-sourced at the end of 2009. AQR continued to provide resources for development through the end of 2011, and continues to contribute bug reports today.

Since January 2012, Lambda Foundry, has been providing development resources, as well as commercial support, training, and consulting for pandas.

Pandas is only made possible by a group of people around the world like you who have contributed new code, bug reports, fixes, comments and ideas. A complete list can be found on Github.

## 5.5 Development Team

Pandas is a part of the PyData project. The PyData Development Team is a collection of developers focused on the improvement of Python’s data libraries. The core team that coordinates development can be found on Github. If you’re interested in contributing, please visit the project website.

## 5.6 License

```
License
=======

Pandas is distributed under a 3-clause ("Simplified" or "New") BSD license. Parts of NumPy, SciPy, numpydoc, bottleneck, which all have
```

Chapter 5. Package overview
BSD-compatible licenses, are included. Their licenses follow the pandas license.

pandas license
==============

Copyright (c) 2011-2012, Lambda Foundry, Inc. and PyData Development Team
All rights reserved.

Copyright (c) 2008-2011 AQR Capital Management, LLC
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 any 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 HOLDER 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 OWNER 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.

About the Copyright Holders
===========================

AQR Capital Management began pandas development in 2008. Development was led by Wes McKinney. AQR released the source under this license in 2009. Wes is now an employee of Lambda Foundry, and remains the pandas project lead.

The PyData Development Team is the collection of developers of the PyData project. This includes all of the PyData sub-projects, including pandas. The core team that coordinates development on GitHub can be found here: http://github.com/pydata.

Full credits for pandas contributors can be found in the documentation.

Our Copyright Policy
====================

5.6. License
PyData uses a shared copyright model. Each contributor maintains copyright over their contributions to PyData. However, it is important to note that these contributions are typically only changes to the repositories. Thus, the PyData source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire PyData Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change when they commit the change to one of the PyData repositories.

With this in mind, the following banner should be used in any source code file to indicate the copyright and license terms:

```bash
#-----------------------------------------------------------------------------
# Copyright (c) 2012, PyData Development Team
# All rights reserved.
# # Distributed under the terms of the BSD Simplified License.
# # The full license is in the LICENSE file, distributed with this software.
# #_____________________________________________________________________

Other licenses can be found in the LICENSES directory.
```
CHAPTER SIX

10 MINUTES TO PANDAS

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the Cookbook. Customarily, we import as follows:

```
In [1]: import pandas as pd
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

### 6.1 Object Creation

See the Data Structure Intro section.

Creating a `Series` by passing a list of values, letting pandas create a default integer index:

```
In [4]: s = pd.Series([1,3,5,np.nan,6,8])
```

```
Out[4]:
0    1
1    3
2    5
3  NaN
4    6
5    8
dtype: float64
```

Creating a `DataFrame` by passing a numpy array, with a datetime index and labeled columns:

```
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', tz=None)
```

```
In [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
In [9]: df
```

```
Out[9]:
          A         B         C         D
0  0.584795  0.345370 -0.332786 -0.422753
1  1.008809 -0.688866 -0.858451 -0.566378
2 -1.345714  0.859985  0.017776  0.392361
3  0.397164  0.759913  0.049669 -0.425391
4  1.027450  0.178606  0.255469 -0.325030
5 -0.263562 -0.004986  0.310039 -0.823989
```
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
Out[11]:
   A    B           C         D     E            F
0  1  2013-01-02   1.000000   3.000000  test    foo
1  1  2013-01-02   1.000000   3.000000  train   foo
2  1  2013-01-02   1.000000   3.000000  test    foo
3  1  2013-01-02   1.000000   3.000000  train   foo

Having specific dtypes

In [12]: df2.dtypes
Out[12]:
A       float64
B  datetime64[ns]
C       float32
D       int32
E    category
F        object
dtype: object

If you’re using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here’s a subset of the attributes that will be completed:

In [13]: df2.<TAB>
df2.A       df2.boxplot
df2.abs       df2.C
df2.add       df2.clip
df2.add_prefix df2.clip_lower
df2.add_suffix df2.clip_upper
df2.align     df2.columns
df2.all       df2.combine
df2.any       df2.combineAdd
df2.append    df2.combine_first
df2.apply     df2.combineMult
df2.applymap  df2.compound
df2.as_blocks df2.consolidate
df2.asfreq    df2.convert_objects
df2.as_matrix df2.copy
df2.astype    df2.corr
df2.at        df2.corrwith
df2.at_time   df2.count
df2.axes      df2.cov
df2.B  df2.cummax
df2.between_time  df2.cummin
df2.bfill  df2.cumprod
df2.blocks  df2.cumsum
df2.bool  df2.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.

6.2 Viewing Data

See the Basics section

See the top & bottom rows of the frame

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

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

Display the index, columns, and the underlying numpy data

In [16]: df.index
Out[16]:

DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
'2013-01-05', '2013-01-06'],
dtype='datetime64[ns]', freq='D', tz=None)

In [17]: df.columns
Out[17]: Index([u'A', u'B', u'C', u'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 ]])

Describe shows a quick statistic summary of your data

In [19]: df.describe()
Out[19]:

count 6.000000 6.000000 6.000000 6.000000
pandas: powerful Python data analysis toolkit, Release 0.16.2

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 -0.767252 -0.386188
75%  0.658444  0.041933 -0.034326  0.461706
max  1.212112  0.567020  0.276232  1.071804

Transposing your data

In [20]: df.T
Out[20]:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.469112</td>
<td>1.212112</td>
<td>-0.861849</td>
<td>0.721555</td>
<td>-0.424972</td>
<td>-0.673690</td>
</tr>
<tr>
<td>B</td>
<td>-0.282863</td>
<td>-0.173215</td>
<td>-2.104569</td>
<td>-0.706771</td>
<td>0.567020</td>
<td>0.113648</td>
</tr>
<tr>
<td>C</td>
<td>-1.509059</td>
<td>0.119209</td>
<td>-0.494929</td>
<td>-1.039575</td>
<td>0.276232</td>
<td>-1.478427</td>
</tr>
<tr>
<td>D</td>
<td>-1.135632</td>
<td>-1.044236</td>
<td>1.071804</td>
<td>0.271860</td>
<td>-1.087401</td>
<td>0.524988</td>
</tr>
</tbody>
</table>

Sorting by an axis

In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>-1.135632</td>
<td>-1.509059</td>
<td>-0.282863</td>
<td>0.469112</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-1.044236</td>
<td>0.119209</td>
<td>-0.173215</td>
<td>1.212112</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>1.071804</td>
<td>-0.494929</td>
<td>-2.104569</td>
<td>-0.861849</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.271860</td>
<td>-1.039575</td>
<td>-0.706771</td>
<td>0.721555</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-1.087401</td>
<td>0.276232</td>
<td>0.567020</td>
<td>-0.424972</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>0.524988</td>
<td>-1.478427</td>
<td>0.113648</td>
<td>-0.673690</td>
</tr>
</tbody>
</table>

Sorting by values

In [22]: df.sort(columns='B')
Out[22]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
</tbody>
</table>

6.3 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

6.3.1 Getting

Selecting a single column, which yields a Series, equivalent to df.A
6.3.2 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

6.3. Selection
Reduction in the dimensions of the returned object

```python
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

```python
In [30]: df.loc[dates[0],'A']
Out[30]: 0.46911229990718628
```

For getting fast access to a scalar (equiv to the prior method)

```python
In [31]: df.at[dates[0],'A']
Out[31]: 0.46911229990718628
```

### 6.3.3 Selection by Position

See more in *Selection by Position*

Select via the position of the passed integers

```python
In [32]: df.iloc[3]
Out[32]:
A   0.721555
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

```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

```python
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

```python
In [35]: df.iloc[1:3,:]
Out[35]:
   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
```
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
```

### 6.3.4 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 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  0.271860 NaN
2013-01-05  NaN  0.567020  0.276232 NaN
2013-01-06  NaN  0.113648  NaN  0.524988
```

A where operation for getting.

**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  0.271860 NaN
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
2013-01-05 -0.424972  0.567020  0.276232 -1.087401 four
```

6.3. Selection
pandas: powerful Python data analysis toolkit, Release 0.16.2

2013-01-06 -0.673690 0.113648 -1.478427 0.524988 three

In [44]: df2[df2['E'].isin(['two','four'])]
Out[44]:
A    B     C     D    E
2013-01-03 -0.861849 -2.104569 -0.494929 1.071804 two
2013-01-05 -0.424972  0.567020  0.276232 -1.087401 four

6.3.5 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]:
   A         B     C          D    F
0 0.000000  0.000000 -1.509059  5  NaN
1 1.212112 -0.173215  0.119209  5   1
2 -0.861849 -2.104569 -0.494929  5   2
3 0.721555 -0.706771 -1.039575  5   3
4 -0.424972  0.567020  0.276232  5   4
5 -0.673690  0.113648 -1.478427  5   5

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
0 0.000000  0.000000 -1.509059  5  NaN
1 1.212112 -0.173215  0.119209  5   1
2 -0.861849 -2.104569 -0.494929  5   2
3 0.721555 -0.706771 -1.039575  5   3
4 -0.424972  0.567020  0.276232  5   4
5 -0.673690  0.113648 -1.478427  5   5
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
0 0.0000000 0.0000000 -1.509059 -5.0000000 5.0000000 1.0000000
1 1.212112 -0.173215 0.119209 5.0000000 1.0000000 1.0000000
2 -0.861849 -2.104569 -0.494929 5.0000000 2.0000000 NaN
3 0.721555 -0.706771 -1.039575 5.0000000 3.0000000 NaN

To drop any rows that have missing data.

In [58]: df1.dropna(how='any')
Out[58]:
   A         B         C         D         F     E
0 1.212112 -0.173215 0.119209 5.0000000 1.0000000 1.0000000

Filling missing data

In [59]: df1.fillna(value=5)
Out[59]:
   A         B         C         D         F     E
0 0.0000000 0.0000000 -1.509059 5.0000000 5.0000000 1.0000000
1 1.212112 -0.173215 0.119209 5.0000000 1.0000000 1.0000000
2 -0.861849 -2.104569 -0.494929 5.0000000 2.0000000 5.0000000
3 0.721555 -0.706771 -1.039575 5.0000000 3.0000000 5.0000000

To get the boolean mask where values are nan

In [60]: pd.isnull(df1)
Out[60]:
   A         B         C         D         F     E
0 False False False False True False
1 False False False False False False
2 False False False False False True
3 False False False False False True

6.5 Operations

See the Basic section on Binary Ops
### 6.5.1 Stats

Operations in general exclude missing data.

Performing a descriptive statistic

```python
In [61]: df.mean()
Out[61]:
A   -0.004474
B   -0.383981
C   -0.687758
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis

```python
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.

```python
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
2013-01-04      3
2013-01-05      5
2013-01-06     NaN
Freq: D, dtype: float64
```

```python
In [65]: df.sub(s, axis='index')
Out[65]:
     A    B    C    D    F
2013-01-01  NaN  NaN  NaN  NaN  NaN
2013-01-02  NaN  NaN  NaN  NaN  NaN
2013-01-03 -1.861849 -3.104569 -1.494929 4  1
2013-01-04 -2.278445 -3.706771 -4.039575 2  0
2013-01-05 -5.424972 -4.432980 -4.723768 0 -1
2013-01-06  NaN  NaN  NaN  NaN  NaN
```

### 6.5.2 Apply

Applying functions to the data

```python
In [66]: df.apply(np.cumsum)
Out[66]:
```
6.5. Operations
6.6 Merge

6.6.1 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()`:

```python
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

```python
In [75]: pieces = [df[:3], df[3:7], df[7:]]
In [76]: pd.concat(pieces)
Out[76]:
   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
```
6.6.2 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

6.6.3 Append

Append rows to a dataframe. See the Appending

In [82]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [83]: df
Out[83]:
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758

In [84]: s = df.iloc[3]

In [85]: df.append(s, ignore_index=True)
Out[85]:
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758
8  0.141809  0.220390  0.435589  0.192451

6.6. Merge
6.7 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 [86]: 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))
```

Grouping and then applying a function `sum` to the resulting groups.

```python
In [87]: df.groupby('A').sum()
Out[87]:
```

Grouping by multiple columns forms a hierarchical index, which we then apply the function.

```python
In [89]: df.groupby(['A', 'B']).sum()
```
6.8 Reshaping

See the sections on Hierarchical Indexing and Reshaping.

6.8.1 Stack

In [90]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                      ....:    'foo', 'foo', 'qux', 'qux'],
                      ....:    ['one', 'two', 'one', 'two',
                      ....:     'one', 'two', 'one', 'two']]))

In [91]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [92]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [93]: df2 = df[:4]

In [94]: df2
Out[94]:
   A    B
first second
   bar one 0.029399 -0.542108
   two  0.282696 -0.087302
   baz one -1.575170  1.771208
   two  0.816482  1.100230

The stack() method “compresses” a level in the DataFrame’s columns.

In [95]: stacked = df2.stack()

In [96]: stacked
Out[96]:
   first second
first  second
   bar one A  0.029399
       B -0.542108
       two A  0.282696
              B -0.087302
   baz one A -1.575170
       B  1.771208
       two A  0.816482
              B  1.100230
dtype: float64

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack() is unstack(), which by default unstacks the last level:

In [97]: stacked.unstack()
Out[97]:
   A    B
first second
   bar one 0.029399 -0.542108
       two 0.282696 -0.087302
   baz one -1.575170  1.771208
       two 0.816482  1.100230

In [98]: stacked.unstack(1)
6.8.2 Pivot Tables

See the section on *Pivot Tables*.

In [100]: 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 [101]: df
Out[101]:
   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  0.314665
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  1.595974
11  three C  bar  0.648740  1.167115

We can produce pivot tables from this data very easily:

In [102]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[102]:
   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
## 6.9 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 [103]:** rng = pd.date_range('1/1/2012', periods=100, freq='S')

**In [104]:** ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

**In [105]:** ts.resample('5Min', how='sum')

```
Out[105]:
2012-01-01  25083
Freq: 5T, dtype: int32
```

**Time zone representation**

**In [106]:** rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

**In [107]:** ts = pd.Series(np.random.randn(len(rng)), rng)

**In [108]:** ts

```
Out[108]:
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 [109]:** ts_utc = ts.tz_localize('UTC')

**In [110]:** ts_utc

```
Out[110]:
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
```

**Convert to another time zone**

**In [111]:** ts_utc.tz_convert('US/Eastern')

```
Out[111]:
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 [112]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [113]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [114]: ts
Out[114]:
2012-01-31 -1.134623
2012-02-29 -1.561819
2012-03-31 -0.260838
2012-04-30  0.281957
2012-05-31  1.523962
Freq: M, dtype: float64

In [115]: ps = ts.to_period()

In [116]: ps
Out[116]:
2012-01 -1.134623
2012-02 -1.561819
2012-03 -0.260838
2012-04  0.281957
2012-05  1.523962
Freq: M, dtype: float64

In [117]: ps.to_timestamp()
Out[117]:
2012-01-01 -1.134623
2012-02-01 -1.561819
2012-03-01 -0.260838
2012-04-01  0.281957
2012-05-01  1.523962
Freq: MS, dtype: float64

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

In [118]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [119]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [120]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [121]: ts.head()
Out[121]:
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

6.10 Categoricals

Since version 0.15, pandas can include categorical data in a DataFrame. For full docs, see the categorical introduction and the API documentation.
Convert the raw grades to a categorical data type.

```python
In [123]: df["grade"] = df["raw_grade"].astype("category")
```

 Rename the categories to more meaningful names (assigning to `Series.cat.categories` is inplace!)

```python
In [125]: 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).

```python
In [126]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"])
```

 Sorting is per order in the categories, not lexical order.

```python
In [128]: df.sort("grade")
```

 Grouping by a categorical column shows also empty categories.

```python
In [129]: df.groupby("grade").size()
```

### 6.10. Categoricals

6.10. Categoricals

```
In [122]: df = pd.DataFrame({"id":[1,2,3,4,5,6], "raw_grade":["a", "b", "b", "a", "a", "e"]})
```

```
In [122]: df
Out[122]:
   id  raw_grade
0   1           a
1   2           b
2   3           b
3   4           a
4   5           e
```

```
In [123]: df["grade"] = df["raw_grade"].astype("category")
```

```
In [124]: df["grade"]
Out[124]:
   grade
0     a
1     b
2     b
3     a
4     a
5     e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

```
In [125]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

```
In [126]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"])
```

```
In [127]: df["grade"]
Out[127]:
   grade
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 [128]: df.sort("grade")
Out[128]:
   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 [129]: df.groupby("grade").size()
Out[129]:
   grade
very bad       1
bad           NaN
medium         NaN
good          2
very good      3
dtype: float64
```

6.10. Categoricals

```
```
6.11 Plotting

Plotting docs.

In [130]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [131]: ts = ts.cumsum()

In [132]: ts.plot()
Out[132]: <matplotlib.axes._subplots.AxesSubplot at 0xaf663c8c>

On DataFrame, plot() is a convenience to plot all of the columns with labels:

In [133]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=['A', 'B', 'C', 'D'])

In [134]: df = df.cumsum()

In [135]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[135]: <matplotlib.legend.Legend at 0xaf663c8c>
6.12 Getting Data In/Out

6.12.1 CSV

Writing to a csv file

In [136]: df.to_csv('foo.csv')

Reading from a csv file

In [137]: pd.read_csv('foo.csv')

Out[137]:

<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>0.266457</td>
<td>-0.399641</td>
<td>-0.219582</td>
<td>1.186860</td>
</tr>
<tr>
<td>1</td>
<td>-1.170732</td>
<td>-0.345873</td>
<td>1.653061</td>
<td>-0.282953</td>
</tr>
<tr>
<td>2</td>
<td>-1.734933</td>
<td>0.530468</td>
<td>2.060811</td>
<td>-0.515536</td>
</tr>
<tr>
<td>3</td>
<td>-1.555121</td>
<td>1.452620</td>
<td>0.239859</td>
<td>-1.156896</td>
</tr>
<tr>
<td>4</td>
<td>0.578117</td>
<td>0.511371</td>
<td>0.103552</td>
<td>-2.428202</td>
</tr>
<tr>
<td>5</td>
<td>0.478344</td>
<td>0.449933</td>
<td>-0.741620</td>
<td>-1.962409</td>
</tr>
<tr>
<td>6</td>
<td>1.235339</td>
<td>-0.091757</td>
<td>-1.543861</td>
<td>-1.084753</td>
</tr>
</tbody>
</table>
6.12.2 HDF5

Reading and writing to HDF Stores

Writing to a HDF5 Store

```python
In [138]: df.to_hdf('foo.h5','df')
```

Reading from a HDF5 Store

```python
In [139]: pd.read_hdf('foo.h5','df')
```

```
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]

6.12.3 Excel

Reading and writing to MS Excel

Writing to an excel file

```python
In [140]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

Reading from an excel file

```python
In [141]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
```

```
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]
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.

### 7.1 Internal Guides

pandas own *10 Minutes to pandas*

More complex recipes are in the *Cookbook*

### 7.2 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.
7.3 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

7.4 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

7.5 Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

7.6 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
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.

### 8.1 Idioms

These are some neat pandas idioms

if-then/if-then-else on one column, and assignment to another one or more columns:

```python
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
```

#### 8.1.1 if-then...

An if-then on one column

```python
In [2]: df.ix[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.ix[df.AAA >= 5,['BBB','CCC']] = 555; df
Out[3]:
   AAA  BBB  CCC
0  4.0  10.0  100
1  5.0  555.0  555
2  6.0  555.0  555
3  7.0  555.0  555

Add another line with different logic, to do the -else
In [4]: df.ix[df.AAA < 5,['BBB','CCC']] = 2000; df
Out[4]:
   AAA  BBB  CCC
0  4.0  2000.0  2000
1  5.0  555.0  555
2  6.0  555.0  555
3  7.0  555.0  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  4.0 -1000.0  2000
1  5.0 -1000.0 -1000
2  6.0 -1000.0  555
3  7.0 -1000.0 -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  4.0  10.0  100
1  5.0  20.0   50
2  6.0  30.0 - 30
3  7.0  40.0 - 50

In [8]: df['logic'] = np.where(df['AAA'] > 5,'high','low'); df
Out[8]:
   AAA  BBB  CCC    logic
0  4.0  10.0  100    low
1  5.0  20.0   50    low
2  6.0  30.0 - 30  high
3  7.0  40.0 - 50  high

8.1.2 Splitting

Split a frame with a boolean criterion
In [9]: df = pd.DataFrame({
...:    'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40],'CCC' : [100,50,-30,-50]}; df
...:)
Out[9]:
   AAA  BBB  CCC
0  4.0  10.0  100
1  5.0  20.0   50
2  6.0  30.0 - 30
3  7.0  40.0 - 50
### 8.1.3 Building Criteria

Select with multi-column criteria

```python
In [13]: df = pd.DataFrame(
    ....:     {'AAA': [4, 5, 6, 7], 'BBB': [10, 20, 30, 40], 'CCC': [100, 50, -30, -50]}); df
```

```plaintext
AAA  BBB  CCC
0   4    10   100
1   5    20    50
2   6    30   -30
3   7    40   -50
```

...and (without assignment returns a Series)

```python
In [14]: newseries = df.loc[(df['BBB'] < 25) & (df['CCC'] >= -40), 'AAA']; newseries
```

```plaintext
0   4
1   5
Name: AAA, dtype: int64
```

...or (without assignment returns a Series)

```python
In [15]: newseries = df.loc[(df['BBB'] > 25) | (df['CCC'] >= -40), 'AAA']; newseries;
```

...or (with assignment modifies the DataFrame.)

```python
In [16]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= 75), 'AAA'] = 0.1; df
```

```plaintext
AAA  BBB  CCC
0   0.1   10   100
1   5.0   20    50
2   0.1   30   -30
3   0.1   40   -50
```

Select rows with data closest to certain value using argsort

```python
In [17]: df = pd.DataFrame(
    ....:     {'AAA': [4, 5, 6, 7], 'BBB': [10, 20, 30, 40], 'CCC': [100, 50, -30, -50]}); df
```

```plaintext
AAA  BBB  CCC
0   4    10   100
1   5    20    50
2   6    30   -30
3   7    40   -50
```
AAABBBCCC
0  4  10  100
1  5  20  50
2  6  30  30
3  7  40  50

In [18]: aValue = 43.0
In [19]: df.ix[(df.CCC-aValue).abs().argsort()]
Out[19]:
AAABBBCCC
1  5  20  50
0  4  10  100
2  6  30  30
3  7  40  50

Dynamically reduce a list of criteria using a binary operators

In [20]: df = pd.DataFrame(
    ....:     {'AAA': [4,5,6,7], 'BBB': [10,20,30,40], 'CCC': [100,50,-30,-50]}); df
    ....:
Out[20]:
AAABBBCCC
0  4  10  100
1  5  20  50
2  6  30  30
3  7  40  50

In [21]: Crit1 = df.AAA <= 5.5
In [22]: Crit2 = df.BBB == 10.0
In [23]: Crit3 = df.CCC > -40.0

One could hard code:
In [24]: AllCrit = Crit1 & Crit2 & Crit3

...Or it can be done with a list of dynamically built criteria
In [25]: CritList = [Crit1,Crit2,Crit3]
In [26]: AllCrit = functools.reduce(lambda x,y: x & y, CritList)
In [27]: df[AllCrit]
Out[27]:
AAABBBCCC
0  4  10  100

8.2 Selection

8.2.1 DataFrames

The indexing docs.
Using both row labels and value conditionals
In [28]: df = pd.DataFrame(
   ....:     {'AAA': [4, 5, 6, 7], 'BBB': [10, 20, 30, 40], 'CCC': [100, 50, -30, -50]}); df
   ....:
Out[28]:
       AAA  BBB  CCC
0      4    10   100
1      5    20    50
2      6    30   -30
3      7    40   -50

In [29]: df[(df.AAA <= 6) & (df.index.isin([0, 2, 4]))]
Out[29]:
       AAA  BBB  CCC
0      4    10   100
2      6    30   -30

Use loc for label-oriented slicing and iloc positional slicing

In [30]: data = {'AAA': [4, 5, 6, 7], 'BBB': [10, 20, 30, 40], 'CCC': [100, 50, -30, -50]}
In [31]: df = pd.DataFrame(data=data, index=['foo', 'bar', 'boo', 'kar']); df
Out[31]:
           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)

In [32]: df.loc['bar':'kar']  #Label
Out[32]:
       AAA  BBB  CCC
bar      5    20    50
boo      6    30   -30
kar      7    40   -50

#Generic
In [33]: df.ix[0:3]  #Same as .iloc[0:3]
Out[33]:
       AAA  BBB  CCC
foo      4    10   100
bar      5    20    50
boo      6    30   -30

In [34]: df.ix['bar':'kar']  #Same as .loc['bar':'kar']
Out[34]:
       AAA  BBB  CCC
bar      5    20    50
boo      6    30   -30
kar      7    40   -50

Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.
In [35]: df2 = pd.DataFrame(data=data,index=[1,2,3,4]); #Note index starts at 1.

In [36]: df2.iloc[1:3] #Position-oriented
Out[36]:
   AAA  BBB  CCC
2  5.0  20.0  50.0
3  6.0  30.0 -30.0

In [37]: df2.loc[1:3] #Label-oriented
Out[37]:
   AAA  BBB  CCC
1  4.0  10.0 100.0
2  5.0  20.0  50.0
3  6.0  30.0 -30.0

In [38]: df2.ix[1:3] #General, will mimic loc (label-oriented)
Out[38]:
   AAA  BBB  CCC
1  4.0  10.0 100.0
2  5.0  20.0  50.0
3  6.0  30.0 -30.0

In [39]: df2.ix[0:3] #General, will mimic iloc (position-oriented), as loc[0:3] would raise a KeyError
Out[39]:
   AAA  BBB  CCC
1  4.0  10.0 100.0
2  5.0  20.0  50.0
3  6.0  30.0 -30.0

Using inverse operator (~) to take the complement of a mask

In [40]: df = pd.DataFrame(
   ....:     { 'AAA': [4,5,6,7], 'BBB': [10,20,30,40], 'CCC': [100,50,-30,-50] }); df
   ....:
Out[40]:
   AAA  BBB  CCC
0  4.0  10.0 100.0
1  5.0  20.0  50.0
2  6.0  30.0 -30.0
3  7.0  40.0 -50.0

In [41]: df[~((df.AAA <= 6) & (df.index.isin([0,2,4])))]
Out[41]:
   AAA  BBB  CCC
1  5.0  20.0  50.0
3  7.0  40.0 -50.0

8.2.2 Panels

Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions

In [42]: rng = pd.date_range('1/1/2013',periods=100,freq='D')

In [43]: data = np.random.randn(100, 4)

In [44]: cols = ['A','B','C','D']
In [45]: df1, df2, df3 = pd.DataFrame(data, rng, cols), pd.DataFrame(data, rng, cols), pd.DataFrame(data, rng, cols)
In [46]: pf = pd.Panel({'df1':df1,'df2':df2,'df3':df3});pf
Out[46]:
<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

#Assignment using Transpose  (pandas < 0.15)
In [47]: pf = pf.transpose(2,0,1)
In [48]: pf['E'] = pd.DataFrame(data, rng, cols)
In [49]: pf = pf.transpose(1,2,0);pf
Out[49]:
<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 E

#Direct assignment  (pandas > 0.15)
In [50]: pf.loc[:,:,'F'] = pd.DataFrame(data, rng, cols);pf
Out[50]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 6 (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

8.2.3 New Columns

Efficiently and dynamically creating new columns using applymap
In [51]: df = pd.DataFrame( 
    ....:   {'AAA' : [1,2,1,3], 'BBB' : [1,1,2,2], 'CCC' : [2,1,3,1]}); df
....:
Out[51]:
       AAA  BBB  CCC
0    1.0  1.0  2.0
1    2.0  1.0  1.0
2    1.0  2.0  3.0
3    3.0  2.0  1.0
In [52]: source_cols = df.columns # or some subset would work too.
In [53]: new_cols = [str(x) + "_cat" for x in source_cols]
In [54]: categories = {1 : 'Alpha', 2 : 'Beta', 3 : 'Charlie' }
In [55]: df[new_cols] = df[source_cols].applymap(categories.get);df
Out[55]:
       AAA  BBB  CCC    AAA_cat  BBB_cat  CCC_cat
0    1.0  1.0  2.0
1    2.0  1.0  1.0
2    1.0  2.0  3.0
3    3.0  2.0  1.0

8.2. Selection
Keep other columns when using min() with groupby

In [56]: df = pd.DataFrame(
     ...:     {'AAA' : [1,1,1,2,2,2,3,3], 'BBB' : [2,1,3,4,5,1,2,3]}); df

Out[56]:
     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 1 : idxmin() to get the index of the mins

In [57]: df.loc[df.groupby("AAA")["BBB"].idxmin()]
Out[57]:
     AAA  BBB
  1   1   1
  5   2   1
  6   3   2

Method 2 : sort then take first of each

In [58]: df.sort("BBB").groupby("AAA", as_index=False).first()
Out[58]:
     AAA  BBB
  0   1   1
  1   2   1
  2   3   2

Notice the same results, with the exception of the index.

### 8.3 MultiIndexing

The `multindexing` docs.

Creating a multi-index from a labeled frame

In [59]: 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[59]:
     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 [60]: df = df.set_index('row'); df
Out[60]:
   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
2   1.1    1.2   1.11   1.22
```

# With Hierarchical Columns

```
In [61]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split('_')) for c in df.columns]); df
Out[61]:
     One  Two
     X   Y  X   Y
row
0   1.1  1.2  1.11 1.22
1   1.1  1.2  1.11 1.22
2   1.1  1.2  1.11 1.22
```

# Now stack & Reset

```
In [62]: df = df.stack(0).reset_index(1); df
Out[62]:
   level_1  X   Y
row
0   One    1.10 1.20
0   Two    1.11 1.22
1   One    1.10 1.20
1   Two    1.11 1.22
2   One    1.10 1.20
2   Two    1.11 1.22
```

# And fix the labels (Notice the label 'level_1' got added automatically)

```
In [63]: df.columns = ['Sample','All_X','All_Y']; df
Out[63]:
   Sample  All_X  All_Y
row
0   One    1.10 1.20
0   Two    1.11 1.22
1   One    1.10 1.20
1   Two    1.11 1.22
2   One    1.10 1.20
2   Two    1.11 1.22
```

### 8.3.1 Arithmetic

Performing arithmetic with a multi-index that needs broadcasting

```
In [64]: cols = pd.MultiIndex.from_tuples([(x,y) for x in ['A','B','C'] for y in ['O','I']])

In [65]: df = pd.DataFrame(np.random.randn(2,6),index=['n','m'],columns=cols); df
Out[65]:
   A        B        C
     O   I   O  I   O  I
n  1.920906 -0.388231 -2.314394 0.665508 0.402562 0.399555
m -1.765956  0.850423  0.388054 0.992312 0.744086 -0.739776
```
In [66]: df = df.div(df['C'],level=1); df
Out[66]:
      A       B       C
       O   I   O   I   I
n 4.771702 -0.971660 -5.749162  1.665625  1  1
m -2.373321 -1.149568  0.521518 -1.341367  1  1

8.3.2 Slicing

Slicing a multi-index with xs

In [67]: coords = [('AA','one'),('AA','six'),('BB','one'),('BB','two'),('BB','six')]
In [68]: index = pd.MultiIndex.from_tuples(coords)
In [69]: df = pd.DataFrame([11,22,33,44,55],index,['MyData']); df
Out[69]:
     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 [70]: df.xs('BB',level=0,axis=0)  
Out[70]:
     MyData
one     33
two     44
    six     55

...and now the 2nd level of the 1st axis.

In [71]: df.xs('six',level=1,axis=0)
Out[71]:
     MyData
AA   22
    BB   55

Slicing a multi-index with xs, method #2

In [72]: index = list(itertools.product(['Ada','Quinn','Violet'],['Comp','Math','Sci']))
In [73]: headr = list(itertools.product(['Exams','Labs'],['I','II']))
In [74]: indx = pd.MultiIndex.from_tuples(index,names=['Student','Course'])
In [75]: cols = pd.MultiIndex.from_tuples(headr)  
   #Notice these are un-named
In [76]: data = [[70+x+y+(x*y)%3 for x in range(4)] for y in range(9)]
In [77]: df = pd.DataFrame(data,indx,cols); df
Out[77]:
     Exams Labs
  I   II   I   II
Student Course
In [78]: All = slice(None)

In [79]: df.loc['Violet']
Out[79]:
     Exams Labs
    I  II  I  II
Course
  Comp   76  77  78  79
  Math   77  79  81  80
  Sci    78  81  81  81

In [80]: df.loc[(All,'Math'),All]
Out[80]:
     Exams Labs
    I  II  I  II
Student Course
  Ada  Math  71  73  75  74
  Quinn Math  74  76  78  77
  Violet Math  77  79  81  80

In [81]: df.loc[(slice('Ada','Quinn'),'Math'),All]
Out[81]:
     Exams Labs
    I  II  I  II
Student Course
  Ada  Math  71  73
  Quinn Math  74  76
  Violet Math  77  79

In [82]: df.loc[(All,'Math'),('Exams')]
Out[82]:
     Exams Labs
    I  II
Student Course
  Ada  Math  71  73
  Quinn Math  74  76
  Violet Math  77  79

In [83]: df.loc[(All,'Math'),(All,'II')]
Out[83]:
     Exams Labs
    II  II
Student Course
  Ada  Math  73  74
  Quinn Math  76  77
  Violet Math  79  80

Setting portions of a multi-index with xs

8.3. Multindexing
### 8.3.3 Sorting

Sort by specific column or an ordered list of columns, with a multi-index

```python
In [84]: df.sort(('Labs', 'II'), ascending=False)
```

```
Out[84]:
   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;

### 8.3.4 Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

### 8.3.5 panelnd

The `panelnd` docs.

Construct a 5D panelnd

### 8.4 Missing Data

The `missing data` docs.

Fill forward a reversed timeseries

```python
In [85]: df = pd.DataFrame(np.random.randn(6,1), index=pd.date_range('2013-08-01', periods=6, freq='B'), columns=list('A'))
In [86]: df.ix[3,'A'] = np.nan
In [87]: df.reindex(df.index[::-1]).ffill()
```

```
Out[87]:
    A
2013-08-01 -1.054874
2013-08-02 -0.179642
2013-08-05  0.639589
2013-08-06   NaN
2013-08-07  1.906684
2013-08-08  0.104050
```

```
In [88]: df.reindex(df.index[::-1]).ffill()
```

```
Out[88]:
```
cumsum reset at NaN values

### 8.4.1 Replace

Using replace with backrefs

### 8.5 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

```python
In [89]: 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[89]:
   adult  animal size  weight
0  False    cat    S      8
1  False   dog    S     10
2  False    cat    M     11
3  False   fish    M      1
4  False   dog    M     20
5   True    cat    L     12
6   True    cat    L     12
```

#List the size of the animals with the highest weight.

```python
In [90]: df.groupby('animal').apply(lambda subf: subf['size'][subf['weight'].idxmax()])
Out[90]:
animal
    cat    L
dog    M
    fish    M
dtype: object
```

Using get_group

```python
In [91]: gb = df.groupby(['animal'])

In [92]: gb.get_group('cat')
Out[92]:
   adult  animal size  weight
0  False    cat    S      8
1  False    cat    M     11
```
Apply to different items in a group

In [93]: 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 [94]: expected_df = gb.apply(GrowUp)
In [95]: expected_df
Out[95]:
      size weight adult
animal
  cat    L  12.4375 True
  dog    L  20.0000 True
  fish   L   1.2500 True

Expanding Apply

In [96]: S = pd.Series([i / 100.0 for i in range(1,11)])
In [97]: def CumRet(x,y):
    ...:    return x * (1 + y)
    ...:
In [98]: def Red(x):
    ...:    return functools.reduce(CumRet,x,1.0)
    ...:
In [99]: pd.expanding_apply(S, Red)
Out[99]:
  0   1.010000
  1   1.030200
  2   1.061106
  3   1.103550
  4   1.158728
  5   1.228251
  6   1.314229
  7   1.419367
  8   1.547110
  9   1.701821
  dtype: float64

Replacing some values with mean of the rest of a group

In [100]: df = pd.DataFrame({'A' : [1, 1, 2, 2], 'B' : [1, -1, 1, 2]})
In [101]: gb = df.groupby('A')
In [102]: def replace(g):
    ...:    mask = g < 0
    ...:    g.loc[mask] = g[~mask].mean()
    ...:    return g
In [103]: gb.transform(replace)
Out[103]:
   B
0  1
1  1
2  1
3  2

Sort groups by aggregated data

In [104]: 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 [105]: code_groups = df.groupby('code')
In [106]: agg_n_sort_order = code_groups[['data']].transform(sum).sort('data')
In [107]: sorted_df = df.ix[agg_n_sort_order.index]
In [108]: sorted_df
Out[108]:
   code  data  flag
0   foo  0.16 False
1   foo   1.6 True
2   baz  0.33 False
3   baz  0.62 True
4   bar  -0.21 True
5   bar  -0.59 False

Create multiple aggregated columns

In [109]: rng = pd.date_range(start="2014-10-07", periods=10, freq='2min')
In [110]: ts = pd.Series(data = list(range(10)), index = rng)
In [111]: def MyCust(x):
       :   if len(x) > 2:
       :       return x[1] * 1.234
       :   return pd.NaT

In [112]: mhc = {'Mean' : np.mean, 'Max' : np.max, 'Custom' : MyCust}
In [113]: ts.resample("5min", how = mhc)
Out[113]:
                          Max   Custom   Mean
2014-10-07 00:00:00       2   1.234    1.0
2014-10-07 00:05:00       4      NaN    3.5
2014-10-07 00:10:00       7    7.404    6.0
2014-10-07 00:15:00       9      NaN    8.5
In [114]: ts
Out[114]:
2014-10-07 00:00:00       0
2014-10-07 00:02:00       1

8.5. Grouping
Create a value counts column and reassign back to the DataFrame

In [115]: df = pd.DataFrame({'Color': 'Red Red Red Blue'.split(),
                       'Value': [100, 150, 50, 50]}); df

Out[115]:
     Color  Value
0    Red    100
1    Red    150
2    Red     50
3  Blue     50

In [116]: df['Counts'] = df.groupby(['Color']).transform(len)

In [117]: df
Out[117]:
     Color  Value  Counts
0    Red    100     3
1    Red    150     3
2    Red     50     3
3  Blue     50     1

Shift groups of the values in a column based on the index

In [118]: 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[118]:
     beyer  line_race
Last Gunfighter   99         10
Last Gunfighter   102         10
Last Gunfighter   103          8
     Paynter   88          10
     Paynter  100          8

In [119]: df['beyer_shifted'] = df.groupby(level=0)['beyer'].shift(1)

In [120]: df
Out[120]:
     beyer  line_race  beyer_shifted
Last Gunfighter   99         10          NaN
Last Gunfighter   102         10           99
Last Gunfighter   103          8          102
     Paynter   103         10          NaN
     Paynter    88         10          103
Select row with maximum value from each group

In [121]: 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 [122]: mask = df.groupby(level=0).agg('idxmax')

In [123]: df_count = df.loc[mask['no']].reset_index()

In [124]: df_count
Out[124]:
    host service no
0   other   web  2
1   that   mail  1
2   this   mail  2

Grouping like Python’s itertools.groupby

In [125]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=['A'])

In [126]: df.A.groupby((df.A != df.A.shift()).cumsum()).groups
Out[126]: {1: [0L], 2: [1L], 3: [2L], 4: [3L, 4L, 5L], 5: [6L], 6: [7L, 8L]}

In [127]: df.A.groupby((df.A != df.A.shift()).cumsum()).cumsum()
Out[127]:
0     0
1     1
2     0
3     1
4     2
5     3
6     0
7     1
8     2
dtype: int64

8.5.1 Expanding Data

Alignment and to-date
Rolling Computation window based on values instead of counts
Rolling Mean by Time Interval

8.5.2 Splitting

Splitting a frame
Create a list of dataframes, split using a delineation based on logic included in rows.

In [128]: df = pd.DataFrame(data={'Case': ['A', 'A', 'A', 'B', 'A', 'A', 'B', 'A', 'A'],
                           'Data': np.random.randn(9)})

8.5. Grouping
```python
In [129]: dfs = list(zip(*df.groupby(pd.rolling_median((1*(df['Case']=='B')).cumsum(),3,True)).values))[-1]
In [130]: dfs[0]
Out[130]:
   Case  Data
0   A  0.174068
1   A  0.439461
2   A -0.741343
3   B  0.079673
In [131]: dfs[1]
Out[131]:
   Case  Data
4   A  0.922875
5   A  0.303638
6   B  0.917368
In [132]: dfs[2]
Out[132]:
   Case  Data
7   A -1.624062
8   A -0.758514
```

### 8.5.3 Pivot

The `Pivot` docs.

Partial sums and subtotals

```python
In [133]: df = pd.DataFrame(data={'Province' : ['ON','QC','BC','AL','AL','MN','ON'],
                     'City' : ['Toronto','Montreal','Vancouver','Calgary','Edmonton','Winnipeg','Windsor'],
                     'Sales' : [13,6,16,8,4,3,1]})
```

```python
In [134]: table = pd.pivot_table(df,values=['Sales'],index=['Province'],columns=['City'],aggfunc=np.sum,margins=True)
In [135]: table.stack('City')
```

```
Province City     Sales
   AL All     12
     Calgary  8
     Edmonton 4
   BC All     16
     Vancouver 16
   MN All     3
     Winnipeg 3
   ...  ...  ...
   All     Calgary  8
             Edmonton 4
             Montreal 6
             Toronto 13
     Vancouver 16
     Windsor 1
     Winnipeg 3

[20 rows x 1 columns]
```
Frequency table like plyr in R

In [136]: grades = [48,99,75,80,42,80,72,68,36,78]

In [137]: df = pd.DataFrame( {'ID': ['x%d %r for r in range(10)],
                       'Gender' : ['F', 'M', 'F', 'M', 'F', 'M', 'M', 'M', 'M', 'M'],
                       'Class': ['algebra','stats','bio','algebra','algebra','stats','stats','algebra','algebra','algebra'],
                       'Participated': ['yes','yes','yes','yes','yes','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})

In [138]: 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[138]:

<table>
<thead>
<tr>
<th>ExamYear</th>
<th>Grade</th>
<th>Employed</th>
<th>Participated</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>74</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2008</td>
<td>68</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2009</td>
<td>60</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

8.5.4 Apply

Rolling Apply to Organize - Turning embedded lists into a multi-index frame

In [139]: df = pd.DataFrame(data={'A' : [[2,4,8,16],[100,200],[10,20,30]], 'B' : [['a','b','c'],['jj','kk']])

In [140]: def SeriesFromSubList(aList):
       ...:     return pd.Series(aList)

In [141]: df_orgz = pd.concat(dict((ind,row.apply(SeriesFromSubList)) for ind,row in df.iterrows())

Rolling Apply with a DataFrame returning a Series

Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

In [142]: df = pd.DataFrame(data=np.random.randn(2000,2)/10000,
                       index=pd.date_range('2001-01-01',periods=2000),
                       columns=['A','B']); df

Out[142]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-01</td>
<td>-0.000056</td>
<td>-0.000059</td>
</tr>
<tr>
<td>2001-01-02</td>
<td>-0.000107</td>
<td>-0.000168</td>
</tr>
<tr>
<td>2001-01-03</td>
<td>0.000040</td>
<td>0.000061</td>
</tr>
<tr>
<td>2001-01-04</td>
<td>0.000039</td>
<td>0.000182</td>
</tr>
<tr>
<td>2001-01-05</td>
<td>0.000071</td>
<td>-0.000067</td>
</tr>
<tr>
<td>2001-01-06</td>
<td>0.000024</td>
<td>0.000031</td>
</tr>
<tr>
<td>2001-01-07</td>
<td>0.000012</td>
<td>-0.000021</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2006-06-17</td>
<td>0.000129</td>
<td>0.000094</td>
</tr>
<tr>
<td>2006-06-18</td>
<td>0.000059</td>
<td>0.000216</td>
</tr>
</tbody>
</table>

8.5. Grouping
In [143]: def gm(aDF,Const):
    .....:     v = (((aDF.A+aDF.B)+1).cumprod()-1)*Const
    .....:     return (aDF.index[0],v.iloc[-1])
    .....:

In [144]: S = pd.Series(dict([ gm(df.iloc[i:min(i+51,len(df)-1)],5) for i in range(len(df)-50) ])); S
Out[144]:
2001-01-01 -0.003108
2001-01-02 -0.001787
2001-01-03  0.000204
2001-01-04 -0.000166
2001-01-05 -0.002148
2001-01-06 -0.001831
2001-01-07 -0.001663
...
2006-04-28 -0.009152
2006-04-29 -0.006728
2006-04-30 -0.005840
2006-05-01 -0.003650
2006-05-02 -0.003801
2006-05-03 -0.004272
2006-05-04 -0.003839
dtype: float64

Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)

In [145]: rng = pd.date_range(start = '2014-01-01',periods = 100)
In [146]: df = pd.DataFrame({'Open' : np.random.randn(len(rng)),
    .....:                'Close' : np.random.randn(len(rng)),
    .....:                'Volume' : np.random.randint(100,2000,len(rng))}, index=rng); df
Out[146]:
        Close     Open  Volume
2014-01-01  1.550590  0.458513   1371
2014-01-02  0.818812 -0.508850   1433
2014-01-03  1.160619  0.257610    645
2014-01-04  0.081521 -1.773393    878
2014-01-05  1.083284 -0.560676   1143
2014-01-06 -0.518721  0.284174   1088
2014-01-07  0.140661  1.146889   1722
...     ...     ...     ...
2014-04-04  0.458193 -0.669474   1768
2014-04-05  0.108502 -1.616315    836
2014-04-06  1.418082 -1.294906    694
2014-04-07  0.486530  1.171647    796
2014-04-08  0.181885  0.501639    265
2014-04-09 -0.707238 -0.361868   1293
2014-04-10  1.211432  1.564429   1088

Chapter 8. Cookbook
8.6 Timeseries

Between times

Using indexer between time

Constructing a datetime range that excludes weekends and includes only certain times

Vectorized Lookup

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

8.6.1 Resampling

The Resample docs.

TimeGrouping of values grouped across time

TimeGrouping #2
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

8.7 Merge

The Concat docs. The Join docs.

Append two dataframes with overlapping index (emulate R rbind)

```
In [152]: rng = pd.date_range('2000-01-01', periods=6)
In [153]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=['A', 'B', 'C'])
In [154]: df2 = df1.copy()
ignore_index is needed in pandas < v0.13, and depending on df construction
In [155]: df = df1.append(df2,ignore_index=True); df
```

```
Out[155]:
     A      B      C
0 -0.174202 -0.477257  0.239870
1 -0.654455 -1.411456 -1.778457
2  0.351578  0.307871 -0.286865
3  0.565398 -0.185821  0.937593
4  0.446473  0.566368  0.721476
5  1.710685 -0.667054 -0.651191
6 -0.174202 -0.477257  0.239870
7 -0.654455 -1.411456 -1.778457
8  0.351578  0.307871 -0.286865
9  0.565398 -0.185821  0.937593
10 0.446473  0.566368  0.721476
11 1.710685 -0.667054 -0.651191
```

Self Join of a DataFrame

```
In [156]: 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[156]:
      Area  Bins  Data  Test_0
0      A  110  -0.399974    0
1      A  110  -1.519206    1
2      A  160   1.678487    0
3      A  160   0.005345    1
4      A  160  -0.185821    2
5      C   40   0.255077    0
6      C   40   1.093310    1
```

```
In [157]: df['Test_1'] = df['Test_0'] - 1
```
In [158]: pd.merge(df, df, left_on=['Bins', 'Area', 'Test_0'], right_on=['Bins', 'Area', 'Test_1'], suffixes=('_L', '_R'))

Out[158]:

<table>
<thead>
<tr>
<th>Area</th>
<th>Bins</th>
<th>Data_L</th>
<th>Test_0_L</th>
<th>Test_1_L</th>
<th>Data_R</th>
<th>Test_0_R</th>
<th>Test_1_R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>110</td>
<td>-0.399974</td>
<td>0</td>
<td>-1</td>
<td>-1.519206</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>160</td>
<td>1.678487</td>
<td>0</td>
<td>-1</td>
<td>0.005345</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>160</td>
<td>0.005345</td>
<td>1</td>
<td>0</td>
<td>-0.534461</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>40</td>
<td>0.255077</td>
<td>0</td>
<td>-1</td>
<td>1.093310</td>
<td>1</td>
</tr>
</tbody>
</table>

How to set the index and join

KDB like asof join

Join with a criteria based on the values

8.8 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 [159]: df = pd.DataFrame(
               ....:     {u'stratifying_var': np.random.uniform(0, 100, 20),
               ....:      u'price': np.random.normal(100, 5, 20)})

In [160]: df[u'quartiles'] = pd.qcut(
               ....:     df[u'stratifying_var'],
               ....:     4,
               ....:     labels=[u'0-25%', u'25-50%', u'50-75%', u'75-100%'])

In [161]: df.boxplot(column=u'price', by=u'quartiles')

Out[161]: <matplotlib.axes._subplots.AxesSubplot at 0xa98ad08c>
8.9 Data In/Out

Performance comparison of SQL vs HDF5

8.9.1 CSV

The CSV docs
read_csv in action
appending to a csv
how to read in multiple files, appending to create a single dataframe
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

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 combinging 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
```

8.9.2 SQL

The SQL docs
Reading from databases with SQL

8.9.3 Excel

The Excel docs
Reading from a filelike handle Reading HTML tables from a server that cannot handle the default request header
8.9.4 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

```python
In [162]: df = pd.DataFrame(np.random.randn(8,3))

In [163]: store = pd.HDFStore('test.h5')

In [164]: store.put('df', df)

# you can store an arbitrary python object via pickle
In [165]: store.get_storer('df').attrs.my_attribute = dict(A = 10)

In [166]: store.get_storer('df').attrs.my_attribute
Out[166]: {'A': 10}
```

8.9.5 Binary Files

pandas readily accepts numpy record arrays, if you need to read in a binary file consisting of an array of C structs. For example, given this C program in a file called main.c compiled with gcc main.c -std=gnu99 on a 64-bit machine,

```c
#include <stdio.h>
#include <stdint.h>

typedef struct _Data
{
  int32_t count;
```
double avg;
float scale;
} Data;

int main(int argc, const char *argv[]) {
    size_t n = 10;
    Data d[n];

    for (int i = 0; i < n; ++i) {
        d[i].count = i;
        d[i].avg = i + 1.0;
        d[i].scale = (float) i + 2.0f;
    }

    FILE *file = fopen("binary.dat", "wb");
    fwrite(&d, sizeof(Data), n, file);
    fclose(file);

    return 0;
}

the following Python code will read the binary file 'binary.dat' into a pandas DataFrame, where each element of the struct corresponds to a column in the frame:

```python
names = 'count', 'avg', 'scale'

# note that the offsets are larger than the size of the type because of # struct padding
offsets = 0, 8, 16
formats = 'i4', 'f8', 'f4'
dt = np.dtype({'names': names, 'offsets': offsets, 'formats': formats},
               align=True)
df = pd.DataFrame(np.fromfile('binary.dat', dt))
```

**Note:** The offsets of the structure elements may be different depending on the architecture of the machine on which the file was created. Using a raw binary file format like this for general data storage is not recommended, as it is not cross platform. We recommend either HDF5 or msgpack, both of which are supported by pandas’ IO facilities.

## 8.10 Computation

Numerical integration (sample-based) of a time series

### 8.11 Timedeltas

The `Timedeltas` docs.

Using timedeltas

```python
In [167]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
In [168]: s - s.max()
```
Adding and subtracting deltas and dates

```python
In [174]:
deltas = pd.Series([datetime.timedelta(days=i) for i in range(3)])
```

```python
In [175]:
df = pd.DataFrame(dict(A = s, B = deltas)); df
df['New Dates'] = df['A'] + df['B'];
df['Delta'] = df['A'] - df['New Dates']; df
```
Another example

Values can be set to NaT using np.nan, similar to datetime

```python
In [179]: y = s - s.shift(); y
Out[179]:
0   NaT
1   1 days
2   1 days
dtype: timedelta64[ns]
```

```python
In [180]: y[1] = np.nan; y
Out[180]:
0   NaT
1   NaT
2   1 days
dtype: timedelta64[ns]
```

## 8.12 Aliasing Axis Names

To globally provide aliases for axis names, one can define these 2 functions:

```python
In [181]: def set_axis_alias(cls, axis, alias):
    .....:     if axis not in cls._AXIS_NUMBERS:
    .....:         raise Exception("invalid axis [@s] for alias [@s]" % (axis, alias))
    .....:     cls._AXIS_ALIASES[alias] = axis
    .....:

In [182]: def clear_axis_alias(cls, axis, alias):
    .....:     if axis not in cls._AXIS_NUMBERS:
    .....:         raise Exception("invalid axis [@s] for alias [@s]" % (axis, alias))
    .....:     cls._AXIS_ALIASES.pop(alias,None)
    .....:

In [183]: set_axis_alias(pd.DataFrame,'columns', 'myaxis2')
```

```python
In [184]: df2 = pd.DataFrame(np.random.randn(3,2),columns=['c1','c2'],index=['i1','i2','i3'])
```

```python
In [185]: df2.sum(axis='myaxis2')
```

```python
Out[185]:
     i1    0.239786
     i2    0.259018
     i3    0.163470
dtype: float64
```

```python
In [186]: clear_axis_alias(pd.DataFrame,'columns', 'myaxis2')
```
8.13 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:

```python
In [187]: def expand_grid(data_dict):
    ....:     rows = itertools.product(*data_dict.values())
    ....:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
    ....:

In [188]: df = expand_grid(
    ....:     {'height': [60, 70],
    ....:      'weight': [100, 140, 180],
    ....:      'sex': ['Male', 'Female']})
    ....:

In [189]: df
Out[189]:
   sex  weight  height
 0  Male      100     60
 1  Male      100     70
 2  Male      140     60
 3  Male      140     70
 4  Male      180     60
 5  Male      180     70
 6 Female    100     60
 7 Female    100     70
 8 Female    140     60
 9 Female    140     70
10 Female   180     60
11 Female   180     70
```
INTRO TO DATA STRUCTURES

We’ll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import numpy and load pandas into your namespace:

```
In [1]: import numpy as np

# will use a lot in examples
In [2]: randn = np.random.randn

In [3]: from pandas import *
```

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.

When using pandas, we recommend the following import convention:

```
import pandas as pd
```

### 9.1 Series

**Warning**: In 0.13.0 Series has internaly been refactored to no longer sub-class ndarray but instead subclass NDFrame, similarly to the rest of the pandas containers. This should be a transparent change with only very limited API implications (See the *Internal Refactoring*)

`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 = 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 [4]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [5]: s
Out[5]:
a   -2.783
   b    0.426
   c   -0.650
   d    1.146
   e   -0.663
dtype: float64

In [6]: s.index
Out[6]: Index([u'a', u'b', u'c', u'd', u'e'], dtype='object')
```

```
In [7]: Series(randn(5))
Out[7]:
0     0.294
1    -0.405
2     1.167
3     0.842
4     0.540
dtype: float64
```

**Note:** Starting in v0.8.0, 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 [8]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [9]: Series(d)
Out[9]:
a    0
b    1
c    2
dtype: float64
```

```
In [10]: Series(d, index=['b', 'c', 'd', 'a'])
Out[10]:
b    1
   c    2
d  NaN
   a    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`
9.1.1 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 [12]: s[0]
Out[12]: -2.7827595933769937
```

```
In [13]: s[:3]
Out[13]:
   a  -2.783
   b   0.426
   c  -0.650
```

```
In [14]: s[s > s.median()]
Out[14]:
   b   0.426
   d   1.146
```

```
In [15]: s[[4, 3, 1]]
Out[15]:
   e  -0.663
   d   1.146
   b   0.426
```

```
In [16]: np.exp(s)
Out[16]:
   a  0.062
   b  1.532
   c  0.522
   d  3.147
   e  0.515
```

We will address array-based indexing in a separate section.

9.1.2 Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [17]: s['a']
Out[17]: -2.7827595933769937
```

```
In [18]: s['e'] = 12.
```
In [19]: s
Out[19]:
a   -2.783
b    0.426
c   -0.650
d    1.146
e    12.000
dtype: float64

In [20]: 'e' in s
Out[20]: True

In [21]: 'f' in s
Out[21]: False

If a label is not contained, an exception is raised:

```python
g>>> s['f']
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```
In [22]: s.get('f')
In [23]: s.get('f', np.nan)
Out[23]: nan
```

See also the section on attribute access.

### 9.1.3 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 be also be passed into most NumPy methods expecting an ndarray.

```
In [24]: s + s
Out[24]:
a   -5.566
b    0.853
c   -1.301
d    2.293
e    24.000
dtype: float64

In [25]: s * 2
Out[25]:
a   -5.566
b    0.853
c   -1.301
d    2.293
e    24.000
dtype: float64

In [26]: np.exp(s)
Out[26]:
a     0.062
b     1.532
c     0.522
```
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 [27]: s[1:] + s[:-1]
Out[27]:
  a   NaN
  b  0.853
  c -1.301
  d  2.293
  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.

### 9.1.4 Name attribute

Series can also have a name attribute:

In [28]: s = Series(np.random.randn(5), name='something')

In [29]: s
Out[29]:
  0  0.541
  1 -1.175
  2  0.129
  3  0.043
  4 -0.429
Name: something, dtype: float64

In [30]: s.name
Out[30]: 'something'

The Series name will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

### 9.2 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.

### 9.2.1 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.

```
In [31]: d = {'one' : Series([1., 2., 3.], index=['a', 'b', 'c']),
       ....:     'two' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
...
In [32]: df = DataFrame(d)
In [33]: df
Out[33]:
   one  two
  a   1   1
  b   2   2
  c   3   3
  d   NaN  4
In [34]: DataFrame(d, index=['d', 'b', 'a'])
Out[34]:
   one  two
  d   NaN  4
  b   2   2
  a   1   1
In [35]: DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[35]:
   two  three
  d   4   NaN
  b   2   NaN
  a   1   NaN
```

The row and column labels can be accessed respectively by accessing the index and columns attributes:

**Note:** When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [36]: df.index
Out[36]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [37]: df.columns
Out[37]: Index(['one', 'two'], dtype='object')
```
9.2.2 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 \texttt{range(n)} where \( n \) is the array length.

\begin{verbatim}
In [38]: d = {'one' : [1., 2., 3., 4.],
      ....:       'two' : [4., 3., 2., 1.]} 
      ....:

In [39]: DataFrame(d)
Out[39]:
    one  two
   0 1  4
   1 2  3
   2 3  2
   3 4  1

In [40]: DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[40]:
    one  two
   a  1  4
   b  2  3
   c  3  2
   d  4  1
\end{verbatim}

9.2.3 From structured or record array

This case is handled identically to a dict of arrays.

\begin{verbatim}
In [41]: data = np.zeros((2,),dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])

In [42]: data[:]=[1,2.,'Hello'],[2,3.,"World"]

In [43]: DataFrame(data)
Out[43]:
    A  B  C
   0 1  2 Hello
   1 2  3 World

In [44]: DataFrame(data, index=['first', 'second'])
Out[44]:
    A  B  C
   first 1  2Hello
   second 2  3World

In [45]: DataFrame(data, columns=['C', 'A', 'B'])
Out[45]:
    C  A  B
   0 Hello 1  2
   1 World 2  3
\end{verbatim}

\textbf{Note:} DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.
9.2.4 From a list of dicts

In [46]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [47]: DataFrame(data2)
Out[47]:
   a  b  c
0  1  2 NaN
1  5 10 20

In [48]: DataFrame(data2, index=['first', 'second'])
Out[48]:
     a  b  c
first 1  2 NaN
second 5 10 20

In [49]: DataFrame(data2, columns=['a', 'b'])
Out[49]:
   a  b
0  1  2
1  5 10

9.2.5 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary

In [50]: DataFrame(((a, b): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:   ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:   ('b', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:   ('b', 'b'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:   ('b', 'a'): {('A', 'D'): 9, ('A', 'B'): 10}))

Out[50]:
     a  b  c  a  b
A  B  4  1  5  8  10  
C  3  2  6  7 NaN
D NaN NaN NaN NaN  9

9.2.6 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.

9.2.7 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 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:

```python
In [51]: data
Out[51]:
array([[1, 2.0, 'Hello'], [2, 3.0, 'World']],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

```python
In [52]: DataFrame.from_records(data, index='C')
Out[52]:
     A  B
C  Hello  1  2
    World 2  3
```

**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:

```python
In [53]: DataFrame.from_items({('A', [1, 2, 3]), ('B', [4, 5, 6])})
Out[53]:
     A  B
0  1  4
1  2  5
2  3  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 [54]: DataFrame.from_items({('A', [1, 2, 3]), ('B', [4, 5, 6])},
                           orient='index', columns=['one', 'two', 'three'])
```

**9.2.8 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 [55]: df['one']
Out[55]:
a  1
b  2
c  3
d  NaN
Name: one, dtype: float64
```
In [56]: df['three'] = df['one'] * df['two']

In [57]: df['flag'] = df['one'] > 2

In [58]: df
Out[58]:
   one  two  three  flag
  a   1   1     1  False
  b   2   2     4  False
  c   3   3     9   True
  d  NaN   4   NaN  False

Columns can be deleted or popped like with a dict:

In [59]: del df['two']

In [60]: three = df.pop('three')

In [61]: df
Out[61]:
   one  flag
  a   1  False
  b   2  False
  c   3   True
  d  NaN  False

When inserting a scalar value, it will naturally be propagated to fill the column:

In [62]: df['foo'] = 'bar'

In [63]: df
Out[63]:
   one  flag  foo
  a   1  False   bar
  b   2  False   bar
  c   3   True   bar
  d  NaN  False   bar

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:

In [64]: df['one_trunc'] = df['one'][:2]

In [65]: df
Out[65]:
   one  flag  foo  one_trunc
  a   1  False   bar     1
  b   2  False   bar     2
  c   3   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 \texttt{insert} function is available to insert at a particular location in the columns:

In [66]: df.insert(1, 'bar', df['one'][:2])

In [67]: df
Out[67]:
9.2.9 Assigning New Columns in Method Chains

New in version 0.16.0.

Inspired by dplyr's mutate verb, DataFrame has an assign() method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [68]: iris = read_csv('data/iris.data')

In [69]: iris.head()
Out[69]:
            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 [70]: (iris.assign(sepal_ratio = iris['SepalWidth'] / iris['SepalLength'])
               ....:
               ....:
               .head())
```
                   SepalLength  SepalWidth  PetalLength  PetalWidth    Name  sepal_ratio
0          5.1           3.5          1.4           0.2  Iris-setosa  0.686
1          4.9           3.0          1.4           0.2  Iris-setosa  0.612
2          4.7           3.2          1.3           0.2  Iris-setosa  0.681
3          4.6           3.1          1.5           0.2  Iris-setosa  0.674
4          5.0           3.6          1.4           0.2  Iris-setosa  0.720
```

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.

```
In [71]: iris.assign(sepal_ratio = lambda x: (x['SepalWidth'] / x['SepalLength'])).head()
```

```
                   SepalLength  SepalWidth  PetalLength  PetalWidth    Name  sepal_ratio
0          5.1           3.5          1.4           0.2  Iris-setosa  0.686
1          4.9           3.0          1.4           0.2  Iris-setosa  0.612
2          4.7           3.2          1.3           0.2  Iris-setosa  0.681
3          4.6           3.1          1.5           0.2  Iris-setosa  0.674
4          5.0           3.6          1.4           0.2  Iris-setosa  0.720
```

assign always returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don’t have a reference to the DataFrame at hand. This is common when using assign in 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:

```
In [72]: (iris.query('SepalLength > 5')
              ....:
              ....:
              .assign(SepalRatio = lambda x: x.SepalWidth / x.SepalLength,
       PetalRatio = lambda x: x.PetalWidth / x.PetalLength))
```

9.2.DataFrame 297
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:

```
In [73]: # 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'])))
```

### 9.2.10 Indexing / Selection

The basics of indexing are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select column</td>
<td>df[col]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by label</td>
<td>df.loc[label]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by integer location</td>
<td>df.iloc[loc]</td>
<td>Series</td>
</tr>
<tr>
<td>Slice rows</td>
<td>df[5:10]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Select rows by boolean vector</td>
<td>df[bool_vec]</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [74]: df.loc['b']
Out[74]:
one    2
bar    2
flag   False
foo    bar
one_trunc    2
Name: b, dtype: object

In [75]: df.iloc[2]
Out[75]:
one    3
bar    3
flag   True
foo    bar
one_trunc  NaN
Name: c, dtype: object
```

For a more exhaustive treatment of more sophisticated label-based indexing and slicing, see the section on indexing.

We will address the fundamentals of reindexing / conforming to new sets of labels in the section on reindexing.

### 9.2.11 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.

```
In [76]: df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])
```
In [77]: df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])

In [78]: df + df2
Out[78]:
   A   B   C   D
0 -1.916 -0.986 -2.421e+00 NaN
1  0.965  1.677  3.298e-01 NaN
2 -1.662  2.197 -1.917e+00 NaN
3 -0.189  0.765 -9.522e-04 NaN
4 -1.076  0.397 -1.177e+00 NaN
5  2.810 -0.179 -5.705e-01 NaN
6 -1.227  0.196  5.312e-01 NaN
7  NaN  NaN  NaN  NaN
8  NaN  NaN  NaN  NaN
9  NaN  NaN  NaN  NaN

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 [79]: df - df.iloc[0]
Out[79]:
   A   B   C   D
0  0.000  0.000  0.000  0.000
1  2.386  1.358  1.223 -2.107
2  2.105  1.700  1.327 -0.689
3  1.874  2.718  2.382 -0.760
4  2.199  0.966  0.826  0.093
5  4.997  1.197  1.330 -0.285
6  1.263  0.578  1.071 -0.525
7  3.463  0.632  1.063 -0.443
8  2.680  3.163  1.298 -1.818
9  1.304  0.196  3.590 -0.867

In the special case of working with time series data, and the DataFrame index also contains dates, the broadcasting will be column-wise:

In [80]: index = date_range('1/1/2000', periods=8)
In [81]: df = DataFrame(randn(8, 3), index=index, columns=list('ABC'))
In [82]: df
Out[82]:
   A   B   C
2000-01-01  0.063 -0.028  0.444
2000-01-02 -0.269 -1.578  1.850
2000-01-03  0.638 -0.557 -0.071
2000-01-04 -0.511  0.156 -1.076
2000-01-05  1.664 -0.438 -0.077
2000-01-06  0.029  0.179  1.740
2000-01-07 -0.729 -0.898 -0.314
2000-01-08 -0.048 -0.876  0.169

In [83]: type(df['A'])
Out[83]: pandas.core.series.Series

In [84]: df - df['A']
Out[84]:
   A   B   C
2000-01-01  0  -0.091  0.381
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 [85]:** `df * 5 + 2`

```
Out[85]:
   A  B   C
2000-01-01  2.314  1.858  4.218
2000-01-02   0.656 -5.888 11.251
2000-01-03   5.190 -0.783  1.644
2000-01-04  -0.557  2.781 -3.378
2000-01-05  10.318 -0.189  1.613
2000-01-06   2.146  2.895  10.700
2000-01-07 -1.645 -2.490   0.429
2000-01-08   1.760 -2.378  2.846
```

**In [86]:** `1 / df`

```
Out[86]:
   A   B   C
2000-01-01 15.948 -35.193  2.255
2000-01-02  -3.721  -0.634  0.540
2000-01-03   1.567  -1.797 -14.039
2000-01-04  -1.955   6.398   0.930
2000-01-05   0.601  -2.285 -12.936
2000-01-06  34.257   5.586   0.575
2000-01-07  -1.372  -1.114  -3.183
2000-01-08  -20.802  -1.142   5.913
```

**In [87]:** `df ** 4`

```
Out[87]:
   A         B         C
2000-01-01 1.546e-05  6.519e-07  3.871e-02
2000-01-02  5.219e-03  6.195e+00  1.172e+01
2000-01-03  1.657e-01  9.598e-02  2.574e-05
2000-01-04  6.841e-02  5.966e-04  1.339e+00
2000-01-05  7.660e+00  3.671e-02  3.571e-05
2000-01-06  7.261e-07  1.027e-03  9.168e+00
2000-01-07  2.825e-01  6.503e-01  9.747e-03
2000-01-08  5.341e-06  5.878e-01  8.178e-04
```

Boolean operators work as well:
In [88]: df1 = DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)

In [89]: df2 = DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)

In [90]: df1 & df2
Out[90]:
a  b
0  True False
1  True True
2  True False

In [91]: df1 | df2
Out[91]:
a  b
0  True True
1  True True
2  True True

In [92]: df1 ^ df2
Out[92]:
a  b
0  True True
1  True False
2  False True

In [93]: -df1
Out[93]:
a  b
0  False True
1  True False
2  False False

9.2.12 Transposing

To transpose, access the T attribute (also the transpose function), similar to an ndarray:

# only show the first 5 rows
In [94]: df[:5].T
Out[94]:
A         0.063  -0.269   0.638   -0.511    1.664
B         -0.028  -1.578  -0.557   0.156   -0.438
C          0.444   1.850  -0.071  -1.076   -0.077

9.2.13 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 [95]: np.exp(df)
Out[95]:
         A         B         C
2000-01-01  1.065  0.972  1.558
2000-01-02  0.764  0.206  6.361
2000-01-03  1.893  0.573  0.931
2000-01-04  0.600  1.169  0.341
In [96]: np.asarray(df)
Out[96]:
array([[ 0.0627, -0.0284, 0.4436],
       [-0.2688, -1.5776, 1.8502],
       [ 0.6381, -0.5566, -0.0712],
       [-0.5114, 0.1563, -1.0756],
       [ 1.6636, -0.4377, -0.0773],
       [ 0.0292, 0.179 , 1.7401],
       [-0.729 , -0.898 , -0.3142],
       [-0.0481, -0.8756, 0.1691]])

The dot method on DataFrame implements matrix multiplication:

In [97]: df.T.dot(df)
Out[97]:
   A  B  C
A  4.047 -0.039 0.178
B -0.039 4.621 -2.581
C  0.178 -2.581 7.943

Similarly, the dot method on Series implements dot product:

In [98]: s1 = Series(np.arange(5,10))
In [99]: s1.dot(s1)
Out[99]: 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.

9.2.14 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 [100]: baseball = read_csv('data/baseball.csv')
In [101]: print(baseball)
   id  player  year  stint ...  hbp  sh  sf  gidp
   0 88641 womacto01 2006  2 ...  0  3  0  0
   1 88643 schilcu01 2006  1 ...  0  0  0  0
   .. ..      ..      .. ...  ..  ..  ..  ..
   98 89533 aloumo01 2007  1 ...  2  0  3 13
   99 89534 alomasa02 2007  1 ...  0  0  0  0

[100 rows x 23 columns]

In [102]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 0 to 99
Data columns (total 23 columns):
id 100 non-null int64
player 100 non-null object
However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won’t always fit the console width:

```python
In [103]: print(baseball.iloc[-20:, :12].to_string())
```

New since 0.10.0, wide DataFrames will now be printed across multiple rows by default:

```python
In [104]: DataFrame(randn(3, 12))
```

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>player</td>
<td>year</td>
<td>stint</td>
<td>team</td>
<td>lg</td>
<td>g</td>
<td>ab</td>
<td>r</td>
<td>h</td>
<td>X2b</td>
<td>X3b</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>--------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
<td>----</td>
<td>---</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>finlest01</td>
<td>2007</td>
<td>1</td>
<td>COL NL</td>
<td>43</td>
<td>94</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>embreal01</td>
<td>2007</td>
<td>1</td>
<td>OAK AL</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>edmonji01</td>
<td>2007</td>
<td>1</td>
<td>SLN NL</td>
<td>117</td>
<td>365</td>
<td>39</td>
<td>92</td>
<td>15</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>easleda01</td>
<td>2007</td>
<td>1</td>
<td>NYN NL</td>
<td>76</td>
<td>193</td>
<td>24</td>
<td>54</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>delgaca01</td>
<td>2007</td>
<td>1</td>
<td>NYN NL</td>
<td>139</td>
<td>538</td>
<td>71</td>
<td>139</td>
<td>30</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>cormirh01</td>
<td>2007</td>
<td>1</td>
<td>CIN NL</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>coninje01</td>
<td>2007</td>
<td>2</td>
<td>NYN NL</td>
<td>21</td>
<td>41</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>coninje01</td>
<td>2007</td>
<td>1</td>
<td>CIN NL</td>
<td>80</td>
<td>215</td>
<td>23</td>
<td>57</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>clemero02</td>
<td>2007</td>
<td>1</td>
<td>NYA AL</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>claytro01</td>
<td>2007</td>
<td>2</td>
<td>BOS AL</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>claytro01</td>
<td>2007</td>
<td>1</td>
<td>TOR AL</td>
<td>69</td>
<td>189</td>
<td>23</td>
<td>48</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>cirilje01</td>
<td>2007</td>
<td>2</td>
<td>ARI NL</td>
<td>28</td>
<td>40</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>cirilje01</td>
<td>2007</td>
<td>1</td>
<td>MIN AL</td>
<td>50</td>
<td>153</td>
<td>18</td>
<td>40</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>bondsba01</td>
<td>2007</td>
<td>1</td>
<td>SFN NL</td>
<td>126</td>
<td>340</td>
<td>75</td>
<td>94</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>biggicr01</td>
<td>2007</td>
<td>1</td>
<td>HOU NL</td>
<td>141</td>
<td>517</td>
<td>68</td>
<td>130</td>
<td>31</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>benitar01</td>
<td>2007</td>
<td>2</td>
<td>FLO NL</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>benitar01</td>
<td>2007</td>
<td>1</td>
<td>SFN NL</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>ausmubr01</td>
<td>2007</td>
<td>1</td>
<td>HOU NL</td>
<td>117</td>
<td>349</td>
<td>38</td>
<td>82</td>
<td>16</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>aloumo01</td>
<td>2007</td>
<td>1</td>
<td>NYN NL</td>
<td>87</td>
<td>328</td>
<td>51</td>
<td>112</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>alomasa02</td>
<td>2007</td>
<td>1</td>
<td>NYN NL</td>
<td>8</td>
<td>22</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New since 0.10.0, wide DataFrames will now be printed across multiple rows by default:
You can change how much to print on a single row by setting the `display.width` option:

```python
In [105]: set_option('display.width', 40)  # default is 80

In [106]: DataFrame(randn(3, 12))
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

### 9.2.15 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:

```python
In [107]: df = DataFrame({'foo1': np.random.randn(5),
      ....:             'foo2': np.random.randn(5)})
      ....:

In [108]: df
Out[108]:
   foo1  foo2
0  0.909160  1.360298
1 -0.667763 -1.603624
2 -0.101656 -1.648929
3  1.189682  0.145121
4 -0.090648 -2.536359

In [109]: df.foo1
Out[109]:
0  0.909160
1 -0.667763
2 -0.101656
3  1.189682
4 -0.090648
Name: foo1, dtype: float64
The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.fo<TAB>
df.fool   df.foo2
```

### 9.3 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:

#### 9.3.1 From 3D ndarray with optional axis labels

```
In [110]: 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 [111]: wp
Out[111]:
<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
```

#### 9.3.2 From dict of DataFrame objects

```
In [112]: data = {'Item1' : DataFrame(randn(4, 3)),
.....:          'Item2' : DataFrame(randn(4, 2))}

In [113]: Panel(data)
Out[113]:
<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:
### Panel

<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:

```
In [114]: Panel.from_dict(data, orient='minor')
Out[114]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: 0 to 2
Major_axis axis: 0 to 3
Minor_axis axis: Item1 to Item2
```

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 `dtype=object` unless you pass `orient='minor'`:

```
In [115]: df = DataFrame({'a': ['foo', 'bar', 'baz'],
                  'b': np.random.randn(3)})

In [116]: df
Out[116]:
     a     b
0   foo -1.264356
1   bar -0.497629
2  baz  1.789719

In [117]: data = {'item1': df, 'item2': df}

In [118]: panel = Panel.from_dict(data, orient='minor')

In [119]: panel['a']
Out[119]:
item1  item2
0    foo    foo
1    bar    bar
2    baz    baz

In [120]: panel['b']
Out[120]:
item1  item2
0 -1.264356 -1.264356
1 -0.497629 -0.497629
2  1.789719  1.789719

In [121]: panel['b'].dtypes
Out[121]:
item1  float64
item2  float64
dtype: object
```

**Note:** Unfortunately 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. This will get worked on, of course, in future releases. And faster if you join me in working on the codebase.
9.3.3 From DataFrame using `to_panel` method

This method was introduced in v0.7 to replace `LongPanel.to_long`, and converts a DataFrame with a two-level index to a Panel.

```python
In [122]: midx = MultiIndex(levels=[['one', 'two'], ['x','y']], labels=[[1,1,0,0],[1,0,1,0]])

In [123]: df = DataFrame({'A' : [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [124]: df.to_panel()
Out[124]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: A to B
Major_axis axis: one to two
Minor_axis axis: x to y
```

9.3.4 Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

```python
In [125]: wp['Item1']
Out[125]:
   A    B    C    D
2000-01-01 0.835993 -0.621868 -0.173710 -0.174326
2000-01-02 -0.354356  2.090183 -0.736019 -1.250412
2000-01-03 -0.581326  0.917119  0.611695
2000-01-04 -1.576078 -0.528562 -0.704643 -0.481453
2000-01-05  1.085093 -1.229749  2.295679 -1.016910

In [126]: 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.

9.3.5 Transposing

A Panel can be rearranged using its `transpose` method (which does not make a copy by default unless the data are heterogeneous):

```python
In [127]: wp.transpose(2, 0, 1)
Out[127]:
<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
```

9.3.6 Indexing / Selection

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select item</td>
<td>wp[item]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at major_axis label</td>
<td>wp.major_xs(val)</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at minor_axis label</td>
<td>wp.minor_xs(val)</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

Chapter 9. Intro to Data Structures
For example, using the earlier example data, we could do:

```python
In [128]: wp['Item1']
Out[128]:
   A        B        C        D
2000-01-01 0.835993 -0.621868 -0.173710 -0.174326
2000-01-02 -0.354356  2.090183 -0.736019 -1.250412
2000-01-03 -0.581326 -0.244477  0.917119  0.611695
2000-01-04 -1.576078 -0.528562 -0.704643 -0.481453
2000-01-05  1.085093 -1.229749  2.295679 -1.016910
```

```python
In [129]: wp.major_xs(wp.major_axis[2])
Out[129]:
      Item1    Item2    Item3
A   -0.581326 -1.271582  0.457167
B   -0.244477 -0.861256  0.283861
C    0.917119 -0.597879 -1.533955
D    0.611695 -0.118700 -5.153265
```

```python
In [130]: wp.minor_axis
Out[130]: Index([u'A', u'B', u'C', u'D'], dtype='object')
```

```python
In [131]: wp.minor_xs('C')
Out[131]:
      Item1    Item2    Item3
2000-01-01 -0.173710  2.381645 -0.072937
2000-01-02 -0.736019 -2.413161  0.305002
2000-01-03  0.917119 -0.597879 -1.533955
2000-01-04 -0.704643 -1.536019  0.458746
2000-01-05  2.295679  0.181524 12.646732
```

9.3.7 Squeezing

Another way to change the dimensionality of an object is to \texttt{squeeze} a 1-len object, similar to \texttt{wp['Item1']}

```python
In [132]: wp.reindex(items=['Item1']).squeeze()
Out[132]:
   A        B        C        D
2000-01-01 0.835993 -0.621868 -0.173710 -0.174326
2000-01-02 -0.354356  2.090183 -0.736019 -1.250412
2000-01-03 -0.581326 -0.244477  0.917119  0.611695
2000-01-04 -1.576078 -0.528562 -0.704643 -0.481453
2000-01-05  1.085093 -1.229749  2.295679 -1.016910
```

```python
In [133]: wp.reindex(items=['Item1'],minor=['B']).squeeze()
Out[133]:
   B
2000-01-01 -0.621868
2000-01-02  2.090183
2000-01-03 -0.244477
2000-01-04 -0.528562
2000-01-05 -1.229749
Freq: D, Name: B, dtype: float64
```
9.3.8 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:

```python
In [134]: panel = Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
      ....:    major_axis=date_range('1/1/2000', periods=5),
      ....:    minor_axis=['a', 'b', 'c', 'd'])
      ....:
```

```python
In [135]: panel.to_frame()
Out[135]:
                                   one       two       three
2000-01-01 a       0.445900  -1.286198  -1.023189
                         -0.574496  -0.407154   0.591682
                         0.872979   0.068084  -0.008919
                         0.297255   2.157051  -0.415572
2000-01-02 a       -1.022617  -0.443982  -0.772683
                         1.091870  -0.881639  -0.516197
                         1.831444   0.851834   0.626655
                         1.271808  -1.352515   0.269623
2000-01-03 a       -0.472876   0.228761   1.709250
                         -0.279340   0.416858  -0.830728
                         0.495966   0.301709  -0.290244
                         0.367858   0.569010  -1.588782
2000-01-04 a       -1.530917   0.047619   0.639406
                         -0.285890   0.413370   1.055533
                         0.943062   0.573056  -0.260898
                         1.361752  -0.154419  -0.289725
2000-01-05 a       0.210373   0.987044   0.279621
                         -1.945608   0.063191   0.454423
                         2.532409   0.439086  -0.065750
                         0.373819   1.657475   1.465709
```

9.4 Panel4D (Experimental)

Panel4D is a 4-Dimensional named container very much like a Panel, but having 4 named dimensions. It is intended as a test bed for more N-Dimensional named containers.

- **labels**: axis 0, each item corresponds to a Panel contained inside
- **items**: axis 1, each item corresponds to a DataFrame contained inside
- **major_axis**: axis 2, it is the index (rows) of each of the DataFrames
- **minor_axis**: axis 3, it is the columns of each of the DataFrames

Panel4D is a sub-class of Panel, so most methods that work on Panels are applicable to Panel4D. The following methods are disabled:

- `join`, `to_frame`, `to_excel`, `to_sparse`, `groupby`

Construction of Panel4D works in a very similar manner to a Panel.
### 9.4.1 From 4D ndarray with optional axis labels

```python
In [136]: 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'])
```

```python
In [137]: p4d
Out[137]:
<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
```

### 9.4.2 From dict of Panel objects

```python
In [138]: data = { 'Label1': Panel({ 'Item1': DataFrame(randn(4, 3)) }),
        'Label2': Panel({ 'Item2': DataFrame(randn(4, 2)) }) }
```

```python
In [139]: Panel4D(data)
Out[139]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 4 (major_axis) x 3 (minor_axis)
Labels axis: Label1 to Label2
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 Panels. Thus, they can be any of the other valid inputs to Panel as per above.

### 9.4.3 Slicing

Slicing works in a similar manner to a Panel. [] slices the first dimension. .ix allows you to slice arbitrarily and get back lower dimensional objects

```python
In [140]: p4d['Label1']
Out[140]:
<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
```

4D -> Panel

```python
In [141]: p4d.ix[:,:,:,'A']
Out[141]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 5 (minor_axis)
Items axis: Label1 to Label2
```
Major_axis axis: Item1 to Item2
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00

4D -> DataFrame

In [142]: p4d.ix[:,:,0,'A']
Out[142]:
Label1   Label2
Item1  1.127489  0.015494
Item2  -1.650400  0.130533

4D -> Series

In [143]: p4d.ix[:,0,0,'A']
Out[143]:
Label1   1.127489
Label2   0.015494
Name: A, dtype: float64

9.4.4 Transposing

A Panel4D can be rearranged using its `transpose` method (which does not make a copy by default unless the data are heterogeneous):

In [144]: p4d.transpose(3, 2, 1, 0)
Out[144]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 4 (labels) x 5 (items) x 2 (major_axis) x 2 (minor_axis)
Labels axis: A to D
Items axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Major_axis axis: Item1 to Item2
Minor_axis axis: Label1 to Label2

9.5 PanelND (Experimental)

PanelND is a module with a set of factory functions to enable a user to construct N-dimensional named containers like Panel4D, with a custom set of axis labels. Thus a domain-specific container can easily be created.

The following creates a Panel5D. A new panel type object must be sliceable into a lower dimensional object. Here we slice to a Panel4D.

In [145]: from pandas.core import panelnd

In [146]: Panel5D = panelnd.create_nd_panel_factory(
    ...:     klass_name = 'Panel5D',
    ...:     orders = [ 'cool', 'labels', 'items', 'major_axis', 'minor_axis' ],
    ...:     slices = { 'labels' : 'labels', 'items' : 'items',
    ...:                'major_axis' : 'major_axis', 'minor_axis' : 'minor_axis' },
    ...:     slicer = Panel4D,
    ...:     aliases = { 'major' : 'major_axis', 'minor' : 'minor_axis' },
    ...:     stat_axis = 2)
    ...

In [147]: p5d = Panel5D(dict(C1 = p4d))
In [148]: p5d
Out[148]:
<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 1 (cool) x 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Cool axis: C1 to C1
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

# print a slice of our 5D
In [149]: p5d.ix['C1',:,:,0:3,:
Out[149]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 3 (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 2000-01-03 00:00:00
Minor_axis axis: A to D

# transpose it
In [150]: p5d.transpose(1,2,3,4,0)
Out[150]:
<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 2 (cool) x 2 (labels) x 5 (items) x 4 (major_axis) x 1 (minor_axis)
Cool axis: Label1 to Label2
Labels axis: Item1 to Item2
Items axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Major_axis axis: A to D
Minor_axis axis: C1 to C1

# look at the shape & dim
In [151]: p5d.shape
Out[151]: (1, 2, 2, 5, 4)
In [152]: p5d.ndim
Out[152]: 5
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:

```
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'])
...:
```

10.1 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.

```
In [5]: long_series = pd.Series(np.random.randn(1000))
In [6]: long_series.head()
Out[6]:
0   -0.305384
1   -0.479195
2    0.095031
3   -0.270099
4   -0.707140
dtype: float64
In [7]: long_series.tail(3)
Out[7]:
997    0.588446
998    0.026465
999   -1.728222
dtype: float64
```
10.2 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
2000-01-01  0.187483 -1.933946  0.377312
2000-01-02  0.734122  2.141616 -0.011225

In [9]: df.columns = [x.lower() for x in df.columns]
```

```python
In [10]: df
Out [10]:
          a        b        c
2000-01-01  0.187483 -1.933946  0.377312
2000-01-02  0.734122  2.141616 -0.011225
2000-01-03  0.048869 -1.360687 -0.479010
2000-01-04 -0.859661 -0.231595 -0.527750
2000-01-05 -1.296337  0.150680  0.123836
2000-01-06  0.571764  1.555563 -0.823761
2000-01-07  0.535420 -1.032853  1.469725
2000-01-08  1.304124  1.449735  0.203109
```

To get the actual data inside a data structure, one need only access the *values* property:

```python
In [11]: s.values
Out [11]: array([ 0.1122,  0.8717, -0.8161, -0.7849,  1.0307])
```

```python
In [12]: df.values
Out [12]:
array([[ 0.1875, -1.9339,  0.3773],
       [ 0.7341,  2.1416, -0.0112],
       [ 0.0489, -1.3607, -0.4790],
       [-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]])
```

```python
In [13]: wp.values
Out [13]:
array([[-1.032 ,  0.9698, -0.9627,  1.3821],
       [-0.9388,  0.6691, -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]])
```
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.

### 10.3 Accelerated operations

Pandas has support for accelerating certain types of binary numerical and boolean operations using the numexpr library (starting in 0.11.0) 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.

### 10.4 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.

#### 10.4.1 Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, ... for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the `index` or `columns` via the `axis` keyword:

```python
In [14]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
                        'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
                        'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd']))
```

10.3. Accelerated operations
In [15]: df
Out[15]:
    one  three  two
a  -0.626544  NaN  -0.351587
b  -0.138894 -0.177289  1.136249
c   0.011617  0.462215 -0.448789
d     NaN     1.124472 -1.101558

In [16]: row = df.ix[1]

In [17]: column = df['two']

In [18]: df.sub(row, axis='columns')
Out[18]:
    one  three  two
a -0.487650  NaN -1.487837
b    0.000000  0.000000  0.000000
c  0.150512  0.639504 -1.585038
d     NaN     1.301762 -2.237808

In [19]: df.sub(row, axis=1)
Out[19]:
    one  three  two
a -0.487650  NaN -1.487837
b    0.000000  0.000000  0.000000
c  0.150512  0.639504 -1.585038
d     NaN     1.301762 -2.237808

In [20]: df.sub(column, axis='index')
Out[20]:
    one  three  two
a -0.274957  NaN   0.000000
b -1.275144 -1.313539  0.000000
c  0.460406  0.911003  0.000000
d     NaN     2.226031  0.000000

In [21]: df.sub(column, axis=0)
Out[21]:
    one  three  two
a -0.274957  NaN   0.000000
b -1.275144 -1.313539  0.000000
c  0.460406  0.911003  0.000000
d     NaN     2.226031  0.000000

Furthermore you can align a level of a multi-indexed DataFrame with a Series.

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]:
first  second
    one  three  two
 1     a -0.274957  NaN   0.000000
       b -1.275144 -1.313539  0.000000
       c  0.460406  0.911003  0.000000
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 -0.546569 -0.260774
B  0.492478  0.147993
C -0.649010 -0.532794
D  0.176307  0.623812
```

```python
In [27]: wp.sub(major_mean, axis='major')
Out[27]:
<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
```

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...

### 10.4.2 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).

```python
In [28]: df
Out[28]:
    one     three     two
a -0.626544     NaN -0.351587
b -0.138894 -0.177289  1.136249
c  0.011617  0.462215 -0.448789
d     NaN   1.124472 -1.101558
```

```python
In [29]: df2
Out[29]:
    one     three     two
a -0.626544  1.000000 -0.351587
b -0.138894 -0.177289  1.136249
c  0.011617  0.462215 -0.448789
d     NaN   1.124472 -1.101558
```

```python
In [30]: df + df2
Out[30]:
    one     three     two
a -1.253088     NaN -0.703174
```
**10.4.3 Flexible Comparisons**

Starting in v0.8, pandas introduced binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` to Series and DataFrame whose behavior is analogous to the binary arithmetic operations described above:

```
In [32]: df.gt(df2)
Out[32]:
    one   three   two
   a    False    False    False
   b    False    False    False
   c    False    False    False
   d    False    False    False
```

```
In [33]: df2.ne(df)
Out[33]:
    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.

**10.4.4 Boolean Reductions**

You can apply the reductions: `empty`, `any()`, `all()`, and `bool()` to provide a way to summarize a boolean result.

```
In [34]: (df>0).all()
Out[34]:
   one   False
   three  False
   two   False
dtype: bool
```

```
In [35]: (df>0).any()
Out[35]:
   one    True
   three    True
   two    True
dtype: bool
```

You can reduce to a final boolean value.
You can test if a pandas object is empty, via the `empty` property.

```python
In [37]: df.empty
Out[37]: False
```

```python
In [38]: pd.DataFrame(columns=list('ABC')).empty
Out[38]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```python
In [39]: pd.Series([True]).bool()
Out[39]: True
```

```python
In [40]: pd.Series([False]).bool()
Out[40]: False
```

```python
In [41]: pd.DataFrame([[True]]).bool()
Out[41]: True
```

```python
In [42]: pd.DataFrame([[False]]).bool()
Out[42]: 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.

### 10.4.5 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:

```python
In [43]: df+df == df*2
Out[43]:
        one  three  two
a   True  False  True
b   True   True  True
c   True   True  True
d  False   True  True
```

```python
In [44]: (df+df == df*2).all()
Out[44]:
        one
one   False
```
three False
two True
dtype: bool

Notice that the boolean DataFrame \( df + df \) \( \Rightarrow \) \( df + 2 \) contains some False values! That is because NaNs do not compare as equals:

```
In [45]: np.nan == np.nan
Out[45]: False
```

So, as of v0.13.1, NDFrames (such as Series, DataFrames, and Panels) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [46]: (df+df).equals(df*2)
Out[46]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [47]: df1 = pd.DataFrame({'col': ['foo', 0, np.nan]})
In [48]: df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2,1,0])
In [49]: df1.equals(df2)
Out[49]: False
In [50]: df1.equals(df2.sort())
Out[50]: True
```

### 10.4.6 Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of “higher quality”. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [51]: df1 = pd.DataFrame({'A': [1., np.nan, 3., 5., np.nan],
                      'B': [np.nan, 2., 3., np.nan, 6.])

In [52]: df2 = pd.DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
                      'B': [np.nan, np.nan, 3., 4., 6., 8.]})

In [53]: df1
Out[53]:
   A   B
0  1  NaN
1 NaN   2
2  3  3
3  5  NaN
4 NaN   6

In [54]: df2
Out[54]:
   A   B
0   5  NaN
1  2  NaN
2  4   3
3  6   4
4  7   6
```

322 Chapter 10. Essential Basic Functionality
0  5  NaN
1  2  NaN
2  4  3
3  NaN  4
4  3  6
5  7  8

In [55]: df1.combine_first(df2)
Out[55]:
     A  B
0  1  NaN
1  2  2
2  3  3
3  5  4
4  3  6
5  7  8

10.4.7 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:

In [56]: combiner = lambda x, y: np.where(pd.isnull(x), y, x)

In [57]: df1.combine(df2, combiner)
Out[57]:
     A  B
0  1  NaN
1  2  2
2  3  3
3  5  4
4  3  6
5  7  8

10.5 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 [58]: df
Out[58]:
     one  three  two
a -0.626544 NaN -0.351587
b -0.138894 -0.177289 1.136249
c 0.011617 0.462215 -0.448789
d NaN 1.124472 -1.101558

**In [59]:** df.mean(0)
**Out[59]:**
   one    -0.251274
  three     0.469799
   two    -0.191421
dtype: float64

**In [60]:** df.mean(1)
**Out[60]:**
   a    -0.489066
   b     0.273355
   c     0.008348
   d     0.011457
dtype: float64

All such methods have a *skipna* option signaling whether to exclude missing data (**True** by default):

**In [61]:** df.sum(0, skipna=False)
**Out[61]:**
   one    NaN
  three    NaN
   two   -0.765684
dtype: float64

**In [62]:** df.sum(axis=1, skipna=True)
**Out[62]:**
   a    -0.978131
   b    0.820066
   c    0.025044
   d    0.022914
dtype: float64

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 [63]:** ts_stand = (df - df.mean()) / df.std()

**In [64]:** ts_stand.std()
**Out[64]:**
   one    1
  three    1
   two    1
dtype: float64

**In [65]:** xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

**In [66]:** xs_stand.std(1)
**Out[66]:**
   a    1
   b    1
   c    1
   d    1
dtype: float64
Note that methods like `cumsum()` and `cumprod()` preserve the location of NA values:

```python
In [67]: df.cumsum()
Out[67]:
          one    three    two
a -0.626544   NaN -0.351587
b -0.765438 -0.177289  0.784662
c -0.753821  0.284925  0.335874
d   NaN       1.409398 -0.765684
```

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-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>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>Unbiased standard deviation</td>
</tr>
<tr>
<td>var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>sem</td>
<td>Unbiased standard error of the mean</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>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 [68]: np.mean(df['one'])
Out[68]: -0.25127365175839511
```

```python
In [69]: np.mean(df['one'].values)
Out[69]: nan
```

Series also has a method `nunique()` which will return the number of unique non-null values:

```python
In [70]: series = pd.Series(np.random.randn(500))
```

```python
In [71]: series[20:500] = np.nan
```

```python
In [72]: series[10:20] = 5
```

```python
In [73]: series.nunique()
Out[73]: 11
```
10.5.1 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 [74]: series = pd.Series(np.random.randn(1000))
In [75]: series[::-2] = np.nan
In [76]: series.describe()
Out[76]:
    count 500.000000
    mean -0.039663
    std  1.069371
    min  -3.463789
    25%  -0.731101
    50%  -0.058918
    75%   0.672758
    max   3.120271
dtype: float64
```

```python
In [77]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [78]: frame.ix[::-2] = np.nan
In [79]: frame.describe()
Out[79]:
            a            b            c            d            e
    count  500.000000  500.000000  500.000000  500.000000  500.000000
    mean   0.000954  -0.044014   0.075936  -0.003679   0.020751
    std    1.005133   0.974882   0.967432   1.004732   0.963812
    min  -3.010899  -2.782760  -3.401252  -2.944925  -3.794127
    25%   -0.682900  -0.681161  -0.528190  -0.663503  -0.615717
    50%   -0.001651  -0.006279   0.040098  -0.003378   0.006282
    75%    0.656439   0.632852   0.717919   0.687214   0.653423
    max    3.007143   2.627688   2.702490   2.850852   3.072117
```

You can select specific percentiles to include in the output:

```python
In [80]: series.describe(percentiles=[.05, .25, .75, .95])
Out[80]:
    count 500.000000
    mean -0.039663
    std  1.069371
    min  -3.463789
    5%   -1.741334
    25%  -0.731101
    75%   0.672758
    95%   1.854383
    max   3.120271
dtype: float64
```

By default, the median is always included.

For a non-numerical Series object, `describe()` will give a simple summary of the number of unique values and most frequently occurring values:
In [81]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [82]: s.describe()
Out[82]:
   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:

In [83]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [84]: frame.describe()
Out[84]:
   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:

In [85]: frame.describe(include=['object'])
Out[85]:
   a
  count 4
  unique 2
  top    No
  freq   2

In [86]: frame.describe(include=['number'])
Out[86]:
   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

In [87]: frame.describe(include='all')
Out[87]:
   a    b
  count 4  4.000000
  unique 2    NaN
  top    No    NaN
  freq   2    NaN
  mean   NaN  1.500000
  std    NaN  1.290994
That feature relies on `select_dtypes`. Refer to there for details about accepted inputs.

### 10.5.2 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:

```python
In [88]: s1 = pd.Series(np.random.randn(5))

In [89]: s1
Out[89]:
0   -0.872725
1    1.522411
2    0.080594
3   -1.676067
4    0.435804
dtype: float64

In [90]: s1.idxmin(), s1.idxmax()
Out[90]: (3, 1)

In [91]: df1 = pd.DataFrame(np.random.randn(5,3), columns=['A','B','C'])

In [92]: df1
Out[92]:
   A       B       C
0  0.445734 -1.649461  0.169660
1  1.246181  0.131682 -2.001988
2 -1.273023  0.870502  0.214583
3  0.088452 -0.173364  1.207466
4  0.546121  0.409515 -0.310515

In [93]: df1.idxmin(axis=0)
Out[93]:
   A  B  C
0  2  0  1
dtype: int64

In [94]: df1.idxmax(axis=1)
Out[94]:
   0  1  2  3  4
A  A  A  C  A
B  B  B  B  B
C  C  C  C  C
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin()` and `idxmax()` return the first matching index:
In [95]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [96]: df3
Out[96]:
   A
  e  2
  d  1
  c  1
  b  3
  a  NaN

In [97]: df3['A'].idxmin()
Out[97]: 'd

Note: idxmin and idxmax are called argmin and argmax in NumPy.

10.5.3 Value counts (histogramming) / Mode

The value_counts() Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

In [98]: data = np.random.randint(0, 7, size=50)

In [99]: data
Out[99]:
array([5, 3, 2, 2, 1, 4, 0, 4, 0, 2, 0, 6, 4, 1, 6, 3, 3, 0, 2, 1, 0, 5, 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])

In [100]: s = pd.Series(data)

In [101]: s.value_counts()
Out[101]:
0   11
3    7
6    6
5    6
4    5
2    5
1    5
dtype: int64

In [102]: pd.value_counts(data)
Out[102]:
0   11
3    7
6    7
5    6
4    5
2    5
1    5
dtype: int64

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:
In [103]: s5 = pd.Series([1, 1, 3, 3, 5, 5, 7, 7, 7])

In [104]: s5.mode()
Out[104]:
0  3
1  7
dtype: int64

In [105]: df5 = pd.DataFrame({"A": np.random.randint(0, 7, size=50),
                        "B": np.random.randint(-10, 15, size=50)})

In [106]: df5.mode()
Out[106]:
   A  B
0   1  -5

10.5.4 Discretization and quantiling

Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

In [107]: arr = np.random.randn(20)

In [108]: factor = pd.cut(arr, 4)

In [109]: factor
Out[109]:
\n\n\n\n\n
In [110]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [111]: factor
Out[111]:
\n\n\n\n\n
qcut() computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

In [112]: arr = np.random.randn(30)

In [113]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [114]: factor
Out[114]:
\n\n\n\n\n
In [115]: pd.value_counts(factor)
Out[115]:
0.0736  1.976  8
[-0.439, -0.139]  8
We can also pass infinite values to define the bins:

```python
In [116]: arr = np.random.randn(20)
In [117]: factor = pd.cut(arr, [-np.inf, 0, np.inf])
In [118]: factor
Out[118]:
[(-inf, 0], (0, inf], (0, inf], (0, inf], (-inf, 0], ..., (-inf, 0], (0, inf], (-inf, 0], (-inf, 0],
Length: 20
Categories (2, object): [(-inf, 0] < {0, inf}]
```

## 10.6 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. **Elementwise function application**: `applymap()`

### 10.6.1 Tablewise Function Application

New in version 0.16.2.

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)`.

```python
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 [119]: import statsmodels.formula.api as sm

In [120]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [121]: (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[121]:
<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:                  Sat, 13 Jun 2015     Pseudo R-squ.:       -143.91
Time:                  14:04:05     Log-Likelihood:         -143.91
converged:             True        LL-Null:                 -460.91
                        LLR p-value:     6.774e-136
===============================================================================
               coef       std err          z       P>|z|       [95.0% Conf. Int.]
-------------------------------------------------------------------------------
Intercept        -1267.3636   457.8670      -2.768   0.006       -2164.767   -369.960
C(lg)[T.NL]      -0.2057      0.1010      -2.044   0.041        -0.403    -0.008
ln_h              0.9280      0.1910       4.866   0.000         0.554     1.302
year              0.6301      0.2282       2.762   0.006         0.183     1.077
g                 0.0099      0.0043       2.754   0.006         0.003     0.017
===============================================================================
```

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).

10.6.2 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:

In [122]: df.apply(np.mean)
Out[122]:
    one   -0.251274
    three  0.469799
    two   -0.191421
dtype: float64

In [123]: df.apply(np.mean, axis=1)
Out[123]:
    a   -0.489066
    b    0.273355
c 0.008348
d 0.011457
dtype: float64

In [124]: df.apply(lambda x: x.max() - x.min())
Out[124]:
one 0.638161
three 1.301762
two 2.237808
dtype: float64

In [125]: df.apply(np.cumsum)
Out[125]:
index  one  three  two
  a -0.626544 NaN -0.351587
  b -0.765438 -0.177289 0.784662
  c -0.753821 0.284925 0.335874
d  NaN 1.409398 -0.765684

In [126]: df.apply(np.exp)
Out[126]:
index  one  three  two
  a 0.534436 NaN 0.703570
  b 0.870320 0.837537 3.115063
  c 1.011685 1.587586 0.638401
d  NaN 3.078592 0.332353

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:

In [127]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
index=pd.date_range('1/1/2000', periods=1000))

In [128]: tsdf.apply(lambda x: x.idxmax())
Out[128]:
A 2001-04-27
B 2002-06-02
C 2000-04-02
dtype: datetime64[ns]

You may also pass additional arguments and keyword arguments to the apply() method. For instance, consider the following function you would like to apply:

```python
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

df.apply(subtract_and_divide, args=(5,), divide=3)

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

In [129]: tsdf apply(lambda x: x.idxmax())
Out[129]:
A 2001-04-27
B 2002-06-02
C 2000-04-02
dtype: datetime64[ns]
Finally, `apply()` takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

See also:
The section on GroupBy demonstrates related, flexible functionality for grouping by some criterion, applying, and combining the results into a Series, DataFrame, etc.

## 10.6.3 Applying elementwise Python functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods `applymap()` on DataFrame and analogously `map()` on Series accept any Python function taking a single value and returning a single value. For example:

```
In [131]: df4
Out[131]:
     one     three    two
a -0.626544   NaN -0.351587
b -0.138894  0.177289  1.136249
c  0.011617  0.462215  0.448789
d   NaN       1.124472 -1.101558

In [132]: f = lambda x: len(str(x))

In [133]: df4['one'].map(f)
Out[133]:
    a    14
    b    15
    c    15
    d     3
Name: one, dtype: int64

In [134]: 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 [135]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'],
   index=['a', 'b', 'c', 'd', 'e'])
   ......:
   ......:

In [136]: t = pd.Series({'six' : 6., 'seven' : 7.})

In [137]: s.map(t)
Out[137]:
   a  6
   b  7
   c  6
   d  7
   e  6

10.6.4 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.

Note: Prior to 0.13.1 apply on a Panel would only work on ufuncs (e.g. np.sum/np.max).

In [139]: import pandas.util.testing as tm
In [140]: panel = tm.makePanel(5)
In [141]: panel
Out[141]:
<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 [142]: panel['ItemA']
Out[142]:

10.6. Function application 335
A transformational apply.

```
In [143]: result = panel.apply(lambda x: x*2, axis='items')
```

```
In [144]: result
Out[144]:
<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 [145]: result['ItemA']
Out[145]:
```

```
A reduction operation.

```
In [146]: panel.apply(lambda x: x.dtype, axis='items')
Out[146]:
```

```
A similar reduction type operation

```
In [147]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[147]:
```

```
This last reduction is equivalent to

```
In [148]: panel.sum('major_axis')
Out[148]:
```

```
A transformation operation that returns a Panel, but is computing the z-score across the major_axis.
```
In [149]: result = panel.apply(
       .....: lambda x: (x-x.mean())/x.std(),
       .....: axis='major_axis')
       .....:
In [150]: result
Out[150]:
<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 [151]: result['ItemA']
Out[151]:
   A    B    C    D
2000-01-03 -0.469761 1.156225 -0.441347 1.341731
2000-01-04  1.422763 -0.444015 -0.882647  0.398661
2000-01-05 -0.156654  1.453694  0.367936 -0.619210
2000-01-06 -1.238841  0.173423  1.581149  0.156654
2000-01-07  0.442494  0.568061 -0.625091 -1.277837

Apply can also accept multiple axes in the axis argument. This will pass a DataFrame of the cross-section to the applied function.

In [152]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T

In [153]: result = panel.apply(f, axis = ['items','major_axis'])

In [154]: result
Out[154]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC

In [155]: result.loc[::,'ItemA']
Out[155]:
   A    B    C    D
2000-01-03  0.864236  1.132969  0.557316  0.575106
2000-01-04  0.795745  0.652527  0.534808 -0.070674
2000-01-05 -0.310864  0.558627  1.086688 -1.051477
2000-01-06 -0.001065  0.832460  0.846006  0.043602
2000-01-07  1.128946  1.152469 -0.218186 -0.891680

This is equivalent to the following

In [156]: result = pd.Panel(dict([ (ax, f(panel.loc[::,ax]))
       .....:     for ax in panel.minor_axis ]))
       .....:

In [157]: result
Out[157]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC
In [158]: result.loc[:, :, 'ItemA']
Out[158]:
<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>0.864236</td>
<td>1.132969</td>
<td>0.557316</td>
<td>0.575106</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.795745</td>
<td>0.652527</td>
<td>0.534808</td>
<td>-0.070674</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.310864</td>
<td>0.558627</td>
<td>1.086688</td>
<td>-1.051477</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.001065</td>
<td>0.832460</td>
<td>0.846006</td>
<td>0.043602</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.128946</td>
<td>1.152469</td>
<td>-0.218186</td>
<td>-0.891680</td>
</tr>
</tbody>
</table>

### 10.7 Reindexing and altering labels

`reindex()` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To reindex means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, fill data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```python
In [159]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```python
In [160]: s
Out[160]:
a    -1.010924
b    -0.672504
c    -1.139222
d     0.354653
e     0.563622
dtype: float64
```

```python
In [161]: s.reindex(['e', 'b', 'f', 'd'])
Out[161]:
e    0.563622
b   -0.672504
f    NaN
d     0.354653
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```python
In [162]: df
Out[162]:
<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>three</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-0.626544</td>
<td>NaN</td>
<td>-0.351587</td>
</tr>
<tr>
<td>b</td>
<td>-0.138894</td>
<td>-0.177289</td>
<td>1.136249</td>
</tr>
<tr>
<td>c</td>
<td>0.011617</td>
<td>0.462215</td>
<td>-0.448789</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
<td>1.124472</td>
<td>-1.101558</td>
</tr>
</tbody>
</table>
```

```python
In [163]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[163]:
<table>
<thead>
<tr>
<th></th>
<th>three</th>
<th>two</th>
<th>one</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
For convenience, you may utilize the `reindex_axis()` method, which takes the labels and a keyword `axis` parameter.

Note that the `Index` objects containing the actual axis labels can be shared between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [164]: rs = s.reindex(df.index)
```

```
In [165]: rs
Out[165]:
a  -1.010924
b  -0.672504
c   0.354653
dtype: float64
```

```
In [166]: rs.index is df.index
Out[166]: True
```

This means that the reindexed Series’s index is the same Python object as the DataFrame’s index.

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.

### 10.7.1 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:

```
In [167]: df2
Out[167]:
   one    two
a -0.626544 -0.351587
b -0.138894  1.136249
c  0.011617 -0.448789
```

```
In [168]: df3
Out[168]:
   one    two
a -0.375270 -0.463545
b  0.112379  1.024292
c  0.262891 -0.560746
```

```
In [169]: df.reindex_like(df2)
Out[169]:
   one    two
a -0.626544 -0.351587
b -0.138894  1.136249
c  0.011617 -0.448789
```
10.7.2 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:

```
In [170]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [171]: s1 = s[:4]
In [172]: s2 = s[1:]
In [173]: s1.align(s2)
Out[173]:
(a  -0.365106
 b   1.092702
 c  -1.481449
 d   1.781190
 e     NaN
dtype: float64, a   NaN
 b   1.092702
 c  -1.481449
 d   1.781190
 e  -0.031543
dtype: float64)
```

```
In [174]: s1.align(s2, join='inner')
Out[174]:
(b   1.092702
 c  -1.481449
 d   1.781190
dtype: float64, b  1.092702
 c -1.481449
 d   1.781190
dtype: float64)
```

```
In [175]: s1.align(s2, join='left')
Out[175]:
(a  -0.365106
 b   1.092702
 c  -1.481449
 d   1.781190
dtype: float64, a   NaN
 b   1.092702
 c -1.481449
```
For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [176]: df.align(df2, join='inner')
Out[176]:
   one  two
a  -0.626544 -0.351587
b  -0.138894  1.136249
c  0.011617  0.448789
```

You can also pass an `axis` option to only align on the specified axis:

```
In [177]: df.align(df2, join='inner', axis=0)
Out[177]:
   one  three  two
a  -0.626544  NaN  -0.351587
b  -0.138894 -0.177289  1.136249
c  0.011617  0.462215 -0.448789
```

If you pass a Series to `DataFrame.align()`, you can choose to align both objects either on the DataFrame’s index or columns using the `axis` argument:

```
In [178]: df.align(df2.ix[0], axis=1)
Out[178]:
   one  three  two
a  -0.626544  NaN  -0.351587
b  -0.138894 -0.177289  1.136249
c  0.011617  0.462215 -0.448789
d  NaN  1.124472 -1.101558
   Name: a, dtype: float64)
```

### 10.7.3 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:

```
In [179]: rng = pd.date_range('1/3/2000', periods=8)
In [180]: ts = pd.Series(np.random.randn(8), index=rng)
In [181]: ts2 = ts[[0, 3, 6]]
```
In [182]: ts
Out[182]:
2000-01-03  0.480993
2000-01-04  0.604244
2000-01-05 -0.487265
2000-01-06  1.990533
2000-01-07  0.327007
2000-01-08  1.053639
2000-01-09 -2.927808
2000-01-10  0.082065
Freq: D, dtype: float64

In [183]: ts2
Out[183]:
2000-01-03  0.480993
2000-01-06  1.990533
2000-01-09 -2.927808
dtype: float64

In [184]: ts2.reindex(ts.index)
Out[184]:
2000-01-03  0.480993
2000-01-04  NaN
2000-01-05  NaN
2000-01-06  1.990533
2000-01-07  NaN
2000-01-08  NaN
2000-01-09 -2.927808
2000-01-10  NaN
Freq: D, dtype: float64

In [185]: ts2.reindex(ts.index, method='ffill')
Out[185]:
2000-01-03  0.480993
2000-01-04  0.480993
2000-01-05  0.480993
2000-01-06  1.990533
2000-01-07  1.990533
2000-01-08  1.990533
2000-01-09 -2.927808
2000-01-10 -2.927808
Freq: D, dtype: float64

In [186]: ts2.reindex(ts.index, method='bfill')
Out[186]:
2000-01-03  0.480993
2000-01-04  1.990533
2000-01-05  1.990533
2000-01-06  1.990533
2000-01-07 -2.927808
2000-01-08 -2.927808
2000-01-09 -2.927808
2000-01-10  NaN
Freq: D, dtype: float64

In [187]: ts2.reindex(ts.index, method='nearest')
Out[187]:
2000-01-03  0.480993

342 Chapter 10. Essential Basic Functionality
2000-01-04  0.480993
2000-01-05  1.990533
2000-01-06  1.990533
2000-01-07  1.990533
2000-01-08 -2.927808
2000-01-09 -2.927808
2000-01-10 -2.927808
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 [188]: ts2.reindex(ts.index).fillna(method='ffill')
Out[188]:
          2000-01-03  0.480993
          2000-01-04  0.480993
          2000-01-05  0.480993
          2000-01-06  1.990533
          2000-01-07  1.990533
          2000-01-08  1.990533
          2000-01-09 -2.927808
          2000-01-10 -2.927808
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.

### 10.7.4 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 [189]: df
Out[189]:
     one   three   two
          a -0.626544 NaN -0.351587
          b -0.138894 -0.177289  1.136249
          c  0.011617  0.462215 -0.448789
          d  NaN  1.124472 -1.101558

In [190]: df.drop(['a', 'd'], axis=0)
Out[190]:
     one   three   two
          b -0.138894 -0.177289  1.136249
          c  0.011617  0.462215 -0.448789

In [191]: df.drop(['one'], axis=1)
Out[191]:
     three   two
          a  NaN -0.351587
          b -0.177289  1.136249
          c  0.462215 -0.448789
          d  1.124472 -1.101558
```

Note that the following also works, but is a bit less obvious / clean:

```
In [192]: df.reindex(df.index.difference(['a', 'd']))
Out[192]:
```

10.7. Reindexing and altering labels
10.7.5 Renaming / mapping labels

The \texttt{rename()} method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

\begin{verbatim}
In [193]: s
Out[193]:
a   -0.365106
b   1.092702
c  -1.481449
d   1.781190
e   -0.031543
dtype: float64

In [194]: s.rename(str.upper)
Out[194]:
A   -0.365106
B    1.092702
C   -1.481449
D    1.781190
E   -0.031543
dtype: float64
\end{verbatim}

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). But if you pass a dict or Series, it need only contain a subset of the labels as keys:

\begin{verbatim}
In [195]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
                 index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
Out[195]:
       foo  three  bar
apple -0.626544  NaN  -0.351587
banana -0.138894 -0.177289  1.136249
c     0.011617  0.462215  -0.448789
durian  NaN    1.124472 -1.101558
\end{verbatim}

The \texttt{rename()} method also provides an \texttt{inplace} named parameter that is by default \texttt{False} and copies the underlying data. Pass \texttt{inplace=True} to rename the data in place. The Panel class has a related \texttt{rename_axis()} class which can rename any of its three axes.

10.8 Iteration

Because Series is array-like, basic iteration produces the values. Other data structures follow the dict-like convention of iterating over the “keys” of the objects. In short:

- **Series**: values
- **DataFrame**: column labels
- **Panel**: item labels

Thus, for example:
In [196]: for col in df:
       ....:     print(col)
       .....
       one
       three
       two

10.8.1 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:

In [197]: for item, frame in wp.iteritems():
       ....:     print(item)
       ....:     print(frame)
       .....
Item1
      A    B    C    D
2000-01-01 -1.032011 0.969818 -0.962723 1.382083
2000-01-02 -0.938794 0.669142 -0.433567 -0.273610
2000-01-03 0.680433 -0.308450 -0.276099 -1.821168
2000-01-04 -1.993606 -1.927385 -2.027924 1.624972
2000-01-05 0.551135 3.059267 0.455264 -0.030740
Item2
      A    B    C    D
2000-01-01 0.935716 1.061192 -2.107852 0.199905
2000-01-02 0.323586 -0.641630 -0.587514 0.053897
2000-01-03 0.194889 -0.381994 0.318587 2.089075
2000-01-04 -0.728293 -0.090255 -0.748199 1.318931
2000-01-05 -2.029766 0.792652 0.461007 -0.542749

10.8.2 iterrows

New in v0.7 is the ability to iterate efficiently through rows of a DataFrame with `iterrows()`. It returns an iterator yielding each index value along with a Series containing the data in each row:

In [198]: for row_index, row in df2.iterrows():
       ....:     print('a
b
c
')
      % (row_index, row))
       ....:     print('b
b
c
')
      % (row_index, row))
6
   a
   one -0.626544
two -0.351587
Name: a, dtype: float64
6
   b
   one -0.138894
two 1.136249
Name: b, dtype: float64
6
   c
   one 0.011617

10.8. Iteration

345
For instance, a contrived way to transpose the DataFrame would be:

```
In [199]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [200]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6

In [201]: print(df2.T)
   0  1  2
   x  1  2  3
   y  4  5  6

In [202]: df2_t = pd.DataFrame(dict((idx,values) for idx, values in df2.iterrows()))

In [203]: print(df2_t)
   0  1  2
   x  1  2  3
   y  4  5  6
```

**Note:** `iterrows` does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [204]: df_iter = pd.DataFrame([[1, 1.0]], columns=['x', 'y'])

In [205]: row = next(df_iter.iterrows())[1]

In [206]: print(row['x'].dtype)
   float64

In [207]: print(df_iter['x'].dtype)
   int64
```

### 10.8.3 `itertuples`

The `itertuples()` method will return an iterator yielding a tuple 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 proper.

For instance,

```
In [208]: for r in df2.itertuples():
   print(r)
   (0, 1, 4)
   (1, 2, 5)
   (2, 3, 6)
```
10.8.4 .dt accessor

Series has an accessor to succinctly return datetime like properties for the `values` of the Series, if its a datetime/period like Series. This will return a Series, indexed like the existing Series.

```python
In [209]: s = pd.Series(pd.date_range('20130101 09:10:12',periods=4))
```

In [210]: s
Out[210]:
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 [211]: s.dt.hour
Out[211]:
0   9
1   9
2   9
3   9
dtype: int64

In [212]: s.dt.second
Out[212]:
0   12
1   12
2   12
3   12
dtype: int64

In [213]: s.dt.day
Out[213]:
0   1
1   2
2   3
3   4
dtype: int64

This enables nice expressions like this:

In [214]: s[s.dt.day==2]
Out[214]:
1  2013-01-02 09:10:12
dtype: datetime64[ns]

You can easily produces tz aware transformations:

In [215]: stz = s.dt.tz_localize('US/Eastern')

In [216]: stz
Out[216]:
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: object
You can also chain these types of operations:

```python
In [218]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[218]:
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: object
```

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

```python
# period
In [219]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))
In [220]: s
Out[220]:
0 2013-01-01
1 2013-01-02
2 2013-01-03
3 2013-01-04
dtype: object
In [221]: s.dt.year
Out[221]:
0 2013
1 2013
2 2013
3 2013
dtype: int64
In [222]: s.dt.day
Out[222]:
0 1
1 2
2 3
3 4
dtype: int64
```

```python
# timedelta
In [223]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))
In [224]: s
Out[224]:
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 [225]: s.dt.days
Out[225]:
0 1
1 1
2 1
3 1
```
In [226]: s.dt.seconds
Out[226]:
0   5
1   6
2   7
3   8
dtype: int64

In [227]: s.dt.components
Out[227]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0      0       1        0          5             0              0              0
1      0       1        0          6             0              0              0
2      0       1        0          7             0              0              0
3      0       1        0          8             0              0              0

Note: Series.dt will raise a TypeError if you access with a non-datetimelike values

10.9 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

In [228]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [229]: s.str.lower()
Out[229]:
0    a
1    b
2    c
3   aaba
4   baca
5   NaN
6    caba
7    dog
8    cat
dtype: object

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.

10.10 Sorting by index and value

There are two obvious kinds of sorting that you may be interested in: sorting by label and sorting by actual values. The primary method for sorting axis labels (indexes) across data structures is the `sort_index()` method.

In [230]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
      columns=['three', 'two', 'one'])
In [231]: unsorted_df.sort_index()
Out[231]:
    three    two    one
  a     NaN  -0.351587  -0.626544
  b -0.177289     1.136249  -0.138894
  c  0.462215  -0.448789   0.011617
  d  1.124472  -1.101558     NaN

In [232]: unsorted_df.sort_index(ascending=False)
Out[232]:
    three    two    one
  d     NaN  -1.101558     NaN
  c  0.462215  -0.448789  -0.138894
  b -0.177289     1.136249  -0.138894
  a     NaN  -0.351587  -0.626544

In [233]: unsorted_df.sort_index(axis=1)
Out[233]:
    one    three    two
  a     NaN     NaN  -0.351587
  d     NaN  1.124472  -1.101558
  c  0.011617  0.462215  -0.448789
  b -0.138894 -0.177289     1.136249

DataFrame.sort_index() 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:

In [234]: df1 = pd.DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})
In [235]: df1.sort_index(by='two')
Out[235]:
    one    three    two
  0     2       5     1
  2     1       3     2
  1     1       4     3
  3     1       2     4

The by argument can take a list of column names, e.g.:

In [236]: df1[ ['one', 'two', 'three']].sort_index(by=['one','two'])
Out[236]:
    one    two    three
  0     2       3     1
  1     1       3     2
  3     1       4     3
  0     2       1     5

Series has the method order() (analogous to R's order function) which sorts by value, with special treatment of NA values via the na_position argument:

In [237]: s[2] = np.nan
In [238]: s.order()
Out[238]:
   0   A
   3  Aaba
   1   B
   4  Baca
pandas: powerful Python data analysis toolkit, Release 0.16.2

In [239]: s.order(na_position='first')
Out[239]:
2  NaN
5  NaN
0  A
3  Aaba
1  B
4  Baca
6  CABA
8  cat
7  dog
dtype: object

Note: Series.sort() sorts a Series by value in-place. This is to provide compatibility with NumPy methods which expect the ndarray.sort behavior. Series.order() returns a copy of the sorted data.

Series has the searchsorted() method, which works similar to numpy.ndarray.searchsorted().

In [240]: ser = pd.Series([1, 2, 3])

In [241]: ser.searchsorted([0, 3])
Out[241]: array([0, 2])

In [242]: ser.searchsorted([0, 4])
Out[242]: array([0, 3])

In [243]: ser.searchsorted([1, 3], side='right')
Out[243]: array([1, 3])

In [244]: ser.searchsorted([1, 3], side='left')
Out[244]: array([0, 2])

In [245]: ser = pd.Series([3, 1, 2])

In [246]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[246]: array([0, 2])

10.10.1 smallest / largest values

New in version 0.14.0.

Series has the nsmallest() and nlargest() methods which return the smallest or largest n values. For a large Series this can be much faster than sorting the entire Series and calling head(n) on the result.

In [247]: s = pd.Series(np.random.permutation(10))

In [248]: s
Out[248]:
0  7

10.10. Sorting by index and value
1  5
2  4
3  6
4  1
5  8
6  9
7  2
8  0
9  3

dtype: int32

In [249]: s.order()
Out[249]:
8  0
4  1
7  2
9  3
2  4
1  5
3  6
0  7
5  8
6  9

dtype: int32

In [250]: s.nsmallest(3)
Out[250]:
8  0
4  1
7  2

dtype: int32

In [251]: s.nlargest(3)
Out[251]:
6  9
5  8
0  7

dtype: int32

10.10.2 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 [252]: df1.columns = pd.MultiIndex.from_tuples([("a","one"),("a","two"),("b","three")])

In [253]: df1.sort_index(by=('a','two'))
Out[253]:
   a   b
one two three
3   1   2   4
2   1   3   2
1   1   4   3
0   2   5   1
10.11 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.

10.12 dtypes

The main types stored in pandas objects are `float`, `int`, `bool`, `datetime64[ns]`, `timedelta[ns]` and `object`. In addition these dtypes have item sizes, e.g. `int64` and `int32`. A convenient `dtypes` attribute for DataFrames returns a Series with the data type of each column.

```python
In [254]: 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')))  
Out[254]:  
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.028931</td>
<td>1</td>
<td>foo</td>
<td>2001-01-02</td>
<td>1</td>
<td>False</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.936706</td>
<td>1</td>
<td>foo</td>
<td>2001-01-02</td>
<td>1</td>
<td>False</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.831782</td>
<td>1</td>
<td>foo</td>
<td>2001-01-02</td>
<td>1</td>
<td>False</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

In [255]: dft.dtypes  
Out[255]:  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>float64</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>int64</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>object</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>datetime64[ns]</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>float32</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>int8</td>
<td></td>
</tr>
</tbody>
</table>

dtype: object
```

On a Series use the `dtype` attribute.

```python
In [257]: dft['A'].dtype  
Out[257]: 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).

10.11. Copying
# these ints are coerced to floats
In [258]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[258]:
   0  1
   1  2
   2  3
   3  4
   4  5
   5  6
dtype: float64

# string data forces an 'object' dtype
In [259]: pd.Series([1, 2, 3, 6., 'foo'])
Out[259]:
   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:

In [260]: dft.get_dtype_counts()
Out[260]:
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 (starting in v0.11.0). 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 [261]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [262]: df1
Out[262]:
     A
0  1.213978
1 -0.505425
2  0.254678
3 -0.744834
4  0.647650
5  0.822993
6 -1.543048
7  1.778703

In [263]: df1.dtypes
Out[263]:
A  float32
dtype: object

In [264]: df2 = pd.DataFrame(dict(A = pd.Series(np.random.randn(8), dtype='float16'), B = pd.Series(np.random.randn(8))),
......:           C = pd.Series(np.random.randn(8)))
C = pd.Series(np.array(np.random.randn(8), dtype='uint8'))

In [265]: df2
Out[265]:
   A    B    C
0 -0.123230 -1.508174  0
1  2.240234 -0.502623  0
2 -0.143799  0.529008  0
3 -2.884766  0.590536  1
4  0.027588  0.296947  0
5 -1.150391  0.007045 255
6  0.246460  0.707877  1
7 -0.455078  0.950661  0

In [266]: df2.dtypes
Out[266]:
A float16
B float64
C uint8

10.12.1 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 [267]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[267]:
a  int64
dtype: object

In [268]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[268]:
a  int64
dtype: object

In [269]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[269]:
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 [270]: frame = pd.DataFrame(np.array([1, 2]))

10.12.2 upcasting

Types can potentially be upcasted when combined with other types, meaning they are promoted from the current type (say int to float)

In [271]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [272]: df3
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*.

### 10.12.3 `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.

```python
Out[276]: df3.dtypes
```

```python
Out[276]:
A  float32
B  float64
C  float64
```

### # conversion of dtypes

```python
In [277]: df3.astype('float32').dtypes
```

```python
Out[277]:
A  float32
B  float32
```
10.12.4 object conversion

convert_objects() is a method to try to force conversion of types from the object dtype to other types. To force conversion of specific types that are number like, e.g. could be a string that represents a number, pass convert_numeric=True. This will force strings and numbers alike to be numbers if possible, otherwise they will be set to np.nan.

In [278]: df3['D'] = '1.'

In [279]: df3['E'] = '1'

In [280]: df3.convert_objects(convert_numeric=True).dtypes
Out [280]:
A  float32
B  float64
C  float64
D  float64
E   int64
dtype: object

# same, but specific dtype conversion
In [281]: df3['D'] = df3['D'].astype('float16')

In [282]: df3['E'] = df3['E'].astype('int32')

In [283]: df3.dtypes
Out [283]:
A  float32
B  float64
C  float64
D  float16
E   int32
dtype: object

To force conversion to datetime64[ns], pass convert_dates='coerce'. This will convert any datetime-like object to dates, forcing other values to NaT. This might be useful if you are reading in data which is mostly dates, but occasionally has non-dates intermixed and you want to represent as missing.

In [284]: import datetime

In [285]: s = pd.Series([datetime.datetime(2001,1,1,0,0),
                      'foo', 1.0, 1, pd.Timestamp('20010104'),
                      '20010105'], dtype='O')

In [286]: s
Out [286]:
0  2001-01-01 00:00:00
1        foo
2        1
3        1
4  2001-01-04 00:00:00
5   20010105
dtype: object
In [287]: s.convert_objects(convert_dates='coerce')
Out[287]:
   0  2001-01-01
   1       NaT
   2       NaT
   3       NaT
   4  2001-01-04
   5  2001-01-05

In addition, convert_objects() will attempt the soft conversion of any object dtypes, meaning that if all the objects in a Series are of the same type, the Series will have that dtype.

10.12.5 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 (starting in 0.11.0) See also integer na gotchas

In [288]: dfi = df3.astype('int32')
In [289]: dfi['E'] = 1
In [290]: dfi
Out[290]:
   A  B  C  D  E
0  1  -1  0  1  1
1  1   0  0  1  1
2  0   0  0  1  1
3 -3   0  1  1  1
4  0   0  0  1  1
5  0  255  1  1
6  2   0  1  1  1
7 -1   0  0  1  1

In [291]: dfi.dtypes
Out[291]:
A    int32
B    int32
C    int32
D    int32
E    int64
dtype: object

In [292]: casted = dfi[dfi>0]

In [293]: casted
Out[293]:
   A  B  C  D  E
0 NaN NaN  1  1
1 NaN NaN  1  1
2 NaN NaN  1  1
3 NaN NaN  1  1
4 NaN NaN  1  1
5 NaN NaN 255  1
6 NaN NaN  1  1
7 NaN NaN  1  1

358 Chapter 10. Essential Basic Functionality
In [294]: casted.dtypes
Out[294]:
A   float64
B   float64
C   float64
D   int32
E   int64
dtype: object

While float dtypes are unchanged.

In [295]: dfa = df3.copy()

In [296]: dfa['A'] = dfa['A'].astype('float32')

In [297]: dfa.dtypes
Out[297]:
A   float32
B   float64
C   float64
D   float16
E   int32
dtype: object

In [298]: casted = dfa[df2>0]

In [299]: casted
Out[299]:
   A     B     C     D    E
0   NaN   NaN   NaN   NaN   NaN
1  1.73481  NaN   NaN   NaN   NaN
2   NaN  0.529008  NaN   NaN   NaN
3  0.590536  1.091376  NaN   NaN   NaN
4   NaN  0.007045  255   NaN   NaN
5   NaN  0.950661  0.000000  NaN     NaN
6  2.025163  0.707877  1.000000  NaN     NaN
7   NaN   NaN   NaN   NaN   NaN

In [300]: casted.dtypes
Out[300]:
A   float32
B   float64
C   float64
D   float16
E   float64
dtype: object

10.13 Selecting columns based on dtype

New in version 0.14.1.

The select_dtypes() method implements subsetting of columns based on their dtype.

First, let’s create a DataFrame with a slew of different dtypes:

In [301]: df = pd.DataFrame({'string': list('abc'),
                         'int64': list(range(1, 4))},
                         ..:     'float64': [1.0, 2.0, 3.0])
In [302]: df['tdeltas'] = df.dates.diff()

In [303]: df['uint64'] = np.arange(3, 6).astype('u8')

In [304]: df['other_dates'] = pd.date_range('20130101', periods=3).values

In [305]: df
Out[305]:
bool1   bool2  category          dates  float64  int64  string
0   True  False          A  2015-06-13 14:04:30.984801   4    1    a
1  False  True          B  2015-06-14 14:04:30.984801   5    2    b
2  True  False          C  2015-06-15 14:04:30.984801   6    3    c
uint8    tdeltas   uint64  other_dates
0        3     NaT      3  2013-01-01
1        4    1 days     4  2013-01-02
2        5    1 days     5  2013-01-03

`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 boolean columns

In [306]: df.select_dtypes(include=[bool])
Out[306]:
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:

In [307]: df.select_dtypes(include=['bool'])
Out[307]:
bool1   bool2
0   True  False
1  False  True
2  True  False

`select_dtypes()` also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers

In [308]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[308]:
bool1   bool2  float64  int64      tdeltas
0   True  False       4    1         NaT
1  False  True        5    2    1 days
2  True  False       6    3    1 days

To select string columns you must use the object dtype:
In [309]: df.select_dtypes(include=['object'])
Out[309]:
     string
0   a
1   b
2   c

To see all the child dtypes of a generic dtype like numpy.number you can define a function that returns a tree of child dtypes:

In [310]: def subdtypes(dtype):
    ....:     subs = dtype.__subclasses__()
    ....:     if not subs:
    ....:         return dtype
    ....:     return [dtype, [subdtypes(dt) for dt in subs]]

All numpy dtypes are subclasses of numpy.generic:

In [311]: subdtypes(np.generic)
Out[311]:
[numpy.generic,
 [numpy.number,
  [[numpy.integer,
    [[numpy.signedinteger,
      [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
        [numpy.uint8,
          numpy.uint16,
          numpy.uint32,
          numpy.uint64]]],
    [numpy.inexact,
      [[numpy.floating,
        [numpy.float16, numpy.float32, numpy.float64, numpy.float96]],
        [numpy.complexfloating,
          [numpy.complex64, numpy.complex128, numpy.complex192]]]
  ],
  [numpy.flexible,
   [numpy.character, [numpy.string_, numpy.unicode_]],
   [numpy.void, [numpy.core.records.record]],
   numpy.bool_,
   numpy.datetime64,
   numpy.object_]]

Note: Pandas also defines an additional category dtype, which is not integrated into the normal numpy hierarchy and won't show up with the above function.

Note: The include and exclude parameters must be non-string sequences.
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:

```
In [1]: s = 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
         1   1
         2   1
         3   4
         4   4
         5  NaN
         6   4
         7   3
         8   3
    dtype: float64
```
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 = DataFrame(randn(3, 2), columns=[' Column A ', ' Column B '],
                   index=range(3))

In [10]: df
Out[10]:
       Column A  Column B
0    0.017428  0.039049
1  -2.240248  0.847859
2  -1.342107  0.368828
```

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    0.017428  0.039049
1  -2.240248  0.847859
2  -1.342107  0.368828
```

### 11.1 Splitting and Replacing Strings

Methods like `split` return a Series of lists:

```python
In [15]: s2 = 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
```
Elements in the split lists can be accessed using get or [] notation:

```
In [17]: s2.str.split('_').str.get(1)
Out[17]:
0   b
1   d
2  NaN
3   g
dtype: object

In [18]: s2.str.split('_').str[1]
Out[18]:
0   b
1   d
2  NaN
3   g
dtype: object
```

Easy to expand this to return a DataFrame using expand.

```
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:

```
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:

```
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
   0 1
0  a_b c
1  c_d e
2  NaN None
3  f_g h
```

Methods like replace and findall take regular expressions, too:

```
In [22]: s3 = Series(['A', 'B', 'C', 'Aaba', 'Baca',
.....:     '', np.nan, 'CABA', 'dog', 'cat'])
.....:

In [23]: s3
Out[23]:
   a
0  A
1  B
2  C
3  Aaba
4  Baca
5  NaN
6  CABA
7  dog
8  cat
```

11.1. Splitting and Replacing Strings
```python
In [24]: s3.str.replace('^a|dog', 'XX-XX ', case=False)
Out[24]:
0    A
1    B
2    C
3   XX-XX ba
4   XX-XX ca
5
6   NaN
7   XX-XX BA
8   XX-XX
type: object

In [25]: dollars = Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
Out[26]:
0   12
1  -10
2 10,000
type: object

# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
Out[27]:
0   12
1  -10
2 $10,000
type: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'\$', '-')
Out[28]:
0   12
1  -10
2 $10,000
type: object
```

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 = Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
Out[26]:
0   12
1  -10
2 10,000
type: object

# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
Out[27]:
0   12
1  -10
2 $10,000
type: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'\$', '-')
Out[28]:
0   12
1  -10
2 $10,000
type: object
```
11.2 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.

```
In [29]: s = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 
               'CABA', 'dog', 'cat'])

In [30]: s.str[0]
Out[30]:
0    A
1    B
2    C
3    A
4    B
5  NaN
6    C
7    d
8    c
dtype: object

In [31]: s.str[1]
Out[31]:
0   NaN
1   NaN
2   NaN
3    a
4    a
5  NaN
6    A
7    o
8    a
dtype: object
```

11.3 Extracting Substrings

The method `extract` (introduced in version 0.13) accepts `regular expressions` with match groups. Extracting a regular expression with one group returns a Series of strings.

```
In [32]: Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)')
Out[32]:
0  1
1  2
2  NaN
```

Elements that do not match return NaN. Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [33]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
Out[33]:
   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 results dtype always is object, even if no match is found and the result only contains NaN.

Named groups like

In [34]: Series(['a1', 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)')
Out[34]:
   letter  digit
0      a        1
1      b        2
2     NaN       NaN

and optional groups like

In [35]: Series(['a1', 'b2', '3']).str.extract('(?P<letter>[ab])?(?P<digit>\d)')
Out[35]:
   letter  digit
0      a        1
1      b        2
2     NaN        3

can also be used.

11.3.1 Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

In [36]: pattern = r'[a-z][0-9]'

In [37]: Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
Out[37]:
0   False
1   False
2   False
3   False
4   False
dtype: bool

or match a pattern:

In [38]: Series(['1', '2', '3a', '3b', '03c']).str.match(pattern, as_indexer=True)
Out[38]:
0   False
1   False
2   False
3   False
4   False
dtype: bool

The distinction between match and contains is strictness: match relies on strict re.match, while contains relies on re.search.
**Warning:** In previous versions, `match` was for extracting groups, returning a not-so-convenient Series of tuples. The new method `extract` (described in the previous section) is now preferred. This old, deprecated behavior of `match` is still the default. As demonstrated above, use the new behavior by setting `as_indexer=True`. In this mode, `match` is analogous to `contains`, returning a boolean Series. The new behavior will become the default behavior in a future release.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```python
In [39]: s4 = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [40]: s4.str.contains('A', na=False)
Out[40]:
0   True
1   False
2   False
3   True
4   False
5   False
6   True
7   False
8   False
dtype: bool
```

### 11.3.2 Creating Indicator Variables

You can extract dummy variables from string columns. For example if they are separated by a `' | '`:

```python
In [41]: s = Series(['a', 'a|b', np.nan, 'a|c'])

In [42]: s.str.get_dummies(sep='|')
Out[42]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1
```

See also `get_dummies()`.

### 11.4 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>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</td>
</tr>
<tr>
<td><code>repeat()</code></td>
<td>Duplicate values (<code>s.str.repeat(3)</code> equivalent to <code>x * 3</code>)</td>
</tr>
<tr>
<td><code>pad()</code></td>
<td>Add whitespace to left, right, or both sides of strings</td>
</tr>
</tbody>
</table>

Continued on next page
Table 11.1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.match</code> on each element, as <code>match</code> does, but return matched groups as strings for convenience.</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>
<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>
12.1 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
```

There is also an API 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, and 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`:

```
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.

### 12.2 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)`

```
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:

```
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):

```
In [19]: pd.reset_option("^display")
height has been deprecated.
line_width has been deprecated, use display.width instead (currently both are identical)

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:
```
12.3 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:

```python
import pandas as pd
pd.set_option('display.max_rows', 999)
pd.set_option('precision', 5)
```

12.4 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.

```python
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
```

```python
In [26]: pd.set_option('max_rows', 5)
In [27]: df
```

```python
Out[27]:
```
```python
pd.reset_option('max_rows')
```

display.expand_frame_repr allows for the representation of dataframes to stretch across pages, wrapped over the full column vs row-wise.

```python
In [28]: pd.reset_option('max_rows')
```

```python
In [29]: df = pd.DataFrame(np.random.randn(5,10))
```

```python
In [30]: pd.set_option('expand_frame_repr', True)
```

```python
In [31]: df
Out[31]:
```

```python
In [32]: pd.set_option('expand_frame_repr', False)
```

```python
In [33]: df
Out[33]:
```

```python
In [34]: pd.reset_option('expand_frame_repr')
```

display.large_repr lets you select whether to display dataframes that exceed \(\max_{\text{columns}}\) or \(\max_{\text{rows}}\) as a truncated frame, or as a summary.

```python
In [35]: df = pd.DataFrame(np.random.randn(10,10))
```

```python
In [36]: pd.set_option('max_rows', 5)
```

```python
In [37]: pd.set_option('large_repr', 'truncate')
```

```python
In [38]: df
Out[38]:
```

In [39]: pd.set_option('large_repr', 'info')

In [40]: df
Out[40]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 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_repr')

In [42]: pd.reset_option('max_rows')

display.max_columnwidth 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 foo bar bim uncomfortably long string
1 horse cow banana apple

In [46]: pd.set_option('max_colwidth', 6)

In [47]: df
Out[47]:
   0 1 2 3
0 foo bar bim uncomfortably long string
1 horse cow banana apple

12.4. Frequently Used Options
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'>
Int64Index: 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 [52]: pd.set_option('max_info_columns', 5)

In [53]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 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 then 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  1  1  1  0  1  NaN  1  NaN
1  1  NaN  0  0  1  1  NaN  1  0  1
2  NaN  NaN  1  1  0  NaN  0  1  NaN
3  0  1  1  NaN  0  NaN  1  NaN  NaN  0
4  0  1  0  0  1  0  0  NaN  0  0
5  NaN  NaN  NaN  NaN  NaN  0  1  NaN
6  0  1  0  0  NaN  1  NaN  NaN  0  NaN
7  NaN  NaN  1  1  NaN  1  1  1  1  NaN
8  0  0  NaN  0  NaN  1  0  0  NaN  NaN

376 Chapter 12. Options and Settings
9 NaN NaN 0 NaN NaN NaN 0 1 1 NaN

In [57]: pd.set_option('max_info_rows', 11)

In [58]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 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'>
Int64Index: 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. 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 -2.049028 2.846612 -1.208049 -0.450392 2.423905
1 0.121108 0.266916 0.843826 -0.222540 2.021981
2 -0.716789 -2.224485 -1.061137 -0.232825 0.430793
3 -0.665478 1.829807 -1.406509 1.078248 0.322774
4 0.200324 0.890024 0.194813 0.351633 0.448881

In [65]: pd.set_option('precision',4)
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 -0.198  0.966 -1.523 -0.117  0.296 -1.048
1  1.641  1.906  2.772  0.089 -1.144 -0.633
2  0.925 -0.006 -0.820 -0.601 -1.039  0.825
3 -0.824 -0.338 -0.928 -0.840  0.249 -0.109
4  0.432 -0.461  0.337 -3.208 -1.536  0.410
5 -0.673 -0.741 -0.111 -2.673  0.864  0.061

In [70]: pd.set_option('chop_threshold', .5)

In [71]: df
Out[71]:
      0    1    2    3    4    5
0  0.000  0.966 -1.523  0.000  0.000 -1.048
1  1.641  1.906  2.772  0.000 -1.144 -0.633
2  0.925  0.000 -0.820 -0.601 -1.039  0.825
3 -0.824  0.000 -0.928 -0.840  0.000  0.000
4  0.000  0.000  0.000 -3.208 -1.536  0.000
5 -0.673 -0.741  0.000 -2.673  0.864  0.000

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]:
      A    B    C
0  0.933  0.3   0
1  0.289  0.2   0
2  1.325  0.2   0
3  0.589  0.7   0
4  0.531  0.1   0
5 -1.199  0.7   0

In [76]: pd.set_option('colheader_justify', 'left')
### 12.5 List of 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 than the given threshold will be displayed as 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></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 005/01/20.</td>
</tr>
<tr>
<td>display.encoding</td>
<td>UTF-8</td>
<td></td>
</tr>
</tbody>
</table>
| display.expand_frame_repr  | True    | The callable should accept a floating point number and return a string with the desired format.
| display.float_format       | None    | Deprecated. Use display.max_rows instead.  |
| display.height             | 60      |                                                                           |
| display.large_repr         | truncate| For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated preview.
| display.line_width         | 80      | Deprecated. Use display.width instead.                                    |
| display.max_columns        | 20      | max_rows and max_columns are used in __repr__() methods to decide if to_string() or info() is used. |
| display.max_colwidth       | 50      | The maximum width in characters of a column in the repr of a pandas data structure. When True, the display.max_colwidth is used. |
| display.max_info_columns   | 100     | When True, prints and parses dates with the day first, eg 20/01/2005.    |
| display.max_info_rows      | 1690785 | When True, prints and parses dates with the year first, eg 005/01/20.   |
| display.max_rows           | 60      |                                                                           |
| display.max_seq_items      | 100     | Setting this to ‘default’ will modify the reParams used by matplotlib to give plots a more pleasing visual style. |
| display.memory_usage       | True    |                                                                           |
| display.mpl_style          | None    | Setting this to ‘default’ will modify the reParams used by matplotlib to give plots a more pleasing visual style. |
| display.multi_sparse       | True    |                                                                           |
| display.notebook_repr_html| True    | When True, IPython notebook will use html representation for pandas objects (if it is available). |
| display.pprint_nest_depth  | 3       | Controls the number of nested levels to process when pretty-printing.     |
| display.precision          | 7       | Floating point output precision (number of significant digits). This is only a suggestion. |
| display.show_dimensions    | truncate| Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only the Width of the display in characters. In case python/IPython is running in a terminal this cannot be controlled. |
| display.width              | 80      | The default Excel writer engine for ‘xls’ files.                         |
| io.excel.xls.writer        | xlwt    | The default Excel writer engine for ‘xls’ files.                         |
| io.excel.xlsx.writer       | openpyxl| The default Excel writer engine for ‘xlsx’ files.                        |
| io.hdf.default_format      | None    | default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘append’. |
| io.hdf.dropna_table       | True    | drop ALL nan rows when appending to a table.                             |
| mode.chained_assignment    | warn    | Raise an exception, warn, or no action if trying to use chained assignment. The default is warn. |
| mode.sim_interactive       | False   | Whether to simulate interactive mode for purposes of testing.            |
| mode.use_inf_as_null       | False   | True means treat None, NaN, -INF, INF as null (old way), False means None and NaN are not. |
12.6 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:

```python
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
```
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:** In 0.15.0 Index has internally been refactored to no longer subclass ndarray but instead subclass PandasObject, similarly to the rest of the pandas objects. This should be a transparent change with only very limited API implications (See the *Internal Refactoring*).

See the *MultiIndex / Advanced Indexing* for MultiIndex and more advanced indexing documentation.

See the *cookbook* for some advanced strategies.

### 13.1 Different Choices for Indexing

New in version 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 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!)
- A boolean array

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

See more at Selection by Position

- .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 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 and Advanced Hierarchical.

Getting values from an object with multi-axes selection uses the following notation (using .loc as an example, but applies to .iloc and .ix 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>

### 13.2 Deprecations

Beginning with version 0.11.0, it’s recommended that you transition away from the following methods as they may be deprecated in future versions.

- irow
- icol
- iget_value

See the section Selection by Position for substitutes.
13.3 Basics

As mentioned when introducing the data structures in the last section, the primary function of indexing with [] (a.k.a. \texttt{__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 = date_range('1/1/2000', periods=8)

In [2]: df = DataFrame(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>-1.044236</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 = 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]: -0.67368970808837025

In [8]: panel['two']

Out[8]:

<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>
</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]:
     A         B         C         D
2000-01-01 -0.282863 -1.509059 -1.135632
2000-01-02 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -0.494929  1.071804
2000-01-04 -0.706771 -1.039575  0.271860
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690 -1.478427  0.524988
2000-01-07  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -1.344312  0.844885
```

```
In [10]: df[['B', 'A']] = df[['A', 'B']]
```

```
In [11]: df
```
```
Out[11]:
     B         A         C         D
2000-01-01 -0.282863 -0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04 -0.706771 -0.706771 -1.039575  0.271860
2000-01-05  0.567020  0.567020  0.276232 -1.087401
2000-01-06 -0.673690 -0.673690 -1.478427  0.524988
2000-01-07 -1.157892 -1.157892 -1.715002 -1.039268
2000-01-08 -1.344312 -1.344312 -1.344312  0.844885
```

You may find this useful for applying a transform (in-place) to a subset of the columns.

### 13.4 Attribute Access

You may access an index on a `Series`, column on a `DataFrame`, and a item on a `Panel` directly as an attribute:

```
In [12]: sa = Series([1,2,3],index=list('abc'))
```

```
In [13]: dfa = df.copy()
```

```
In [14]: sa.b
```
```
Out[14]: 2
```

```
In [15]: dfa.A
```
```
Out[15]:
Freq: D, Name: A, dtype: float64
```

```
In [16]: panel.one
```

---

**Chapter 13. Indexing and Selecting Data**

384
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 fails silently, creating a new attribute rather than a new column.

```python
In [17]: sa.a = 5

In [18]: sa
Out[18]:
   a  b
0  5  2
1  3

dtype: int64

In [19]: dfa.A = list(range(len(dfa.index)))  # ok if A already exists

In [20]: dfa
Out[20]:
     A  B    C  D
2000-01-01  0  0.469112 -1.509059 -1.135632
2000-01-02  1  1.212112  0.119209 -1.044236
2000-01-03  2  0.861849 -0.494929  1.071804
2000-01-04  3  0.721555 -1.039575  0.271860
2000-01-05  4  0.424972  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.404705 -1.715002 -1.039268
2000-01-08  7  0.370647 -1.344312  0.844885

In [21]: dfa['A'] = list(range(len(dfa.index)))  # use this form to create a new column

In [22]: dfa
Out[22]:
     A  B    C  D
2000-01-01  0  0.469112 -1.509059 -1.135632
2000-01-02  1  1.212112  0.119209 -1.044236
2000-01-03  2  0.861849 -0.494929  1.071804
2000-01-04  3  0.721555 -1.039575  0.271860
2000-01-05  4  0.424972  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.404705 -1.715002 -1.039268
2000-01-08  7  0.370647 -1.344312  0.844885

13.4. Attribute Access
Warning:

- You can use this access only if the index element is a valid python identifier, e.g. s['l'] 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.
- The Series/Panel accesses are available starting in 0.13.0.

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:

```python
In [23]: x = pd.DataFrame({'x': [1, 2, 3], 'y': [3, 4, 5]})
In [24]: x.iloc[1] = dict(x=9, y=99)
```

```python
In [25]: x
Out[25]:
   x  y
0  1  3
1  9  99
2  3  5
```

### 13.5 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the Selection by Position section detailing the .iloc method. For now, we explain the semantics of slicing using the [] operator.

With Series, the syntax works exactly as with an ndarray, returning a slice of the values and the corresponding labels:

```python
In [26]: s[:5]
Out[26]:
   2000-01-01   -0.282863
   2000-01-02   -0.173215
   2000-01-03   -2.104569
   2000-01-04   -0.706771
   2000-01-05    0.567020
Freq: D, Name: A, dtype: float64
```

```python
In [27]: s[::2]
Out[27]:
   2000-01-01   -0.282863
   2000-01-03   -2.104569
   2000-01-05    0.567020
   2000-01-07    0.577046
Freq: 2D, Name: A, dtype: float64
```

```python
In [28]: s[::-1]
Out[28]:
   2000-01-08   -1.157892
   2000-01-07    0.577046
   2000-01-06    0.113648
   2000-01-05    0.567020
   2000-01-04   -0.706771
```
2000-01-03   -2.104569  
2000-01-02   -0.173215  
2000-01-01   -0.282863 
Freq: -1D, Name: A, dtype: float64 

Note that setting works as well: 

```python
In [29]: s2 = s.copy()
In [30]: s2[:5] = 0
In [31]: s2
```

```
Out[31]:
2000-01-01  0.000000
2000-01-02  0.000000
2000-01-03  0.000000
2000-01-04  0.000000
2000-01-05  0.000000
2000-01-06  0.113648
2000-01-07  0.577046
2000-01-08  -1.157892 
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 [32]: df[:3]
Out[32]:
```

```
A   B    C    D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632 
2000-01-02 -0.173215  1.212112  0.119209 -1.044236 
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804 
```

```python
In [33]: df[::-1]
Out[33]:
```

```
A   B    C    D
2000-01-08  -1.157892  -0.370647  -0.344312  0.844885 
2000-01-07   0.577046  0.404705  -1.715002  -1.039268 
2000-01-06   0.113648  -0.673690  -1.478427  0.524988 
2000-01-05   0.567020  -0.424972  0.276232  -1.087401 
2000-01-04  -0.706771  0.721555  -1.039575  0.271860 
2000-01-03  -2.104569  -0.861849  -0.494929  1.071804 
2000-01-02  -0.173215  1.212112  0.119209  -1.044236 
2000-01-01   -2.104569  0.469112  -1.509059  -1.135632 
```

### 13.6 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 [34]: df1 = DataFrame(np.random.randn(5,4), columns=list('ABCD'), index=date_range('20130101',periods=5))

In [35]: df1
Out[35]:
   A         B         C         D
0  2013-01-01  1.075770  -0.109050  1.643563  -1.469388
1  2013-01-02   0.357021  -0.674600  -1.776904   0.968914
2  2013-01-03  -1.294524   0.413738   0.276662  -0.472035
3  2013-01-04  -0.013960  -0.362543  -0.006154  -0.923061
4  2013-01-05   0.895717   0.805244  -1.206412   2.565646

In [4]: df1.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 [36]: df1.loc['20130102':'20130104']
Out[36]:
   A         B         C         D
0  2013-01-02   0.357021  -0.674600  -1.776904  -0.968914
1  2013-01-03  -1.294524   0.413738   0.276662  -0.472035
2  2013-01-04  -0.013960  -0.362543  -0.006154  -0.923061

pandas provides a suite of methods in order to have purely label based indexing. This is a strict inclusion based protocol. At least 1 of the labels for which you ask, must be in the index or a KeyError will be raised! When slicing, the start bound is included, AND the stop bound is included. 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!)
- A boolean array

In [37]: s1 = Series(np.random.randn(6),index=list('abcdef'))

In [38]: s1
Out[38]:
   a  1.431256
   b  1.340309
   c -1.170299
   d -0.226169
   e  0.410835
   f  0.813850
dtype: float64

In [39]: s1.loc['c']
Out[39]:
   c  -1.170299
c   -1.170299
 d   -0.226169
 e    0.410835
 f    0.813850
dtype: float64

In [40]: s1.loc['b']
Out[40]: 1.3403088497993827

Note that setting works as well:

In [41]: s1.loc['c'] = 0
In [42]: s1
Out[42]:
       a    b    c    d    e    f
0  1.431256  1.340309  0.0  0.0  0.0  0.0

With a DataFrame

In [43]: df1 = DataFrame(np.random.randn(6,4),
                     index=list('abcdef'),
                     columns=list('ABCD'))

In [44]: df1
Out[44]:
     A         B         C         D
a  0.132003 -0.827317  -0.076467 -1.187678
b  1.130127 -1.436737  -1.413681  1.607920
c  1.024180  0.569605   0.875906 -2.211372
d  0.974466  2.006747   0.410001  0.078638
e  0.545952  1.219217  -1.226825  0.769804
f -1.281247 -0.727707  -0.121306 -0.097883

In [45]: df1.loc[['a','b','d'],:]
Out[45]:
     A         B         C         D
a  0.132003 -0.827317  -0.076467 -1.187678
b  1.130127 -1.436737  -1.413681  1.607920
d  0.974466 -2.006747   0.410001  0.078638

Accessing via label slices

In [46]: df1.loc[['a','b','d'],:]
Out[46]:
     A         B         C         D
a  0.132003 -0.827317  -0.076467 -1.187678
b  1.130127 -1.436737  -1.413681  1.607920
d  0.974466  2.006747   0.410001  0.078638

e  0.545952 -1.219217  -1.226825  0.769804
f -1.281247 -0.727707  -0.121306 -0.097883

e  0.545952 -1.219217  -1.226825  0.769804
f -1.281247 -0.727707  -0.121306 -0.097883

For getting a cross section using a label (equiv to df.xs('a'))
In [47]: df1.loc['a']
Out[47]:
   A  0.132003
   B -0.827317
   C -0.076467
   D -1.187678
Name: a, dtype: float64

For getting values with a boolean array

In [48]: df1.loc['a'] > 0
Out[48]:
   A   True
   B  False
   C  False
   D  False
Name: a, dtype: bool

In [49]: df1.loc[:, df1.loc['a'] > 0]
Out[49]:
   A
a  0.132003
b  1.130127
c  1.024180
d  0.974466
e  0.545952
f -1.281247

For getting a value explicitly (equiv to deprecated df.get_value('a', 'A'))

# this is also equivalent to `df1.at['a', 'A']`

In [50]: df1.loc['a', 'A']
Out[50]: 0.13200317033032927

13.7 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 a 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

In [51]: s1 = Series(np.random.randn(5), index=list(range(0,10,2)))

In [52]: s1
Out[52]:

390 Chapter 13. Indexing and Selecting Data
In [53]: sl.iloc[:3]
Out[53]:
0  0.695775
2  0.341734
4  0.959726
dtype: float64

In [54]: sl.iloc[3]
Out[54]: -1.1103361028911667

Note that setting works as well:

In [55]: sl.iloc[:3] = 0

In [56]: sl
Out[56]:
0 0.000000
2 0.000000
4 0.000000
6 -1.110336
8 -0.619976
dtype: float64

With a DataFrame

In [57]: df1 = DataFrame(np.random.randn(6,4),
   ...:                    index=list(range(0,12,2)),
   ...:                    columns=list(range(0,8,2)))

In [58]: df1
Out[58]:
       0  2  4  6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
6 -0.826591 -0.345352  1.314232  0.690579
8  0.995761  2.396780  0.014871  3.357427
10 -0.317441 -1.236269  0.896171 -0.487602

Select via integer slicing

In [59]: df1.iloc[:3]
Out[59]:
       0  2  4  6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161

In [60]: df1.iloc[1:5,2:4]
Out[60]:
       4  6
Select via integer list

```
In [61]: df1.iloc[[1,3,5],[1,3]]
Out[61]:
2   6
2 -0.154951 -2.179861
6 -0.345352  0.690579
10 -1.236269 -0.487602
```

For slicing rows explicitly (equiv to deprecated `df.irow(slice(1,3))`).

```
In [62]: df1.iloc[1:3, :]
Out[62]:
0   2   4   6
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
```

For slicing columns explicitly (equiv to deprecated `df.icol(slice(1,3))`).

```
In [63]: df1.iloc[:, 1:3]
Out[63]:
2   4
0 -0.732339  0.687738
2 -0.154951  0.301624
4 -0.954208  1.462696
6 -0.345352  1.314232
8  2.396780  0.014871
10 -1.236269  0.896171
```

For getting a scalar via integer position (equiv to deprecated `df.get_value(1,1)`)

```
# this is also equivalent to `df1.iat[1,1]`
In [64]: df1.iloc[1,1]
Out[64]: -0.15495077442490321
```

For getting a cross section using an integer position (equiv to `df.xs(1)`)

```
In [65]: df1.iloc[1]
Out[65]:
0   0.403310
2  -0.154951
4   0.301624
6  -2.179861
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/Numpy.

```
# these are allowed in python/numpy.
# Only works in Pandas starting from v0.14.0.
In [66]: x = list('abcdef')
In [67]: x[4:10]
Out[67]: ['d', 'e', 'f']
In [68]: x[4:10]
```
Out[68]: ['e', 'f']

In [69]: x[8:10]
Out[69]: []

In [70]: s = Series(x)

In [71]: s
Out[71]:
0 a
1 b
2 c
3 d
4 e
5 f
dtype: object

In [72]: s.iloc[4:10]
Out[72]:
4 e
5 f
dtype: object

In [73]: s.iloc[8:10]
Out[73]: Series([], dtype: object)

Note: Prior to v0.14.0, iloc would not accept out of bounds indexers for slices, e.g. a value that exceeds the length of the object being indexed.

Note that this could result in an empty axis (e.g. an empty DataFrame being returned)

In [74]: dfl = DataFrame(np.random.randn(5,2),columns=list('AB'))

In [75]: dfl
Out[75]:
     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 [76]: dfl.iloc[:,2:3]
Out[76]:
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [77]: dfl.iloc[:,1:3]
Out[77]:
     B
0 -2.182937
1  0.084844
2  1.519970
3  0.600178
4  0.132885

In [78]: dfl.iloc[4:6]
A single indexer that is out of bounds will raise an `IndexError`. A list of indexers where any element is out of bounds will raise an `IndexError`:

```python
dfl.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds
```

```python
dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

### 13.8 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 [79]: s = Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1 row.
In [80]: s.sample()
Out[80]:
5 5
dtype: int64

# One may specify either a number of rows:
In [81]: s.sample(n=3)
Out[81]:
1 1
5 5
2 2
dtype: int64

# Or a fraction of the rows:
In [82]: s.sample(frac=0.5)
Out[82]:
4 4
5 5
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 [83]: s = Series([0, 1, 2, 3, 4, 5])

# Without replacement (default):
In [84]: s.sample(n=6, replace=False)
Out[84]:
4 4
1 1
3 3
5 5
0 0
2 2
```
# With replacement:

```python
In [85]: s.sample(n=6, replace=True)
Out[85]:
2  2
0  0
3  3
2  2
4  4
4  4
```

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 [86]: s = Series([0,1,2,3,4,5])
In [87]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]
In [88]: s.sample(n=3, weights=example_weights)
```

```
Out[88]:
2  2
5  5
4  4
```

```python
# Weights will be re-normalized automatically
In [89]: example_weights2 = [0.5, 0, 0, 0, 0, 0]
In [90]: s.sample(n=1, weights=example_weights2)
Out[90]:
0  0
```

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.

```python
In [91]: df2 = DataFrame({'col1':[9,8,7,6], 'weight_column':[0.5, 0.4, 0.1, 0]})
In [92]: df2.sample(n=3, weights='weight_column')
```

```python
Out[92]:
col1  weight_column
0  9    0.5
1  8    0.4
2  7    0.1
```

`sample` also allows users to sample columns instead of rows using the `axis` argument.

```python
In [93]: df3 = DataFrame({'col1':[1,2,3], 'col2':[2,3,4]})
In [94]: df3.sample(n=1, axis=1)
```

```python
Out[94]:
col
0  1
1  2
```
Finally, one can also set a seed for `sample`'s random number generator using the `random_state` argument, which will accept either an integer (as a seed) or a numpy RandomState object.

```
In [95]: df4 = DataFrame({'col1':[1,2,3], 'col2':[2,3,4]})

# With a given seed, the sample will always draw the same rows.
In [96]: df4.sample(n=2, random_state=2)
Out[96]:
   col1 col2
0   2    3
1   1    2

In [97]: df4.sample(n=2, random_state=2)
Out[97]:
   col1 col2
0   2    3
1   1    2
```

### 13.9 Setting With Enlargement

New in version 0.13.

The `.loc/.ix/[]` operations can perform enlargement when setting a non-existant key for that axis.

In the `Series` case this is effectively an appending operation

```
In [98]: se = Series([1,2,3])

In [99]: se
Out[99]:
0    1
1    2
2    3
dtype: int64

In [100]: se[5] = 5.

In [101]: se
Out[101]:
0    1
1    2
2    3
5    5
dtype: float64
```

A `DataFrame` can be enlarged on either axis via `.loc`

```
In [102]: dfi = DataFrame(np.arange(6).reshape(3,2),
      ...:                     columns=['A','B'])

In [103]: dfi
Out[103]:
     A  B
0  0   1
```
1 2 3
2 4 5

In [104]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [105]: dfi
Out[105]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4

This is like an append operation on the DataFrame.

In [106]: dfi.loc[3] = 5

In [107]: dfi
Out[107]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5

13.10 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 [108]: s.iat[5]
Out[108]: 5

In [109]: df.at[dates[5], 'A']
Out[109]: 0.11364840968888545

In [110]: df.iat[3, 0]
Out[110]: -0.70677113363008437

You can also set using these same indexers.

In [111]: df.at[dates[5], 'E'] = 7
In [112]: df.iat[3, 0] = 7

at may enlarge the object in-place as above if the indexer is missing.

In [113]: df.at[dates[-1]+1, 0] = 7

In [114]: df
Out[114]:
    A    B    C   D    E
0 2000-01-01 0.282863 0.469112 -1.509059 -1.135632 NaN  NaN
0 2000-01-02 -0.173215 1.212112 0.119209 -1.044236 NaN  NaN
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 [115]: s = Series(range(-3, 4))
In [116]: s
Out[116]:
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int32
In [117]: s[s > 0]
Out[117]:
4    1
5    2
6    3
dtype: int32
In [118]: s[(s < -1) | (s > 0.5)]
Out[118]:
0   -3
1   -2
4    1
5    2
6    3
dtype: int32
In [119]: s[~(s < 0)]
Out[119]:
3    0
4    1
5    2
6    3
dtype: int32
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame’s index (for example, something derived from one of the columns of the DataFrame):
In [120]: df[df['A'] > 0]
Out[120]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-04</td>
<td>7.000000</td>
<td>0.721555</td>
<td>-1.039575</td>
<td>0.271860</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
<td>0.276232</td>
<td>-1.087401</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
<td>-0.673690</td>
<td>-1.478427</td>
<td>0.524988</td>
<td>7</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
<td>-0.404705</td>
<td>-1.715002</td>
<td>-1.039268</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

List comprehensions and map method of Series can also be used to produce more complex criteria:

In [121]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                            'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                            'c' : randn(7)})

# only want 'two' or 'three'
In [122]: criterion = df2['a'].map(lambda x: x.startswith('t'))
In [123]: df2[criterion]
Out[123]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>two</td>
<td>y</td>
<td>1.450520</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td>x</td>
<td>0.206053</td>
</tr>
<tr>
<td>4</td>
<td>two</td>
<td>y</td>
<td>-0.251905</td>
</tr>
</tbody>
</table>

# equivalent but slower
In [124]: df2[[x.startswith('t') for x in df2['a']]]
Out[124]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>two</td>
<td>y</td>
<td>1.450520</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td>x</td>
<td>0.206053</td>
</tr>
<tr>
<td>4</td>
<td>two</td>
<td>y</td>
<td>-0.251905</td>
</tr>
</tbody>
</table>

# Multiple criteria
In [125]: df2[criterion & (df2['b'] == 'x')]
Out[125]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>three</td>
<td>x</td>
<td>0.206053</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 [126]: df2.loc[criterion & (df2['b'] == 'x'),'b':'c']
Out[126]:

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>x</td>
<td>0.206053</td>
</tr>
</tbody>
</table>

### 13.12 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 [127]: s = Series(np.arange(5),index=np.arange(5)[::-1],dtype='int64')

In [128]: s
Out[128]:

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>0</th>
</tr>
</thead>
</table>

13.12. Indexing with isin
In [129]: s.isin([2, 4, 6])
Out[129]:
4  False
3  False
2  True
1  False
0  True
dtype: bool

In [130]: s[s.isin([2, 4, 6])]
Out[130]:
2 2
0 4
dtype: int64

The same method is available for Index objects and is useful for the cases when you don’t know which of the sought labels are in fact present:

In [131]: s[s.index.isin([2, 4, 6])]
Out[131]:
4 0
2 2
dtype: int64

# compare it to the following
In [132]: s[[2, 4, 6]]
Out[132]:
2 2
4 0
6 NaN
dtype: float64

In addition to that, MultiIndex allows selecting a separate level to use in the membership check:

In [133]: s_mi = Series(np.arange(6),
    index=pd.MultiIndex.from_product([range(2), ['a', 'b', 'c']]))

In [134]: s_mi
Out[134]:
a 0
b 1
c 2
a 3
b 4
c 5
dtype: int32

In [135]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
Out[135]:
0  c 2
1  a 3
dtype: int32

# compare it to the following
In [136]: s_mi[[1, 'a'), (2, 'b'), (0, 'c')])
Out[136]:
0  c 2
1  a 3
dtype: int32
In [136]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
Out[136]:
          0  
    a  0  
    c  2  
          1  
    a  3  
    c  5  
dtype: int32

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.

In [137]: df = DataFrame({
                           'vals': [1, 2, 3, 4],
                           'ids': ['a', 'b', 'f', 'n'],
                           'ids2': ['a', 'n', 'c', 'n']})

In [138]: values = ['a', 'b', 1, 3]

In [139]: df.isin(values)
Out[139]:
    ids  ids2  vals
0  True  True  True
1  True  False  False
2  False  False  True
3  False  False  False

Oftentimes you’ll want to match certain values with certain columns. Just make values a dict where the key is the column, and the value is a list of items you want to check for.

In [140]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [141]: df.isin(values)
Out[141]:
    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 [142]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}

In [143]: row_mask = df.isin(values).all(1)

In [144]: df[row_mask]
Out[144]:
    ids  ids2  vals
0  a  a  1

13.13 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 [145]: s[s > 0]
Out[145]:
3  1
2  2
1  3
0  4
dtype: int64

To return a Series of the same shape as the original
In [146]: s.where(s > 0)
Out[146]:
4   NaN
3   1
2   2
1   3
0   4
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 [147]: df[df < 0]
Out[147]:
     A      B     C      D
2000-01-01  NaN   NaN  -0.863838  NaN
2000-01-02 -1.048089 -0.025747 -0.988387  NaN
2000-01-03  NaN   NaN   NaN   -0.055758
2000-01-04  NaN  -0.489682   NaN  -0.034571
2000-01-05 -2.484478 -0.281461   NaN   NaN
2000-01-06  NaN  -0.977349   NaN  -0.064034
2000-01-07 -1.282782   NaN  -1.071357   NaN
2000-01-08  NaN   NaN   NaN  -0.744471

In addition, where takes an optional other argument for replacement of values where the condition is False, in the
returned copy.
In [148]: df.where(df < 0, -df)
Out[148]:
     A      B     C      D
2000-01-01  1.266143 -0.299368  0.863838  0.408204
2000-01-02  1.048089 -0.025747  0.988387  0.094055
2000-01-03  1.262731 -1.289997 -0.082423 -0.055758
2000-01-04  0.536580 -0.489682  0.369374 -0.034571
2000-01-05  2.484478 -0.281461  0.307111 -0.109121
2000-01-06  1.126203 -0.977349 -1.474071 -0.064034
2000-01-07  1.282782 -0.781836 -1.071357 -0.441153
2000-01-08  2.353925 -0.583787 -0.221471 -0.744471

You may wish to set values based on some boolean criteria. This can be done intuitively like so:
In [149]: s2 = s.copy()

In [150]: s2[s2 < 0] = 0

In [151]: s2
Out[151]:
4   0
3   1

Chapter 13. Indexing and Selecting Data
In [152]: df2 = df.copy()

In [153]: df2[df2 < 0] = 0

In [154]: df2

```
Out[154]:
          A       B       C       D
2000-01-01 1.266143 0.299368 0.000000 0.408204
2000-01-02 0.000000 0.000000 0.000000 0.094055
2000-01-03 1.262731 1.289997 0.082423 0.000000
2000-01-04 0.536580 0.000000 0.369374 0.000000
2000-01-05 0.000000 0.000000 0.030711 0.109121
2000-01-06 1.126203 0.000000 1.474071 0.000000
2000-01-07 0.000000 0.781836 0.000000 0.441153
2000-01-08 2.353925 0.583787 0.221471 0.000000
```

By default, where returns a modified copy of the data. There is an optional parameter inplace so that the original data can be modified without creating a copy:

In [155]: df_orig = df.copy()

In [156]: df_orig.where(df > 0, -df, inplace=True);

In [157]: df_orig

```
Out[157]:
          A       B       C       D
2000-01-01 1.266143 0.299368 -0.863838 0.408204
2000-01-02 -1.048089 -0.025747 -0.988387 3.000000
2000-01-03 3.000000 3.000000 3.000000 -0.055758
2000-01-04 0.536580 0.489682 0.369374 0.034571
2000-01-05 2.484478 0.281461 1.474071 0.064034
2000-01-06 1.126203 0.977349 1.474071 0.064034
2000-01-07 1.282782 0.781836 1.071357 0.441153
2000-01-08 2.353925 0.583787 0.221471 0.744471
```

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 .ix (but on the contents rather than the axis labels)

In [158]: df2 = df.copy()

In [159]: df2[df2[1:4] > 0 ] = 3

In [160]: df2

```
Out[160]:
          A       B       C       D
2000-01-01 1.266143 0.299368 -0.863838 0.408204
2000-01-02 -1.048089 -0.025747 -0.988387 3.000000
2000-01-03 3.000000 3.000000 3.000000 -0.055758
2000-01-04 3.000000 -0.489682 3.000000 -0.034571
2000-01-05 -2.484478 -0.281461 0.030711 0.109121
2000-01-06 1.126203 -0.977349 1.474071 -0.064034
2000-01-07 -1.282782 0.781836 -1.071357 0.441153
2000-01-08 2.353925 0.583787 0.221471 0.744471
```

13.13. The where() Method and Masking
New in version 0.13.

Where can also accept axis and level parameters to align the input when performing the where.

```
In [161]: df2 = df.copy()

In [162]: df2.where(df2>0,df2['A'],axis='index')
Out[162]:
   A          B         C           D
2000-01-01  1.266143  0.299368  1.266143  0.408204
2000-01-02 -1.048089 -1.048089 -1.048089  0.094055
2000-01-03  1.262731  1.289997  0.082423  1.262731
2000-01-04  0.536580  0.536580  0.369374  0.536580
2000-01-05 -2.484478 -2.484478  0.030711  0.109121
2000-01-06  1.126203  1.126203  1.474071  1.126203
2000-01-07 -1.282782  0.781836 -1.282782  0.441153
2000-01-08  2.353925  0.583787  0.221471  2.353925
```

This is equivalent (but faster than) the following.

```
In [163]: df2 = df.copy()

In [164]: df.apply(lambda x, y: x.where(x>0,y), y=df['A'])
Out[164]:
   A          B         C           D
2000-01-01  1.266143  0.299368  1.266143  0.408204
2000-01-02 -1.048089 -1.048089 -1.048089  0.094055
2000-01-03  1.262731  1.289997  0.082423  1.262731
2000-01-04  0.536580  0.536580  0.369374  0.536580
2000-01-05 -2.484478 -2.484478  0.030711  0.109121
2000-01-06  1.126203  1.126203  1.474071  1.126203
2000-01-07 -1.282782  0.781836 -1.282782  0.441153
2000-01-08  2.353925  0.583787  0.221471  2.353925
```

**mask**

mask is the inverse boolean operation of where.

```
In [165]: s.mask(s >= 0)
Out[165]:
   4  NaN
   3  NaN
   2  NaN
   1  NaN
   0  NaN
dtype: float64

In [166]: df.mask(df >= 0)
Out[166]:
   A          B         C           D
2000-01-01  NaN         NaN  -0.863838 NaN
2000-01-02 -1.048089 -0.025747 -0.988387 NaN
2000-01-03  NaN         NaN         NaN -0.055758
2000-01-04  NaN   -0.489682  NaN  -0.034571
2000-01-05 -2.484478 -0.281461  NaN  NaN
2000-01-06  NaN   -0.977349  NaN  -0.064034
2000-01-07 -1.282782  NaN  -1.071357 NaN
2000-01-08  NaN         NaN         NaN -0.744471
```

404 Chapter 13. Indexing and Selecting Data
13.14 The `query()` Method (Experimental)

New in version 0.13.

`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:

```python
In [167]: n = 10

In [168]: df = DataFrame(rand(n, 3), columns=list('abc'))

In [169]: df
Out[169]:
   a    b    c
0  0.191519  0.622109  0.437728
1  0.785359  0.779976  0.272593
2  0.276464  0.801872  0.958139
3  0.875933  0.357817  0.500995
4  0.683463  0.712702  0.370251
5  0.561196  0.503083  0.013768
6  0.772827  0.882641  0.364886
7  0.615396  0.075381  0.397203
8  0.933140  0.651378  0.397203
9  0.788730  0.316836  0.568099

# pure python
In [170]: df[(df.a < df.b) & (df.b < df.c)]
Out[170]:
   a    b    c
2  0.276464  0.801872  0.958139

# query
In [171]: df.query('(a < b) & (b < c)')
Out[171]:
   a    b    c
2  0.276464  0.801872  0.958139

Do the same thing but fall back on a named index if there is no column with the name `a`.

In [172]: df = DataFrame(randint(n / 2, size=(n, 2)), columns=list('bc'))

In [173]: df.index.name = 'a'

In [174]: df
Out[174]:
   b    c
   a
0  2  3
1  4  1
2  4  0
3  4  1
4  1  4
5  1  4
6  0  1
7  0  0
8  4  0
9  4  2
In [175]: df.query('a < b and b < c')
Out[175]:
   b  c
0 2  3

If instead you don’t want to or cannot name your index, you can use the name index in your query expression:

In [176]: df = DataFrame(randint(n, size=(n, 2)), columns=list('bc'))

In [177]: df
Out[177]:
   b  c
0 3  1
1 2  5
2 5  6
3 4  3
4 5  6
5 6  6
6 4  6
7 2  4
8 2  7
9 9  7

In [178]: df.query('index < b < c')
Out[178]:
   b  c
1 2  5
3 6  7

Note: If the name of your index overlaps with a column name, the column name is given precedence. For example,

In [179]: df = DataFrame({'a': randint(5, size=5)})

In [180]: df.index.name = 'a'

In [181]: df.query('a > 2')  # uses the column 'a', not the index
Out[181]:
   a
0 3
3 4

You can still use the index in a query expression by using the special identifier ‘index’:

In [182]: df.query('index > 2')
Out[182]:
   a
0 3
3 4
4 1

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.
13.14.1 MultiIndex query() Syntax

You can also use the levels of a DataFrame with a MultiIndex as if they were columns in the frame:

In [183]: import pandas.util.testing as tm
In [184]: n = 10
In [185]: colors = tm.choice(['red', 'green'], size=n)
In [186]: foods = tm.choice(['eggs', 'ham'], size=n)
In [187]: colors
Out[187]:
array(['red', 'green', 'red', 'green', 'red', 'green', 'red', 'green',
       'green', 'green'],
       dtype='|S5')
In [188]: foods
Out[188]:
array(['ham', 'eggs', 'ham', 'ham', 'ham', 'eggs', 'eggs', 'eggs',
       'ham', 'eggs'],
       dtype='|S4')
In [189]: index = MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
In [190]: df = DataFrame(randn(n, 2), index=index)
In [191]: df
Out[191]:
     0       1
color food
red  ham  0.157622 -0.293555
    eggs  0.111560  0.597679
  ham -1.270093  0.120949
green ham -0.193898  1.804172
   eggs -0.234694  0.939908
  ham -0.171520 -0.153055
red eggs -0.363095 -0.067318
green eggs  1.444721  0.325771
  ham -0.855732 -0.697595
eggs -0.276134 -1.258759
ham -0.855732 -0.697595
eggs -0.276134 -1.258759

In [192]: df.query('color == "red"')
Out[192]:
     0       1
color food
red  ham  0.157622 -0.293555
  ham -1.270093  0.120949
green ham -0.193898  1.804172
   eggs -0.234694  0.939908
red eggs -0.363095 -0.067318
green eggs  1.444721  0.325771
  ham -0.855732 -0.697595
eggs -0.276134 -1.258759
ham -0.855732 -0.697595
eggs -0.276134 -1.258759

If the levels of the MultiIndex are unnamed, you can refer to them using special names:

In [193]: df.index.names = [None, None]
In [194]: df
Out[194]:
     0       1
color food
red  ham  0.157622 -0.293555
    eggs  0.111560  0.597679
  ham -1.270093  0.120949
green ham -0.193898  1.804172
   eggs -0.234694  0.939908
  ham -0.171520 -0.153055
red eggs -0.363095 -0.067318
green eggs  1.444721  0.325771
  ham -0.855732 -0.697595
eggs -0.276134 -1.258759
The convention is `ilevel_0`, which means “index level 0” for the 0th level of the index.

### 13.14.2 `query()` Use Cases

A use case for `query()` is when you have a collection of `DataFrame` objects that have a subset of column names (or index levels/names) in common. You can pass the same query to both frames without having to specify which frame you’re interested in querying.

```
In [196]: df = DataFrame(rand(n, 3), columns=list('abc'))
In [197]: df
Out[197]:
     a       b       c
0  0.972113  0.046532  0.917354
1  0.158930  0.943383  0.763162
2  0.053878  0.254082  0.927973
3  0.838312  0.156925  0.690776
4  0.366946  0.937473  0.613365
5  0.699350  0.502946  0.711111
6  0.134386  0.828932  0.742846
7  0.457034  0.079103  0.373047
8  0.933636  0.418725  0.234212
9  0.572485  0.572111  0.416893
```

```
In [198]: df2 = DataFrame(rand(n + 2, 3), columns=df.columns)
In [199]: df2
Out[199]:
     a       b       c
0  0.625883  0.220362  0.622059
1  0.477672  0.974342  0.772985
2  0.027139  0.221022  0.120328
3  0.175274  0.429462  0.657769
4  0.565899  0.569035  0.654196
5  0.368558  0.952385  0.196770
6  0.849930  0.960458  0.381118
7  0.330936  0.260923  0.665491
8  0.181795  0.376800  0.014259
```
9  0.339135  0.401351  0.467574
10 0.652106  0.997192  0.517462
11 0.403612  0.058447  0.045196

In [200]: expr = '0.0 <= a <= c <= 0.5'

In [201]: map(lambda frame: frame.query(expr), [df, df2])
Out[201]:
[Empty DataFrame
  Columns: [a, b, c]
  Index: [], a b c
  2  0.027139  0.221022  0.120328
  9  0.339135  0.401351  0.467574]

13.14.3 query() Python versus pandas Syntax Comparison

Full numpy-like syntax

In [202]: df = DataFrame(randint(n, size=(n, 3)), columns=list('abc'))
In [203]: df
Out[203]:
   a  b  c
0  5  3  8
1  8  8  1
2  3  6  8
3  9  1  5
4  8  4  1
5  1  1  2
6  3  4  2
7  1  9  4
8  0  0  2
9  1  2  5

In [204]: df.query('a < b & b < c')
Out[204]:
   a  b  c
2  3  6  8
9  1  2  5

In [205]: df[(df.a < df.b) & (df.b < df.c)]
Out[205]:
   a  b  c
2  3  6  8
9  1  2  5

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than &/)

In [206]: df.query('a < b and b < c')
Out[206]:
   a  b  c
2  3  6  8
9  1  2  5

Use English instead of symbols

In [207]: df.query('a < b and b < c')
Out[207]:
Pretty close to how you might write it on paper

```
In [208]: df.query('a < b < c')
Out[208]:
    a  b  c
0  2  3  6
1  9  1  2
```

### 13.14.4 The `in` and `not in` operators

`query()` also supports special use of Python’s `in` and `not in` 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 [209]: df = DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbccccc'),
                        'c': randint(5, size=12), 'd': randint(9, size=12)})

In [210]: df
Out[210]:
    a  b  c  d
0  a  a  1  7
1  a  a  0  0
2  b  a  0  2
3  b  a  2  8
4  c  b  0  4
5  c  b  0  8
6  d  b  1  3
7  d  b  1  2
8  e  c  4  4
9  e  c  3  7
10 f  c  2  7
11 f  c  0  0

In [211]: df.query('a in b')
Out[211]:
    a  b  c  d
0  a  a  1  7
1  a  a  0  0
2  b  a  0  2
3  b  a  2  8
4  c  b  0  4
5  c  b  0  8

# How you’d do it in pure Python
In [212]: df[df.a.isin(df.b)]
Out[212]:
    a  b  c  d
0  a  a  1  7
1  a  a  0  0
2  b  a  0  2
3  b  a  2  8
4  c  b  0  4
In [213]: df.query('a not in b')
Out[213]:
   a  b  c  d
 0  c  b 0  8

# pure Python
In [214]: df[~df.a.isin(df.b)]
Out[214]:
   a  b  c  d
 0  c  b 0  8

You can combine this with other expressions for very succinct queries:

# rows where cols a and b have overlapping values and col c's values are less than col d's
In [215]: df.query('a in b and c < d')
Out[215]:
   a  b  c  d
 0  a  a 1  7
 2  b  a 0  2
 3  b  a 2  8
 4  c  b 0  4
 5  c  b 0  8

# pure Python
In [216]: df[(df.b.isin(df.a) & (df.c < df.d))]
Out[216]:
   a  b  c  d
 0  a  a 1  7
 2  b  a 0  2
 3  b  a 2  8
 4  c  b 0  4
 5  c  b 0  8
 6  d  b 1  3
 7  d  b 1  2
 9  e  c 3  7
10  f  c 2  7
11  f  c 0  0

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

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.
13.14.5 Special use of the == operator with list objects

Comparing a list of values to a column using ==/!= works similarly to in/not in

```
In [217]: df.query('b == ["a", "b", "c"]')
Out[217]:
  a  b  c  d
0  a  a  1  7
1  a  a  0  0
2  b  a  0  2
3  b  a  2  8
4  c  b  0  4
5  c  b  0  8
6  d  b  1  3
7  d  b  1  2
8  e  c  4  4
9  e  c  3  7
10  f  c  2  7
11  f  c  0  0
```

```
# pure Python
In [218]: df[df.b.isin(["a", "b", "c")]]
Out[218]:
  a  b  c  d
0  a  a  1  7
1  a  a  0  0
2  b  a  0  2
3  b  a  2  8
4  c  b  0  4
5  c  b  0  8
6  d  b  1  3
7  d  b  1  2
8  e  c  4  4
9  e  c  3  7
10  f  c  2  7
11  f  c  0  0
```

```
In [219]: df.query('c == [1, 2]')
Out[219]:
  a  b  c  d
4  c  b  0  8
5  c  b  0  8
8  e  c  4  4
9  e  c  3  7
10  f  c  2  7
11  f  c  0  0
```

```
In [220]: df.query('c != [1, 2]')
Out[220]:
  a  b  c  d
1  a  a  0  0
2  b  a  0  2
4  c  b  0  4
5  c  b  0  8
8  e  c  4  4
9  e  c  3  7
11  f  c  0  0
```

```
# using in/not in
In [221]: df.query('1 in c')
```
13.14.6 Boolean Operators

You can negate boolean expressions with the word `not` or the `~` operator.

In [224]: df = DataFrame(rand(n, 3), columns=list('abc'))

In [225]: df['bools'] = rand(len(df)) > 0.5

In [226]: df.query('~bools')

Out[226]:
   a    b    c  bools
0 0.3958 0.0355 0.1717 False
2 0.5823 0.8988 0.4350 False
3 0.0783 0.2247 0.6976 False
5 0.8771 0.2211 0.2874 False
6 0.9933 0.8616 0.1088 False

In [227]: df.query('not bools')

Out[227]:
   a    b    c  bools
0 0.3958 0.0355 0.1717 False
2 0.5823 0.8988 0.4350 False
3 0.0783 0.2247 0.6976 False
5 0.8771 0.2211 0.2874 False
6 0.9933 0.8616 0.1088 False

In [228]: df.query('not bools') == df[~df.bools]

Out[228]:
Of course, expressions can be arbitrarily complex too

# short query syntax
In [229]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [230]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]

In [231]: shorter
Out[231]:
   a    b    c  bools
0  0.078368 0.224708 0.697626  False

In [232]: longer
Out[232]:
   a    b    c  bools
0  0.078368 0.224708 0.697626  False

In [233]: shorter == longer
Out[233]:
   a    b    c  bools
0  True  True  True  True

13.14.7 Performance of query()

DataFrame.query() using numexpr is slightly faster than Python for large frames

Note: You will only see the performance benefits of using the numexpr engine with DataFrame.query() if your frame has more than approximately 200,000 rows
13.15 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 `take_last` parameter that indicates the last observed row should be taken instead.

```python
In [234]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                   'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                   'c' : np.random.randn(7)})

In [235]: df2.duplicated(['a','b'])
Out[235]:
0    False
1    False
2    False
3    False
4     True
5     True
6    False
dtype: bool

In [236]: df2.drop_duplicates(['a','b'])
Out[236]:
   a  b       c
0  one  x  0.932713
An alternative way to drop duplicates on the index is \texttt{.groupby(level=0).first()} or \texttt{.last()}.

\begin{Verbatim}
In [238]: df3 = df2.set_index('b')

In [239]: df3

Out[239]:

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>one</td>
</tr>
<tr>
<td>y</td>
<td>one</td>
</tr>
<tr>
<td>y</td>
<td>two</td>
</tr>
<tr>
<td>x</td>
<td>three</td>
</tr>
<tr>
<td>y</td>
<td>two</td>
</tr>
<tr>
<td>x</td>
<td>one</td>
</tr>
<tr>
<td>x</td>
<td>six</td>
</tr>
</tbody>
</table>
\end{Verbatim}

\begin{Verbatim}
In [240]: df3.groupby(level=0).first()

Out[240]:

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>one</td>
</tr>
<tr>
<td>y</td>
<td>one</td>
</tr>
</tbody>
</table>
\end{Verbatim}

\begin{Verbatim}
# a bit more verbose
In [241]: df3.reset_index().drop_duplicates(subset='b', take_last=False).set_index('b')

Out[241]:

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>one</td>
</tr>
<tr>
<td>y</td>
<td>one</td>
</tr>
</tbody>
</table>
\end{Verbatim}

### 13.16 Dictionary-like \texttt{get()} method

Each of Series, DataFrame, and Panel have a \texttt{get()} method which can return a default value.

\begin{Verbatim}
In [242]: s = Series([1,2,3], index=['a','b','c'])

In [243]: s.get('a')  # equivalent to s['a']

Out[243]: 1

In [244]: s.get('x', default=-1)

Out[244]: -1
\end{Verbatim}
13.17 The select() Method

Another way to extract slices from an object is with the select method of Series, DataFrame, and Panel. This method should be used only when there is no more direct way. select takes a function which operates on labels along axis and returns a boolean. For instance:

```
In [245]: df.select(lambda x: x == 'A', axis=1)
Out[245]:
          A
2000-01-01  0.454389
2000-01-02  0.036249
2000-01-03  0.378125
2000-01-04  0.075871
2000-01-05 -0.677097
2000-01-06  1.482845
2000-01-07  0.272681
2000-01-08 -0.459059
```

13.18 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,

```
In [246]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])
In [247]: dflookup.lookup(list(range(0,10,2)), ['B','C','A','B','D'])
Out[247]: array([ 0.012 , 0.3551, 0.3261, 0.4702, 0.3107])
```

13.19 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:

```
In [248]: index = Index(['e', 'd', 'a', 'b'])
In [249]: index
Out[249]: Index(['e', 'd', 'a', 'b'], dtype='object')
In [250]: 'd' in index
Out[250]: True
```

You can also pass a name to be stored in the index:

```
In [251]: index = Index(['e', 'd', 'a', 'b'], name='something')
In [252]: index.name
Out[252]: 'something'
```

The name, if set, will be shown in the console display:
In [253]: index = Index(list(range(5)), name='rows')

In [254]: columns = Index(['A', 'B', 'C'], name='cols')

In [255]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [256]: df
Out[256]:
   rows
0 0.603791 0.388713 0.544331
1 -0.152978 1.929541 0.202138
2  0.024972 0.117533 -0.184740
3  1.054144 -0.736061 -0.785352
4 -1.362549 -0.063514  0.487562

In [257]: df['A']
Out[257]:
    rows
0  0.603791
1 -0.152978
2  0.024972
3  1.054144
4 -1.362549

Name: A, dtype: float64

13.19.1 Setting metadata

New in version 0.13.0. 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 [258]: ind = Index([1, 2, 3])

In [259]: ind.rename("apple")
Out[259]: Int64Index([1, 2, 3], dtype='int64', name=u'apple')

In [260]: ind
Out[260]: Int64Index([1, 2, 3], dtype='int64')

In [261]: ind.set_names(['apple'], inplace=True)

In [262]: ind.name = "bob"

In [263]: ind
Out[263]: Int64Index([1, 2, 3], dtype='int64', name=u'bob')

New in version 0.15.0.

set_names, set_levels, and set_labels also take an optional level` argument

In [264]: index = MultiIndex.from_product([range(3), ['one', 'two']], names=['first', 'second'])

In [265]: index
Out[265]:
MultiIndex(levels=[[0, 1, 2], [u'one', u'two']]
   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
   names=[u'first', u'second'])

In [266]: index.levels[1]
Out[266]: Index([u'one', u'two'], dtype='object', name=u'second')

In [267]: index.set_levels(['a', 'b'], level=1)
Out[267]:
MultiIndex(levels=[[0, 1, 2], [u'a', u'b']]
   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
   names=[u'first', u'second'])

13.19.2 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 [268]: a = Index(['c', 'b', 'a'])
In [269]: b = Index(['c', 'e', 'd'])
In [270]: a | b
Out[270]: Index([u'a', u'b', u'c', u'd', u'e'], dtype='object')

In [271]: a & b
Out[271]: Index([u'c'], dtype='object')

In [272]: a.difference(b)
Out[272]: Index([u'a', u'b'], dtype='object')

Also available is the sym_diff (^) 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 [273]: idx1 = Index([1, 2, 3, 4])
In [274]: idx2 = Index([2, 3, 4, 5])
In [275]: idx1.sym_diff(idx2)
Out[275]: Int64Index([1, 5], dtype='int64')
In [276]: idx1 ^ idx2
Out[276]: Int64Index([1, 5], dtype='int64')

13.20 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.
13.20.1 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:

```
In [277]: data
Out[277]:
    a   b   c   d
0  bar  one  z   1
1  bar  two  y   2
2  foo  one  x   3
3  foo  two  w   4
```

```
In [278]: indexed1 = data.set_index('c')
```

```
In [279]: indexed1
Out[279]:
     a   b   d
    c
   z  bar  one  1
   y  bar  two  2
   x  foo  one  3
   w  foo  two  4
```

```
In [280]: indexed2 = data.set_index(['a', 'b'])
```

```
In [281]: indexed2
Out[281]:
    c   d
   a   b
  bar  one  z  1
     two  y  2
  foo  one  x  3
     two  w  4
```

The append keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [282]: frame = data.set_index('c', drop=False)
```

```
In [283]: frame = frame.set_index(['a', 'b'], append=True)
```

```
In [284]: frame
Out[284]:
    c   d
   a   b
  bar  one  z  1
     two  y  2
  foo  one  x  3
     two  w  4
```

Other options in set_index allow you not drop the index columns or to add the index in-place (without creating a new object):

```
In [285]: data.set_index('c', drop=False)
Out[285]:
     a   b   c   d
    c
   z  bar  one  1
   y  bar  two  2
```

420 Chapter 13. Indexing and Selecting Data
x  foo  one  x  3  
w  foo  two  w  4

In [286]: data.set_index(['a', 'b'], inplace=True)

In [287]: data
Out[287]:
   c  d
   a  b
   bar  one  z  1
       two  y  2
   foo  one  x  3
       two  w  4

13.20.2 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.

In [288]: data
Out[288]:
   c  d
   a  b
   bar  one  z  1
       two  y  2
   foo  one  x  3
       two  w  4

In [289]: data.reset_index()
Out[289]:
    a  b  c  d
   0  bar  one  z  1
   1  bar  two  y  2
   2  foo  one  x  3
   3  foo  two  w  4

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the names attribute.

You can use the level keyword to remove only a portion of the index:

In [290]: frame
Out[290]:
   c  d
   c  a  b
   z  bar  one  z  1
   y  bar  two  y  2
   x  foo  one  x  3
   w  foo  two  w  4

In [291]: frame.reset_index(level=1)
Out[291]:
   a  c  d
   c  b
   z  one  bar  z  1
   y  two  bar  y  2
   x  one  foo  x  3
   w  two  foo  w  4
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.

### 13.20.3 Adding an ad hoc index

If you create an index yourself, you can just assign it to the index field:

```python
data.index = index
```

### 13.21 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.

```python
In [292]: dfmi = DataFrame([list('abcd'),
                      list('efgh'),
                      list('ijkl'),
                      list('mnop')],
                      columns=MultiIndex.from_product([['one','two'],
                                                      ['first','second']]))
```

```python
In [293]: dfmi
Out[293]:
   one  two
first second first second
0     a     b     c     d
1     e     f     g     h
2     i     j     k     l
3     m     n     o     p
```

Compare these two access methods:

```python
In [294]: dfmi['one']['second']
Out[294]:
0     b
1     f
2     j
3     n
Name: second, dtype: object
```

```python
In [295]: dfmi.loc[:,('one','second')]
Out[295]:
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 data frame 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.

13.21.1 Why does the assignment when using chained indexing fail!

So, why does this show the SettingWithCopy warning / and possibly not work when you do chained indexing and assignment:

    dfmi['one']['second'] = value

Since the chained indexing is 2 calls, it is possible that either call may return a copy of the data because of the way it is sliced. Thus when setting, you are actually setting a copy, and not the original frame data. It is impossible for pandas to figure this out because their are 2 separate python operations that are not connected.

The SettingWithCopy warning is a ‘heuristic’ to detect this (meaning it tends to catch most cases but is simply a lightweight check). Figuring this out for real is way complicated.

The .loc operation is a single python operation, and thus can select a slice (which still may be a copy), but allows pandas to assign that slice back into the frame after it is modified, thus setting the values as you would think.

The reason for having the SettingWithCopy warning is this. Sometimes when you slice an array you will simply get a view back, which means you can set it no problem. However, even a single dtyped array can generate a copy if it is sliced in a particular way. A multi-dtyped DataFrame (meaning it has say float and object data), will almost always yield a copy. Whether a view is created is dependent on the memory layout of the array.

13.21.2 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 exception will be raised (this raise/warn behavior is new starting in 0.13.0)

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.

In [296]: dfb = 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 [297]: dfb['c'][dfb.a.str.startswith('o')] = 42

This however is operating on a copy and will not work.

In

>>> pd.set_option('mode.chained_assignment', 'warn')
>>> dfb[dfb.a.str.startswith('o')]['c'] = 42
Traceback (most recent call last)
...
SettingWithCopyWarning:

13.21. Returning a view versus a copy

423
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/.ix`

This is the correct access method

```
In [298]: dfc = DataFrame({"A": ["aaa", "bbb", "ccc"], "B": [1, 2, 3]})

In [299]: dfc.loc[0, 'A'] = 11

In [300]: dfc
Out[300]:
   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

```
In [301]: dfc = dfc.copy()

In [302]: dfc['A'][0] = 111

In [303]: dfc
Out[303]:
   A  B
0  111  1
1  bbb  2
2  ccc  3
```

This will **not** work at all, and so should be avoided

```
>>> 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

**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 should be a transparent change with only very limited API implications (See the Internal Refactoring)

See the cookbook for some advanced strategies

### 14.1 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 the 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

#### 14.1.1 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 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', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
```

```
In [2]: tuples = list(zip(*arrays))
```
In [3]: tuples
Out[3]:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [5]: index
Out[5]:
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 [6]: s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]:
first  second
bar   one   0.469112
      two  -0.282863
baz   one  -1.509059
      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 [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])
Out[9]:
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 [10]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
         np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]

In [11]: s = pd.Series(np.random.randn(8), index=arrays)

In [12]: s
Out[12]:
bar   one  -0.861849
      two  -2.104569
baz   one  -0.494929
two 1.071804
foo one 0.721555
two -0.706771
qux one -1.039575
two 0.271860
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 -1.776904 -0.968914
foo one  1.075770 -0.109050  1.643563 -1.469388
two  0.357021 -0.674600 -1.776904 -0.968914
qux one -1.294524  0.413738  0.276662 -0.472035
two -0.013960 -0.362543 -0.006154 -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  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.

It’s worth keeping in mind that there’s nothing preventing you from using tuples as atomic labels on an axis:

14.1. Hierarchical indexing (MultiIndex) 427
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 by be controlled using the multi_sparse option in pandas.set_printoptions:

In [20]: pd.set_option('display.multi_sparse', False)

In [21]: df
Out[21]:
<table>
<thead>
<tr>
<th>first</th>
<th>bar</th>
<th>bar</th>
<th>baz</th>
<th>baz</th>
<th>foo</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>second</td>
<td>one</td>
<td>two</td>
<td>one</td>
<td>two</td>
<td>one</td>
<td>two</td>
<td>one</td>
</tr>
<tr>
<td>A</td>
<td>0.895717</td>
<td>0.805244</td>
<td>-1.206412</td>
<td>2.565646</td>
<td>1.431256</td>
<td>1.340309</td>
<td>-1.170299</td>
</tr>
<tr>
<td>B</td>
<td>0.410835</td>
<td>0.813850</td>
<td>0.132003</td>
<td>-0.827317</td>
<td>-0.076467</td>
<td>-1.187678</td>
<td>1.130127</td>
</tr>
<tr>
<td>C</td>
<td>-1.413681</td>
<td>1.607920</td>
<td>1.024180</td>
<td>0.569605</td>
<td>0.875906</td>
<td>-2.211372</td>
<td>0.974466</td>
</tr>
<tr>
<td></td>
<td>first</td>
<td></td>
<td>qux</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>-0.226169</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-1.436737</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-2.006747</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In [22]: pd.set_option('display.multi_sparse', True)

14.1.2 Reconstructing the level labels

The method get_level_values will return a vector of the labels for each location at a particular level:

In [23]: index.get_level_values(0)
Out[23]: Index([u'bar', u'bar', u'baz', u'baz', u'foo', u'foo', u'qux'], dtype='object', name='first')

In [24]: index.get_level_values('second')
Out[24]: Index([u'one', u'two', u'one', u'two', u'one', u'two', u'one'], dtype='object', name='second')

14.1.3 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:
In [25]: df['bar']
Out[25]:
second  one  two
A    0.895717  0.805244
B    0.410835  0.813850
C   -1.413681  1.607920

In [26]: df['bar', 'one']
Out[26]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64

In [27]: df['bar']['one']
Out[27]:
A    0.895717
B    0.410835
C   -1.413681
Name: one, dtype: float64

In [28]: s['qux']
Out[28]:
one -1.039575
two  0.271860
dtype: float64

See `Cross-section with hierarchical index` for how to select on a deeper level.

### 14.1.4 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:

In [29]: s + s[:-2]
Out[29]:
bar  one  -1.723698
two  -4.209138
baz  one  -0.989859
two   2.143608
foo  one   1.443110
two  -1.413542
qux  one   NaN
     two   NaN
dtype: float64

In [30]: s + s[::2]
Out[30]:
bar  one  -1.723698
two  -2.079150
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:

```
In [31]: s.reindex(index[:3])
Out[31]:
     first  second
bar   one  -0.861849
       two  -2.104569
baz   one  -0.494929
dtype: float64
```

```
In [32]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
Out[32]:
     foo  two  -0.706771
    bar  one  -0.861849
   qux  one  -1.039575
  baz  one  -0.494929
dtype: float64
```

### 14.2 Advanced indexing with hierarchical index

Syntactically integrating MultiIndex in advanced indexing with `.loc/.ix` is a bit challenging, but we’ve made every effort to do so. For example the following works as you would expect:

```
In [33]: df = df.T

In [34]: df
Out[34]:
     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 [35]: df.loc['bar']
Out[35]:
     A   B   C
second
one    0.895717 0.410835 -1.413681
two    0.805244 0.813850 1.607920

In [36]: df.loc['bar', 'two']
Out[36]:
   A   B   C
Name: (bar, two), dtype: float64
```

“Partial” slicing also works quite nicely.

```
In [37]: df.loc['baz':'foo']
Out[37]:
     A   B   C
```

430 Chapter 14. MultiIndex / Advanced Indexing
You can slice with a 'range' of values, by providing a slice of tuples.

```python
In [38]: df.loc[('baz', 'two'):('qux', 'one')]
Out[38]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>second</td>
<td></td>
<td></td>
</tr>
<tr>
<td>baz</td>
<td>two</td>
<td>2.565646</td>
<td>-0.827317</td>
</tr>
<tr>
<td>foo</td>
<td>one</td>
<td>1.431256</td>
<td>-0.076467</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>1.340309</td>
<td>-1.187678</td>
</tr>
<tr>
<td>qux</td>
<td>one</td>
<td>-1.170299</td>
<td>1.130127</td>
</tr>
</tbody>
</table>
```

```python
In [39]: df.loc[('baz', 'two'):('foo')]
Out[39]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>second</td>
<td></td>
<td></td>
</tr>
<tr>
<td>baz</td>
<td>two</td>
<td>2.565646</td>
<td>-0.827317</td>
</tr>
<tr>
<td>foo</td>
<td>one</td>
<td>1.431256</td>
<td>-0.076467</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>1.340309</td>
<td>-1.187678</td>
</tr>
</tbody>
</table>
```

Passing a list of labels or tuples works similar to reindexing:

```python
In [40]: df.ix[[(('bar', 'two'), ('qux', 'one'))]
Out[40]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>second</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>two</td>
<td>0.805244</td>
<td>0.813850</td>
</tr>
<tr>
<td>qux</td>
<td>one</td>
<td>-1.170299</td>
<td>1.130127</td>
</tr>
</tbody>
</table>
```

### 14.2.1 Using slicers

New in version 0.14.0.

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.
Warning: You should specify all axes in the `.loc` specifier, meaning the indexer for the `index` and for the `columns`. Their are some ambiguous cases where the passed indexer could be mis-interpreted as indexing both axes, rather than into, say, the MultiIndex for the rows.

You should do this:

```python
df.loc[(slice('A1', 'A3'), ......), :]
```

rather than this:

```python
df.loc[(slice('A1', 'A3'), ......)]
```

Warning: You will need to make sure that the selection axes are fully lexsorted!

```
In [41]: def mklbl(prefix,n):
    ...:     return ["%s%s" % (prefix,i) for i in range(n)]
    ...:

In [42]: miindex = pd.MultiIndex.from_product([mklbl('A',4),
    ...:         mklbl('B',2),
    ...:         mklbl('C',4),
    ...:         mklbl('D',2)])

In [43]: micolumns = pd.MultiIndex.from_tuples([('a','foo'),('a','bar'),
    ...:                                       ('b','foo'),('b','bah')],
    ...:                                      names=['lvl0', 'lvl1'])

In [44]: dfmi = pd.DataFrame(np.arange(len(miindex)*len(micolumns)).reshape((len(miindex),len(micolumns))),
    ...:                     index=miindex,
    ...:                     columns=micolumns).sortlevel().sortlevel(axis=1)

In [45]: dfmi
Out[45]:
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 [46]: dfmi.loc[(slice('A1','A3'),slice(None), ['C1','C3'])),:]
Out[46]:
lvl0    a    b
lvl1   bar  foo  bah  foo
A1  B0  C1  D0  73  72  75  74
  D1  77  76  79  78
  C3  D0  89  88  91  90
  D1  93  92  95  94
B1  C1  D0  105 104 107 106
  D1  109 108 111 110
  C3  D0  121 120 123 122
  D1  125 124 127 126
  ... ... ... ... ...
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
  D1  237 236 239 238
  C3  D0  249 248 251 250
  D1  253 252 255 254
[24 rows x 4 columns]

You can use a `pd.IndexSlice` to have a more natural syntax using : rather than using `slice(None)`

In [47]: idx = pd.IndexSlice

In [48]: dfmi.loc[idx[:,:,['C1','C3']]],idx[:,['foo']]
Out[48]:
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  217 218
  D1  221 222
B1  C1  D0  233 234
  D1  237 238
  C3  D0  249 250
  D1  253 254
[32 rows x 2 columns]

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

In [49]: dfmi.loc['A1',{slice(None),'foo'}]
Out[49]:
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

14.2. Advanced indexing with hierarchical index 433
Using a boolean indexer you can provide selection related to the \textit{values}.

\textbf{In [51]}: \texttt{mask = dfmi[\{'a','foo'\}] >200}

\textbf{In [52]}: \texttt{dfmi.loc[idx[mask,:,\['C1','C3'\]],idx[:,\['foo'\]]}}

\textbf{Out[52]}:

\begin{verbatim}
 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]
\end{verbatim}

You can also specify the \texttt{axis} argument to \texttt{.loc} to interpret the passed slicers on a single axis.

\textbf{In [53]}: \texttt{dfmi.loc(axis=0)[ :,\['C1','C3'\]]}

\textbf{Out[53]}:

\begin{verbatim}
 lvl0  a  b
 lvl1  bar  foo  bah  foo
 A0  B0  C1  D0  9  8  11  10
     D1  13  12  15  14
\end{verbatim}
Furthermore you can set the values using these methods:

```python
In [54]: df2 = dfmi.copy()

In [55]: df2.loc(axis=0)[::,::,["C1","C3"]]= -10

In [56]: df2
Out[56]:
   a   b
0  3  2
1  7  6
2 -10 -10
3 -10 -10
   ... ... ...
229 228 231 230
230 230 230 230

[32 rows x 4 columns]
```

You can use a right-hand-side of an alignable object as well.

```python
In [57]: df2 = dfmi.copy()

In [58]: df2.loc[idx[::,::,["C1","C3"]],::] = df2*1000

In [59]: df2
Out[59]:
   a   b
0  3  2
1  7  6
2 -10 -10
3 -10 -10
   ... ... ...
229 228 231 230
230 230 230 230

[64 rows x 4 columns]
```
14.2.2 Cross-section

The \texttt{xs} method of \texttt{DataFrame} additionally takes a level argument to make selecting data at a particular level of a MultiIndex easier.

In [60]: df
Out[60]:
   A     B     C
first second
bar  one  0.895717  0.410835 -1.413681
two  -1.206412  0.132003  1.024180
baz  one  2.565646 -0.827317  0.569605
two  1.431256 -0.076467  0.875906
foo  one  1.340309 -1.187678 -2.211372
two  1.170299  1.130127  0.974466
qux  one -0.226169 -1.436737 -2.006747

In [61]: df.xs('one', level='second')
Out[61]:
   A     B     C
first
bar  one  0.895717  0.410835 -1.413681
baz  one -1.206412  0.132003  1.024180
foo  one  2.565646 -0.827317  0.569605
qux  one  1.431256 -0.076467  0.875906

# using the slicers (new in 0.14.0)  
In [62]: df.loc[(slice(None),'one'),:]
Out[62]:
   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 \texttt{xs()}, by providing the axis argument

In [63]: df = df.T

In [64]: df.xs('one', level='second', axis=1)
Out[64]:
   first  bar  baz  foo  qux
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
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

# using the slicers (new in 0.14.0)
In [65]: df.loc[:,(slice(None),'one')]
Out[65]:
first   bar   baz   foo   qux
second  one   one   one   one
A   0.895717  -1.206412  1.431256  -1.170299
B   0.410835   0.132003  -0.076467   1.130127
C  -1.413681   1.024180   0.875906   0.974466

xs() also allows selection with multiple keys

In [66]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out[66]:
first   bar
second  one
A   0.895717
B   0.410835
C  -1.413681

# using the slicers (new in 0.14.0)
In [67]: df.loc[:,('bar','one')]
Out[67]:
A   0.895717
B   0.410835
C  -1.413681
Name: (bar, one), dtype: float64

New in version 0.13.0.

You can pass drop_level=False to xs() to retain the level that was selected

In [68]: df.xs('one', level='second', axis=1, drop_level=False)
Out[68]:
first   bar   baz   foo   qux
second  one   one   one   one
A   0.895717  -1.206412  1.431256  -1.170299
B   0.410835   0.132003  -0.076467   1.130127
C  -1.413681   1.024180   0.875906   0.974466

versus the result with drop_level=True (the default value)

In [69]: df.xs('one', level='second', axis=1, drop_level=True)
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

14.2.3 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 [70]: midx = pd.MultiIndex(levels=[[‘zero’, ‘one’], [‘x’, ‘y’]],
                    labels=[[1, 1, 0, 0], [1, 0, 1, 0]])

In [71]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)

In [72]: df
Out[72]:
          0  1
one  y  1.519970 -0.493662
    x  0.600178  0.274230
zero y  0.132885 -0.023688
    x  2.410179  1.450520

In [73]: df2 = df.mean(level=0)

In [74]: df2
Out[74]:
          0  1
zero  1.271532  0.713416
one  1.060074 -0.109716

In [75]: df2.reindex(df.index, level=0)
Out[75]:
          0  1
one  y  1.060074 -0.109716
    x  1.060074 -0.109716
zero y  1.271532  0.713416
    x  1.271532  0.713416

# aligning
In [76]: df_aligned, df2_aligned = df.align(df2, level=0)

In [77]: df_aligned
Out[77]:
          0  1
one  y  1.519970 -0.493662
    x  0.600178  0.274230
zero y  0.132885 -0.023688
    x  2.410179  1.450520

In [78]: df2_aligned
Out[78]:
          0  1
one  y  1.060074 -0.109716
    x  1.060074 -0.109716
zero y  1.271532  0.713416
    x  1.271532  0.713416

14.2.4 Swapping levels with swaplevel()

The swaplevel function can switch the order of two levels:

In [79]: df[:5]
Out[79]:
      0  1
one  y  1.519970 -0.493662
14.2.5 Reordering levels with reorder_levels()

The `reorder_levels` function generalizes the `swaplevel` function, allowing you to permute the hierarchical index levels in one step:

```python
In [81]: df[:5].reorder_levels([1, 0], axis=0)
Out[81]:
   0  1
y one 1.51997  -0.49366
x one 0.60018  0.27423
y zero 0.13289  -0.02369
x zero 2.41018  1.45052
```

14.3 The need for sortedness with MultiIndex

Caveat emptor: the present implementation of `MultiIndex` requires that the labels be sorted for some of the slicing / indexing routines to work correctly. You can think about breaking the axis into unique groups, where at the hierarchical level of interest, each distinct group shares a label, but no two have the same label. However, the `MultiIndex` does not enforce this: you are responsible for ensuring that things are properly sorted. There is an important new method `sortlevel` to sort an axis within a `MultiIndex` so that its labels are grouped and sorted by the original ordering of the associated factor at that level. Note that this does not necessarily mean the labels will be sorted lexicographically!

```python
import random; random.shuffle(tuples)
In [82]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))
In [83]: s.sortlevel(0)
Out[83]:
foo two   0.206053
bar one  -0.251905
baz two  -2.213588
foo one   1.063327
baz one   1.266143
qux two   0.299368
bar two  -0.863838
qux one   0.408204
dtype: float64
```
two  -0.863838
baz  1.266143
two -2.213588
foo  1.063327
two  0.206053
qux  0.408204
two  0.299368
dtype: float64

In [86]: s.sortlevel(1)
Out[86]:
bar one  -0.251905
baz one  1.266143
foo one  1.063327
qux one  0.408204
bar two  -0.863838
baz two -2.213588
foo two  0.206053
qux two  0.299368
dtype: float64

Note, you may also pass a level name to sortlevel if the MultiIndex levels are named.

In [87]: s.index.set_names(['L1', 'L2'], inplace=True)

In [88]: s.sortlevel(level='L1')
Out[88]:
L1  L2
bar one  -0.251905
two  -0.863838
baz one  1.266143
two  -2.213588
foo one  1.063327
two  0.206053
qux one  0.408204
two  0.299368
dtype: float64

In [89]: s.sortlevel(level='L2')
Out[89]:
L1  L2
bar one  -0.251905
baz one  1.266143
foo one  1.063327
qux one  0.408204
bar two  -0.863838
baz two -2.213588
foo two  0.206053
qux two  0.299368
dtype: float64

Some indexing will work even if the data are not sorted, but will be rather inefficient and will also return a copy of the data rather than a view:

In [90]: s['qux']
Out[90]:
L2
two  0.299368
one  0.408204
On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

```python
In [92]: df.T.sortlevel(1, axis=1)
Out[92]:
   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
```

The MultiIndex object has code to **explicitly check the sort depth**. Thus, if you try to index at a depth at which the index is not sorted, it will raise an exception. Here is a concrete example to illustrate this:

```python
In [93]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]
In [94]: idx = pd.MultiIndex.from_tuples(tuples)
In [95]: idx.lexsort_depth
Out[95]: 2
In [96]: reordered = idx[[1, 0, 3, 2]]
In [97]: reordered.lexsort_depth
Out[97]: 1
In [98]: s = pd.Series(np.random.randn(4), index=reordered)
In [99]: s.ix['a':'a']
Out[99]:
   a  b
0 -1.048089
1  0.025747
dtype: float64
```

However:

```python
>>> s.ix[('a', 'b'):('b', 'a')]
Traceback (most recent call last)
...
KeyError: Key length (3) was greater than MultiIndex lexsort depth (2)
```

### 14.4 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.

```python
In [100]: index = pd.Index(np.random.randint(0, 1000, 10))
In [101]: index
```
For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

It is important to note that the take method on pandas objects are not intended to work on boolean indices and may return unexpected results.
In [115]: ser.take([False, False, True, True])
Out[115]:
     0  0.233141  0.233141
     1 -0.223540 -0.223540
dtype: float64

In [116]: ser.ix[[0, 1]]
Out[116]:
     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.

### 14.5 CategoricalIndex

New in version 0.16.1.

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 [117]: df = pd.DataFrame({'A': np.arange(6),
      'B': list('aabbc')} )

In [119]: df['B'] = df['B'].astype('category', categories=list('cab'))

In [120]: df.dtypes
Out[120]:
A     int32
B   category
dtype: object

In [121]: df.B.cat.categories
Out[121]: Index([u'c', u'a', u'b'], dtype='object')

Setting the index, will create a `CategoricalIndex`

In [122]: df2 = df.set_index('B')

In [123]: df2.index
Out[123]: CategoricalIndex([u'a', u'a', u'b', u'b', u'c', u'a'], categories=[u'c', u'a', u'b'], ordered=False, 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 [124]: df2.loc['a']
Out[124]:
   A  
  a 0
  a 1
  a 5
```

These PRESERVE the CategoricalIndex

```
In [125]: df2.loc['a'].index
Out[125]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Sorting will order by the order of the categories

```
In [126]: df2.sort_index()
Out[126]:
   A  
   B  
  c 4
  a 0
  a 1
  a 5
  b 2
  b 3
```

Groupby operations on the index will preserve the index nature as well

```
In [127]: df2.groupby(level=0).sum()
Out[127]:
   A  
   B  
  c 4
  a 6
  b 5
```

```
In [128]: df2.groupby(level=0).sum().index
Out[128]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Reindexing operations, will return a resulting index based on the type of the passed indexer, meaning that passing a list will return a plain-old-Index; indexing with a Categorical will return a CategoricalIndex, indexed according to the categories of the PASSED Categorical dtype. This allows one to arbitrarily index these even with values NOT in the categories, similarly to how you can reindex ANY pandas index.

```
In [129]: df2.reindex(['a','e'])
Out[129]:
   A  
   B  
  a 0
  a 1
  a 5
  e NaN

In [130]: df2.reindex(['a','e']).index
Out[130]: Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [131]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde')))
```
Warning: Reshaping and Comparison operations on a CategoricalIndex must have the same categories or a TypeError will be raised.

14.6 Float64Index

Note: As of 0.14.0, Float64Index is backed by a native float64 dtype array. Prior to 0.14.0, Float64Index was backed by an object dtype array. Using a float64 dtype in the backend speeds up arithmetic operations by about 30x and boolean indexing operations on the Float64Index itself are about 2x as fast.

New in version 0.13.0.

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 [], .ix, .loc will always be label based. An integer will match an equal float index (e.g. 3 is
equivalent to 3.0)

In [137]: sf[3]
Out[137]: 2

In [138]: sf[3.0]
Out[138]: 2

In [139]: sf.ix[3]
Out[139]: 2

In [140]: sf.ix[3.0]
Out[140]: 2

In [141]: sf.loc[3]
Out[141]: 2

In [142]: sf.loc[3.0]
Out[142]: 2

The only positional indexing is via iloc

In [143]: sf.iloc[3]
Out[143]: 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 [144]: sf[2:4]
Out[144]:
2 1
3 2
dtype: int64

In [145]: sf.ix[2:4]
Out[145]:
2 1
3 2
dtype: int64

In [146]: sf.loc[2:4]
Out[146]:
2 1
3 2
dtype: int64

In [147]: sf.iloc[2:4]
Out[147]:
3.0 2
4.5 3
dtype: int64

In float indexes, slicing using floats is allowed

In [148]: sf[2.1:4.6]
Out[148]:
3.0 2
4.5 3
dtype: int64
In [149]: sf.loc[2.1:4.6]
Out[149]:
3.0  2
4.5  3
dtype: int64

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 (Int64Index)

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

In [3]: pd.Series(range(5))[3.0]
Out[3]: 3

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 [150]: 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 [151]: dfir
Out[151]:
     A       B
0  0.00  -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
1000.4  0.310610 -0.108002
1250.5 -0.974226 -1.147708
1500.6 -2.281374  0.760010
1750.7 -0.742532  1.533318
2000.8  2.495362 -0.432771
2250.9 -0.068954  0.043520

Selection operations then will always work on a value basis, for all selection operators.

In [152]: dfir[0:1000.4]
Out[152]:
     A       B
0  0.00  -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
1000.4  0.310610 -0.108002
In [153]: dfir.loc[0:1001,'A']
Out[153]:

You could then easily pick out the first 1 second (1000 ms) of data then.

In [155]: dfir[0:1000]
Out[155]:
   A      B
0  0.997289 -1.693316
250 -0.179129 -1.598062
500  0.936914  0.912560
750 -1.003401  1.632781
1000 -0.724626  0.178219

Of course if you need integer based selection, then use iloc

In [156]: dfir.iloc[0:5]
Out[156]:
   A      B
0  0.997289 -1.693316
250 -0.179129 -1.598062
500  0.936914  0.912560
750 -1.003401  1.632781
1000 -0.724626  0.178219
15.1 Statistical functions

15.1.1 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    1    2    3
0       NaN  1.6  4.3  0.2
1  -2.0  1.4 -0.2  4.3
2   4.3  4.3 -1.1  4.3
3  4.3  4.3 -1.1  4.3
```

```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  NaN
1  NaN  NaN  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN  NaN
3 -0.2  1.0  1.9  0.6  0.3
4 -0.4  2.0  1.8  3.5 -3.8
5 -0.1  2.0  3.5 -1.5 -1.5
6 -0.1  1.0  1.6  0.5 -0.6
7 -2.5  1.8  1.8 -2.1 -4.1
8 -0.2  0.3  0.3 -1.2 -0.6
9  2.4  2.3  2.1 -1.2 -0.6
```

15.1.2 Covariance

The `Series` object has a method `cov` to compute covariance between series (excluding NA/null values).
In [5]: s1 = pd.Series(np.random.randn(1000))
In [6]: s2 = pd.Series(np.random.randn(1000))
In [7]: s1.cov(s2)
Out[7]: 0.00068010881743109993

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.

In [8]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [9]: frame.cov()
Out[9]:
        a        b        c        d        e
a  1.000882 -0.003177 -0.002698 -0.006889  0.031912
b -0.003177  1.024721  0.000191  0.009212  0.000857
c -0.002698  0.000191  0.950735 -0.031743 -0.005087
d -0.006889  0.009212 -0.031743  1.002983 -0.047952
e  0.031912  0.000857 -0.005087 -0.047952  1.042487

DataFrame.cov also supports an optional min_periods keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

In [10]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])
In [11]: frame.ix[:5, 'a'] = np.nan
In [12]: frame.ix[5:10, 'b'] = np.nan
In [13]: frame.cov()
Out[13]:
        a        b        c
a  1.210090 -0.430629  0.018002
b -0.430629  1.240960  0.347188
c  0.018002  0.347188  1.301149

In [14]: frame.cov(min_periods=12)
Out[14]:
        a        b        c
a  1.210090   NaN   0.018002
b   NaN  1.240960  0.347188
c  0.018002  0.347188  1.301149

### 15.1.3 Correlation

Several methods for computing correlations are provided:
<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pearson (default)</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.ix[: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]: -0.0072898851595406388

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:
   a     b     c     d     e
a  1.000000 -0.049269 -0.042239 -0.028525
b -0.049269  1.000000 -0.011139  0.005654
c -0.042239 -0.011139  1.000000 -0.054269
d -0.028525  0.005654 -0.054269  1.000000
e

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.ix[:5, 'a'] = np.nan

In [22]: frame.ix[5:10, 'b'] = np.nan

In [23]: frame.corr()
Out[23]:
   a     b     c
a  1.000000 -0.076520  0.160092
b -0.076520  1.000000  0.135967
c  0.160092  0.135967  1.000000

In [24]: frame.corr(min_periods=12)
Out[24]:
   a     b     c
a  1.000000  NaN  0.160092
b  NaN  1.000000  0.135967
c  0.160092  0.135967  1.000000
```

A related method corrwith is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

```
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   -0.125501
   two   -0.493244
   three  0.344056
   four   0.004183

...: df2.corrwith(df1, axis=1)
Out[30]:
   a  -0.675817
   b   0.458296
   c   0.190809
   d  -0.186275
   e    NaN

15.1.4 Data ranking

The \texttt{rank} method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

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   4.0

\texttt{rank} is also a DataFrame method and can rank either the rows (axis=0) or the columns (axis=1). \texttt{NaN} values are excluded from the ranking.

In [34]: df = pd.DataFrame(np.random.randn(10, 6))


In [36]: df
Out[36]:
   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
In [37]: df.rank(1)
Out[37]:
    0  1  2  3  4  5
0  4  3  1.5 5  1.5 6
1  2  6  4.5 1  4.5 3
2  1  6  3.5 5  3.5 2
3  4  5  1.5 3  1.5 6
4  5  3  1.5 4  1.5 6
5  1  2  5.0 3  NaN 4
6  4  5  3.0 1  NaN 2
7  2  5  3.0 4  NaN 1
8  2  5  3.0 4  NaN 1
9  2  3  1.0 4  NaN 5

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

## 15.2 Moving (rolling) statistics / moments

For working with time series data, a number of functions are provided for computing common moving or rolling statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis. All of these methods are in the pandas namespace, but otherwise they can be found in pandas.stats.moments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolling_count</td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td>rolling_sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>rolling_mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>rolling_median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>rolling_min</td>
<td>Minimum</td>
</tr>
<tr>
<td>rolling_max</td>
<td>Maximum</td>
</tr>
<tr>
<td>rolling_std</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>rolling_var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>rolling_skew</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>rolling_kurt</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>rolling_quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>rolling_apply</td>
<td>Generic apply</td>
</tr>
<tr>
<td>rolling_cov</td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td>rolling_corr</td>
<td>Correlation (binary)</td>
</tr>
<tr>
<td>rolling_window</td>
<td>Moving window function</td>
</tr>
</tbody>
</table>

Generally these methods all have the same interface. The binary operators (e.g. rolling_corr) take two Series or DataFrames. Otherwise, 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)
- freq: optionally specify a frequency string or DateOffset to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument time_rule was used instead of freq that referred to the legacy time rule constants
- how: optionally specify method for down or re-sampling. Default is min for rolling_min, max for rolling_max, median for rolling_median, and mean for all other rolling functions. See DataFrame.resample()’s how argument for more information.

These functions can be applied to ndarrays or Series objects:

```python
In [38]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [39]: ts = ts.cumsum()

In [40]: ts.plot(style='k--')
Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0xa95eda2c>

In [41]: pd.rolling_mean(ts, 60).plot(style='k')
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0xa95eda2c>
```

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:

```
In [42]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                        columns=["A", "B", "C", "D"])
```

```
In [43]: df = df.cumsum()
```

```
In [44]: pd.rolling_sum(df, 60).plot(subplots=True)
```

The `rolling_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:

```
In [45]: mad = lambda x: np.fabs(x - x.mean()).mean()
```

```
In [46]: pd.rolling_apply(ts, 60, mad).plot(style='k')
```

15.2. Moving (rolling) statistics / moments
The `rolling_window` function performs a generic rolling window computation on the input data. 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 [47]: ser = pd.Series(np.random.randn(10), index=pd.date_range('1/1/2000', periods=10))

In [48]: pd.rolling_window(ser, 5, 'triang')
Out[48]:
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 rolling_mean.

In [49]: pd.rolling_window(ser, 5, 'boxcar')
Out[49]:
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 [50]: pd.rolling_mean(ser, 5)
Out[50]:
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 [51]: pd.rolling_window(ser, 5, 'gaussian', std=0.1)
Out[51]:
2000-01-01    NaN
2000-01-02    NaN
2000-01-03    NaN
2000-01-04    NaN
2000-01-05 -1.309989
2000-01-06 -1.153000
2000-01-07  0.606382
2000-01-08 -0.681101
2000-01-09 -0.289724
2000-01-10     -0.996632
Freq: D, dtype: float64

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. This keyword is available in other rolling functions as well.

```
In [52]: pd.rolling_window(ser, 5, 'boxcar')
Out[52]:
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 [53]: pd.rolling_window(ser, 5, 'boxcar', center=True)
Out[53]:
2000-01-01       NaN
2000-01-02       NaN
2000-01-03    -0.841164
2000-01-04    -0.779948
2000-01-05    -0.565487
2000-01-06    -0.502815
2000-01-07    -0.553755
2000-01-08    -0.472211
2000-01-09       NaN
2000-01-10       NaN
Freq: D, dtype: float64
```

```
In [54]: pd.rolling_mean(ser, 5, center=True)
Out[54]:
2000-01-01       NaN
2000-01-02       NaN
2000-01-03    -0.841164
2000-01-04    -0.779948
2000-01-05    -0.565487
2000-01-06    -0.502815
2000-01-07    -0.553755
2000-01-08    -0.472211
2000-01-09       NaN
2000-01-10       NaN
Freq: D, dtype: float64
```

**Note:** In rolling sum mode (mean=False) there is no normalization done to the weights. 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 rolling mean calculation (mean=True) 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.
15.2.1 Binary rolling moments

rolling_cov and rolling_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 Panel whose items are the dates in question (see the next section).

For example:

In [55]: df2 = df[:20]

In [56]: pd.rolling_corr(df2, df2['B'], window=5)

Out[56]:

<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>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.262853</td>
<td>1</td>
<td>0.334449</td>
<td>0.193380</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.083745</td>
<td>1</td>
<td>-0.521587</td>
<td>-0.556126</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.292940</td>
<td>1</td>
<td>-0.658532</td>
<td>-0.458128</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000-01-14</td>
<td>0.519499</td>
<td>1</td>
<td>-0.687277</td>
<td>0.192822</td>
</tr>
<tr>
<td>2000-01-15</td>
<td>0.048982</td>
<td>1</td>
<td>0.167669</td>
<td>-0.061463</td>
</tr>
<tr>
<td>2000-01-16</td>
<td>0.217190</td>
<td>1</td>
<td>0.167564</td>
<td>-0.326034</td>
</tr>
<tr>
<td>2000-01-17</td>
<td>0.641180</td>
<td>1</td>
<td>-0.164780</td>
<td>-0.111487</td>
</tr>
<tr>
<td>2000-01-18</td>
<td>0.130422</td>
<td>1</td>
<td>0.322833</td>
<td>0.632383</td>
</tr>
<tr>
<td>2000-01-19</td>
<td>0.317278</td>
<td>1</td>
<td>0.384528</td>
<td>0.813656</td>
</tr>
<tr>
<td>2000-01-20</td>
<td>0.293598</td>
<td>1</td>
<td>0.159538</td>
<td>0.742381</td>
</tr>
</tbody>
</table>

[20 rows x 4 columns]

15.2.2 Computing rolling pairwise covariances and correlations

In financial data analysis and other fields it’s common to compute covariance and correlation matrices for a collection of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by passing the pairwise keyword argument, which in the case of DataFrame inputs will yield a Panel whose items 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.

In [57]: covs = pd.rolling_cov(df[['B', 'C', 'D']], df[['A', 'B', 'C']], 50, pairwise=True)

In [58]: covs[df.index[-50]]

Out[58]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2.667506</td>
<td>1.671711</td>
</tr>
</tbody>
</table>

15.2. Moving (rolling) statistics / moments
In [59]: correls = pd.rolling_corr(df, 50)

In [60]: correls[df.index[-50]]
Out[60]:
   A    B    C    D
A  1.000000  0.604221  0.767429 -0.776170
B  0.604221  1.000000  0.461484 -0.381148
C  0.767429  0.461484  1.000000 -0.748863
D -0.776170 -0.381148 -0.748863  1.000000

Note: Prior to version 0.14 this was available through rolling_corr_pairwise which is now simply syntactic sugar for calling rolling_corr(..., pairwise=True) and deprecated. This is likely to be removed in a future release.

You can efficiently retrieve the time series of correlations between two columns using ix indexing:

In [61]: correls.ix[:, 'A', 'C'].plot()
Out[61]: <matplotlib.axes._subplots.AxesSubplot at 0xa63e68ac>
15.3 Expanding window moment functions

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. As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```
In [62]: pd.rolling_mean(df, window=len(df), min_periods=1)[:5]
Out[62]:
   A    B    C    D
2000-01-01 -1.3883  3.3172  0.3445 -0.0369
2000-01-02 -1.1231  3.6223  1.6759  0.5953
2000-01-03 -0.6285  3.6265  2.4552  1.0601
2000-01-04 -0.7687  3.8891  2.4513  1.2818
2000-01-05 -0.8240  4.1080  2.5561  1.1407
```

```
In [63]: pd.expanding_mean(df)[:5]
Out[63]:
   A    B    C    D
2000-01-01 -1.3883  3.3172  0.3445 -0.0369
2000-01-02 -1.1231  3.6223  1.6759  0.5953
2000-01-03 -0.6285  3.6265  2.4552  1.0601
2000-01-04 -0.7687  3.8891  2.4513  1.2818
2000-01-05 -0.8240  4.1080  2.5561  1.1407
```

Like the rolling functions, the following methods are included in the pandas namespace or can be located in pandas.stats.moments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expanding_count</td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td>expanding_sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>expanding_mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>expanding_median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>expanding_min</td>
<td>Minimum</td>
</tr>
<tr>
<td>expanding_max</td>
<td>Maximum</td>
</tr>
<tr>
<td>expanding_std</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>expanding_var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>expanding_skew</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>expanding_kurt</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>expanding_quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>expanding_apply</td>
<td>Generic apply</td>
</tr>
<tr>
<td>expanding_cov</td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td>expanding_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 counterpart. 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.

- freq: optionally specify a frequency string or DateOffset to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument time_rule was used instead of freq that referred to the legacy time rule constants.

**Note:** The output of the rolling_ and expanding_ functions do not return a NaN if there are at least min_periods non-null values in the current window. This differs from cumsum, cumprod, cummax, and cummin, which return NaN in the output wherever a NaN is encountered in the input.
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 expanding\_mean output for the previous time series dataset:

```
In [64]: ts.plot(style='k--')
Out[64]: <matplotlib.axes._subplots.AxesSubplot at 0xa9ba130c>

In [65]: pd.expanding_mean(ts).plot(style='k')
Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0xa9ba130c>
```

### 15.4 Exponentially weighted moment functions

A related set of functions are exponentially weighted versions of several of the above statistics. A number of expanding EW (exponentially weighted) functions are provided:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ewma</td>
<td>EW moving average</td>
</tr>
<tr>
<td>ewmvar</td>
<td>EW moving variance</td>
</tr>
<tr>
<td>ewmstd</td>
<td>EW moving standard deviation</td>
</tr>
<tr>
<td>ewmcov</td>
<td>EW moving covariance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ewmcorr</td>
<td>EW moving correlation</td>
</tr>
</tbody>
</table>

---

Chapter 15. Computational tools
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 at $y_t$ is the result.

The EW functions support two variants of exponential weights: The default, adjust=True, uses the weights $w_i = (1 - \alpha)^i$. When adjust=False is specified, moving averages are calculated as

\begin{align*}
y_0 &= x_0 \\
y_t &= (1 - \alpha)y_{t-1} + \alpha x_t,
\end{align*}

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.

$$y_t = \alpha' y_{t-1} + (1 - \alpha')x_t.$$ 

One must have $0 < \alpha \leq 1$, but rather than pass $\alpha$ directly, it’s easier to think about either the span, center of mass (com) or halflife of an EW moment:

$$\alpha = \begin{cases} 
\frac{2}{s+1}, & s = \text{span} \\
\frac{1}{1+c}, & c = \text{center of mass} \\
1 - \exp \frac{\log 0.5}{h}, & h = \text{halflife}
\end{cases}$$

One must specify precisely one of the three to the EW functions. Span corresponds to what is commonly called a “20-day EW moving average” for example. Center of mass has a more physical interpretation. For example, span = 20 corresponds to com = 9.5. Halflife is the period of time for the exponential weight to reduce to one half.

Here is an example for a univariate time series:

```python
In [66]: plt.close('all')

In [67]: ts.plot(style='k--')
Out[67]: <matplotlib.axes._subplots.AxesSubplot at 0xa9b1732c>

In [68]: pd.ewma(ts, span=20).plot(style='k')
Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0xa9b1732c>
```
All the EW functions have a `min_periods` argument, which has the same meaning it does for all the `expanding_` and `rolling_` functions: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window. (This is a change from versions prior to 0.15.0, in which the `min_periods` argument affected only the `min_periods` consecutive entries starting at the first non-null value.)

All the EW functions also have 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` (which reproduces the behavior in versions prior to 0.15.0), 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 `ewmvar`, `ewmstd`, and `ewmcov` 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 http://en.wikipedia.org/wiki/Weighted_arithmetic_mean#Weighted_sample_variance for further details.
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

16.1 Missing data basics

16.1.1 When / why does data become missing?

Some might quibble over our usage of missing. By “missing” we simply mean null 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  0.469112 -0.282863 -1.509059   bar   True
  c -1.135632  1.212112 -0.173215   bar   False
  e  0.119209 -1.044236 -0.861849   bar   True
  f -2.104569 -0.494929  1.071804   bar   False
  h  0.721555 -0.706771 -1.039575   bar   True
```

```
In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
Out[6]:
   one   two    three  four  five
  a  0.469112 -0.282863 -1.509059   bar   True
  b   nan    nan   nan   nan   nan
  c -1.135632  1.212112 -0.173215   bar   False
  d   nan    nan   nan   nan   nan
  e  0.119209 -1.044236 -0.861849   bar   True
  f -2.104569 -0.494929  1.071804   bar   False
  g   nan    nan   nan   nan   nan
  h  0.721555 -0.706771 -1.039575   bar   True
```
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 “null”.

Prior to version v0.10.0 inf and -inf were also considered to be “null” in computations. This is no longer the case by default; use the mode.use_inf_as_null option to recover it. To make detecting missing values easier (and across different array dtypes), pandas provides the isnull() and notnull() functions, which are also methods on Series objects:

```
In [7]: df2['one']
Out[7]:
   a   0.469112
   b  NaN
   c  -1.135632
   d  NaN
   e  0.119209
   f  -2.104569
   g  NaN
   h  0.721555
Name: one, dtype: float64

In [8]: isnull(df2['one'])
Out[8]:
   a   False
   b   True
   c   False
   d   True
   e   False
   f   False
   g   True
   h   False
Name: one, dtype: bool

In [9]: df2['four'].notnull()
Out[9]:
   a   True
   b   False
   c   True
   d   False
   e   True
   f   True
   g   False
   h   True
Name: four, dtype: bool
```
Summary: NaN and None (in object arrays) are considered missing by the isnull and notnull functions. inf and -inf are no longer considered missing by default.

16.2 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 [10]: df2 = df.copy()

In [11]: df2['timestamp'] = Timestamp('20120101')

In [12]: df2
Out[12]:
   one two three four five  timestamp
0  NaN -0.282863 -1.509059 bar  True 2012-01-01
1  NaN  1.212112 -0.173215 bar False 2012-01-01
2  NaN -1.135632  1.212112 bar  True 2012-01-01
3  NaN  0.119209 -0.861849 bar False 2012-01-01
4  NaN -2.104569 -0.494929 bar False 2012-01-01
5  NaN  0.721555 -1.039575 bar  True 2012-01-01

In [13]: df2.ix[['a','c','h'],['one','timestamp']] = np.nan

In [14]: df2
Out[14]:
   one two three four five  timestamp
0  NaN -0.282863 -1.509059 bar  True  NaT
1  NaN  1.212112 -0.173215 bar False  NaT
2  NaN -1.135632  1.212112 bar  True  NaT
3  NaN  0.119209 -0.861849 bar False  NaT
4  NaN -2.104569 -0.494929 bar False  NaT
5  NaN  0.721555 -1.039575 bar  True  NaT

In [15]: df2.get_dtype_counts()
Out[15]:
bool 1
datetime64[ns] 1
float64 3
object 1
dtype: int64

16.3 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 [16]: s = pd.Series([1, 2, 3])

In [17]: s.loc[0] = None

In [18]: s
Out[18]:
0  NaN
1  2
dtype: float64

Likewise, datetime containers will always use NaT.

For object containers, pandas will use the value given:

```
In [19]: s = pd.Series(["a", "b", "c"])
In [20]: s.loc[0] = None
In [21]: s.loc[1] = np.nan
In [22]: s
Out[22]:
0   None
1   NaN
2    c
dtype: object
```

### 16.4 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [23]: a
Out[23]:
     one     two
a   NaN -0.282863
b -2.104569 -0.494929
h -2.104569 -0.706771

In [24]: b
Out[24]:
     one     two     three
a   NaN -0.282863 -1.509059
b -2.104569 -0.494929  1.071804
h   NaN -0.706771 -1.039575

In [25]: a + b
Out[25]:
     one     three     two
a   NaN   NaN -0.565727
b  0.800000  0.505147  2.097339
h  0.800000  0.293829 -0.373164
```

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 [26]: df
Out [26]:
   one    two    three
  a  NaN -0.282863 -1.509059
  c  NaN  1.212112 -0.173215
  e  0.119209 -1.044236 -0.861849
  f -2.104569 -0.494929  1.071804
  h  NaN  0.706771 -1.039575

In [27]: df['one'].sum()
Out [27]: -1.9853605075978744

In [28]: df.mean(1)
Out [28]:
   a  -0.895961
   c   0.519449
   e  -0.595625
   f  -0.509232
   h  -0.873173
dtype: float64

In [29]: df.cumsum()
Out [29]:
   one    two    three
  a  NaN -0.282863 -1.509059
  c  NaN  0.929249 -1.682273
  e  0.119209 -0.114987 -2.544122
  f -1.985361 -0.609917 -1.472318
  h  NaN  1.316688 -2.511893
```

16.4.1 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example.

16.5 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

16.5.1 Filling missing values: `fillna`

The `fillna` function can “fill in” NA values with non-null data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```python
In [30]: df2
Out [30]:
   one    two    three    four    five          timestamp
  a  NaN -0.282863 -1.509059   bar   True    NaT
  c  NaN  1.212112 -0.173215  bar  False    NaT
  e  0.119209 -1.044236 -0.861849  bar  True  2012-01-01
  f -2.104569 -0.494929  1.071804  bar  False  2012-01-01
  h  NaN  0.706771 -1.039575   bar   True    NaT
```
In [31]: df2.fillna(0)
Out[31]:
    one   two   three   four  five  timestamp
a 0.000000 -0.282863 -1.509059  bar   True  1970-01-01
b 0.000000  1.212112 -0.173215  bar  False  1970-01-01
c 0.119209 -1.044236 -0.861849  bar   True  2012-01-01
d -2.104569 -0.494929  1.071804  bar  False  2012-01-01

In [32]: df2['four'].fillna('missing')
Out[32]:
a    bar
b    bar
c    bar
d    bar
e    bar
f    bar
h    bar
Name: four, dtype: object

Fill gaps forward or backward

Using the same filling arguments as reindexing, we can propagate non-null values forward or backward:

In [33]: df
Out[33]:
    one   two   three
a  NaN -0.282863 -1.509059
b  NaN  1.212112 -0.173215
c  NaN  NaN  NaN
d  NaN  NaN  NaN
e  NaN  NaN  NaN
f  NaN -0.706771 -1.039575
h  NaN -0.706771 -1.039575

In [34]: df.fillna(method='pad')
Out[34]:
    one   two   three
a  NaN -0.282863 -1.509059
b  NaN  1.212112 -0.173215
c  NaN  NaN  NaN
d  NaN  NaN  NaN
e  NaN  NaN  NaN
f  NaN  NaN  NaN
h  NaN -0.706771 -1.039575

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 [35]: df
Out[35]:
    one   two   three
a  NaN -0.282863 -1.509059
b  NaN  1.212112 -0.173215
c  NaN  NaN  NaN
d  NaN  NaN  NaN
e  NaN  NaN  NaN
f  NaN  NaN  NaN
h  NaN -0.706771 -1.039575

In [36]: df.fillna(method='pad', limit=1)
Out[36]:
    one   two   three
a  NaN -0.282863 -1.509059
b  NaN  1.212112 -0.173215
c  NaN  1.212112 -0.173215
d  NaN  NaN  NaN
e  NaN  NaN  NaN
f  NaN  NaN  NaN
h  NaN -0.706771 -1.039575
To remind you, these are the available filling methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad / ffill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
</tbody>
</table>

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

The `ffill()` function is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

### 16.5.2 Filling with a PandasObject

New in version 0.12.

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```
In [37]: dff = pd.DataFrame(np.random.randn(10,3),columns=list('ABC'))
In [38]: dff.iloc[3:5,0] = np.nan
In [39]: dff.iloc[4:6,1] = np.nan
In [40]: dff.iloc[5:8,2] = np.nan
In [41]: dff
Out[41]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.271860</td>
<td>-0.424972</td>
<td>0.567020</td>
</tr>
<tr>
<td>1</td>
<td>0.276232</td>
<td>-1.087401</td>
<td>-0.673690</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>NaN</td>
<td>-1.157892</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>
```

```
In [42]: dff.fillna(dff.mean())
Out[42]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.271860</td>
<td>-0.424972</td>
<td>0.567020</td>
</tr>
<tr>
<td>1</td>
<td>0.276232</td>
<td>-1.087401</td>
<td>-0.673690</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>-0.140857</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>-0.140857</td>
<td>-0.401419</td>
<td>-1.157892</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>-0.401419</td>
<td>-0.293543</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>-0.293543</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>-0.293543</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>
```
In [43]: dff.fillna(dff.mean()[:, 'B': 'C'])
Out[43]:
   A     B     C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3  NaN  0.577046 -1.715002
4  NaN -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960

New in version 0.13.

Same result as above, but is aligning the ‘fill’ value which is a Series in this case.

In [44]: dff.where(notnull(dff), dff.mean(), axis='columns')
Out[44]:
   A     B     C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3 -0.140857  0.577046 -1.715002
4 -0.140857 -0.401419 -0.293543
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960

16.5.3 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 [45]: df
Out[45]:
   one    two    three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN    0.000000    0.000000
f  NaN    0.000000    0.000000
h  NaN -0.706771 -1.039575

In [46]: df.dropna(axis=0)
Out[46]:
Empty DataFrame
Columns: [one, two, three]
Index: []

In [47]: df.dropna(axis=1)
Out[47]:
   two    three
a -0.282863 -1.509059
c  1.212112 -0.173215
e    0.000000    0.000000
f    0.000000    0.000000

Chapter 16. Working with missing data
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.

### 16.5.4 Interpolation

New in version 0.13.0: interpolate(), and interpolate() have revamped interpolation methods and functionality.

Both Series and DataFrame objects have an interpolate method that, by default, performs linear interpolation at missing datapoints.

```python
In [49]: ts
Out[49]:
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, dtype: float64
```

```python
In [50]: ts.count()
Out[50]: 61

In [51]: ts.interpolate().count()
Out[51]: 100

In [52]: ts.interpolate().plot()
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x9b28fc8c>
```
Index aware interpolation is available via the `method` keyword:

```
In [53]: ts2
Out[53]:
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 [54]: ts2.interpolate()
Out[54]:
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 [55]: ts2.interpolate(method='time')
Out[55]:
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':

In [56]: ser
Out[56]:
   0   0
   1  NaN
  10  10
dtype: float64

In [57]: ser.interpolate()
Out[57]:
   0   0
   1   5
  10  10
dtype: float64

In [58]: ser.interpolate(method='values')
Out[58]:
   0   0
   1   1
  10  10
dtype: float64

You can also interpolate with a DataFrame:

In [59]: 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 [60]: df
Out[60]:
   A    B
0   1   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 [61]: df.interpolate()
Out[61]:
   A    B
0   1   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

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. For example, 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.
Warning: These methods require scipy.

```
In [62]: df.interpolate(method='barycentric')
Out[62]:
   A    B
0  1.0  0.25
1  2.1 -7.66
2  3.5 -4.51
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40

In [63]: df.interpolate(method='pchip')
Out[63]:
   A       B
0  1.00  0.250
1  2.10  1.130
2  3.43  2.338
3  4.70  4.000
4  5.60 12.200
5  6.80 14.400

When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

In [64]: df.interpolate(method='spline', order=2)
Out[64]:
   A       B
0  1.00  0.250
1  2.10 -0.429
2  3.40  1.207
3  4.70  4.000
4  5.60 12.200
5  6.80 14.400

In [65]: df.interpolate(method='polynomial', order=2)
Out[65]:
   A       B
0  1.00  0.250
1  2.10 -4.162
2  3.55 -2.912
3  4.70  4.000
4  5.60 12.200
5  6.80 14.400

Compare several methods:

In [66]: np.random.seed(2)

In [67]: ser = pd.Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))

In [68]: bad = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])

In [69]: ser[bad] = np.nan

In [70]: methods = ['linear', 'quadratic', 'cubic']

In [71]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})
```
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 [73]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [74]: new_index = ser.index | Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [75]: interp_s = ser.reindex(new_index).interpolate(method='pchip')
```

```
In [76]: interp_s[49:51]
Out[76]:
49.00  0.471410
49.25  0.476841
49.50  0.481780
49.75  0.485998
50.00  0.489266
50.25  0.491814
50.50  0.493995
50.75  0.495763
```
Like other pandas fill methods, `interpolate` accepts a `limit` keyword argument. Use this to limit the number of consecutive interpolations, keeping NaN values for interpolations that are too far from the last valid observation:

```
In [77]: ser = pd.Series([1, 3, np.nan, np.nan, np.nan, 11])
```

```
In [78]: ser.interpolate(limit=2)
Out[78]:
0   1
1   3
2   5
3   7
4  NaN
5  11
dtype: float64
```

### 16.5.5 Replacing Generic Values

Often times we want to replace arbitrary values with other values. New in v0.8 is the `replace` method in Series/DataFrame that 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 [79]: ser = pd.Series([0., 1., 2., 3., 4.])
```

```
In [80]: ser.replace(0, 5)
Out[80]:
0   5
1   1
2   2
3   3
4   4
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [81]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
```

```
Out[81]:
0   4
1   3
2   2
3   1
4   0
dtype: float64
```

You can also specify a mapping dict:

```
In [82]: ser.replace({0: 10, 1: 100})
```

```
Out[82]:
0  10
1 100
2   2
3   3
4   4
dtype: float64
```
For a DataFrame, you can specify individual values by column:

In [83]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [84]: df.replace({'a': 0, 'b': 5}, 100)
Out[84]:
   a  b
0 100 100
1  1  6
2  2  7
3  3  8
4  4  9

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

In [85]: ser.replace([1, 2, 3], method='pad')
Out[85]:
0 0
dtype: float64

16.5.6 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)

In [86]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}

In [87]: df = pd.DataFrame(d)

In [88]: df.replace('.', np.nan)
Out[88]:
   a  b  c
0  0  a  a
1  1  b  b
2  2  NaN NaN
3  3  NaN  d

don['a', 'b', np.nan, 'd']}

In [87]: df = pd.DataFrame(d)

In [88]: df.replace('.', np.nan)
Out[88]:
   a  b  c
0  0  a  a
1  1  b  b
2  2  NaN NaN
3  3  NaN  d

Now do it with a regular expression that removes surrounding whitespace (regex -> regex)

In [89]: df.replace(r'\s*\.\s*', np.nan, regex=True)
Out[89]:
   a  b  c
0  0  a  a
1  1  b  b
2  2  NaN NaN
3  3  NaN  d

Replace a few different values (list -> list)

In [86]: df = pd.DataFrame(d)

In [87]: df.replace(['a', 'b', np.nan, 'd'], ['a', 'b', np.nan, 'd'])
Out[87]:
   a  b  c
0  0  a  a
1  1  b  b
2  2  NaN NaN
3  3  NaN  d

16.5. Cleaning / filling missing data 481
In [90]: df.replace(['a', ' '], ['b', np.nan])
Out[90]:
   a  b  c
0 0  b  b
1 1  b  b
2 2  NaN NaN
3 3  NaN  d

list of regex -> list of regex

In [91]: df.replace([r' .', r'(a)'], ['dot', '\1stuff'], regex=True)
Out[91]:
   a  b  c
0 0  {stuff {stuff
1 1  b  b
2 2  dot  NaN
3 3  dot  d

Only search in column ’b’ (dict -> dict)

In [92]: df.replace({'b': '.'}, {'b': np.nan})
Out[92]:
   a  b  c
0 0  a  a
1 1  b  b
2 2  NaN NaN
3 3  NaN  d

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict)

In [93]: df.replace({'b': r'\s*\.'}, {'b': np.nan}, regex=True)
Out[93]:
   a  b  c
0 0  a  a
1 1  b  b
2 2  NaN NaN
3 3  NaN  d

You can pass nested dictionaries of regular expressions that use regex=True

In [94]: df.replace({'b': {'b': r''}}, regex=True)
Out[94]:
   a  b  c
0 0  a  a
1 1  b  b
2 2  .  NaN
3 3  .  d

or you can pass the nested dictionary like so

In [95]: df.replace(regex={'b': {r'\s*\.': np.nan}})
Out[95]:
   a  b  c
0 0  a  a
1 1  b  b
2 2  NaN NaN
3 3  NaN  d

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well
You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex)

In [97]: df.replace([r'\s*\.\s*', r'a|b'], np.nan, regex=True)
Out[97]:
    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 [98]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=np.nan)
Out[98]:
    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.

### 16.5.7 Numeric Replacement

Similar to DataFrame.fillna

In [99]: df = pd.DataFrame(np.random.randn(10, 2))

In [100]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5

In [101]: df.replace(1.5, np.nan)
Out[101]:
          0   1
0     -0.844214   NaN
1      0.432396  -0.323580
2      0.423825   0.799180
3      1.262614   0.751965
4        NaN      NaN
5        NaN      NaN
6    -0.498174  -1.060799
7      0.591667  -0.183257
8     1.019855  -1.482465
9        NaN      NaN
Replacing more than one value via lists works as well

In [102]: df00 = df.values[0, 0]

In [103]: df.replace([1.5, df00], [np.nan, 'a'])
Out[103]:
      0      1  
0  a  -1.021415
1  0.4323957 -0.323580
2  0.4238247  0.799180
3  1.262614  0.751965
4  NaN    NaN
5  NaN    NaN
6 -0.4981742 -1.021415
7  0.5918665 -0.183257
8  1.0198557 -1.482465
9  NaN    NaN

In [104]: df[1].dtype
Out[104]: dtype('float64')

You can also operate on the DataFrame in place

In [105]: 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 [106]: s = pd.Series([True, False, True])

In [107]: s.replace('a string', 'another string')
Out[107]:
      0      1
0  True    False
1  True     True

dtype: bool

the original 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.

16.6 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:
data type | Cast to
---|---
integer | float
boolean | object
float | no cast
object | no cast

For example:

```
In [108]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])
In [109]: s > 0
Out[109]:
0   True
2   True
4   True
6   True
7   True
dtype: bool
In [110]: (s > 0).dtype
Out[110]: dtype('bool')
In [111]: crit = (s > 0).reindex(list(range(8)))
In [112]: crit
Out[112]:
0   True
1   NaN
2   True
3   NaN
4   True
5   NaN
6   True
7   True
dtype: object
In [113]: crit.dtype
Out[113]: 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 NaNs, an exception will be generated:

```
In [114]: reindexed = s.reindex(list(range(8))).fillna(0)
In [115]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                              Traceback (most recent call last)
<ipython-input-115-2da204ed1ac7> in <module>()
      1 reindexed[crit]
----> 2 self[key]
/home/joris/scipy/pandas/pandas/core/series.pyc in __getitem__(self, key)
     556     key = list(key)
     557 --> 558     if is_bool_indexer(key):
     559         key = check_bool_indexer(self.index, key)
     560
/home/joris/scipy/pandas/pandas/core/common.pyc in is_bool_indexer(key)
```

16.6. Missing data casting rules and indexing
if not lib.is_bool_array(key):
    if isnull(key).any():
        raise ValueError('cannot index with vector containing NA / NaN values')
    return False

ValueError: cannot index with vector containing NA / NaN values

However, these can be filled in using `fillna` and it will work fine:

```python
In [116]: reindexed[crit.fillna(False)]
Out[116]:
0   0.126504
2   0.696198
4   0.697416
6   0.601516
7   0.003659
dtype: float64

In [117]: reindexed[crit.fillna(True)]
Out[117]:
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
```
GROUP BY: SPLIT-APPLY-COMBINE

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting**: the data into groups based on some criteria
- **Applying**: a function to each group independently
- **Combining**: the results into a data structure

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 method 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.
17.1 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
>>> 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.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`, but it makes life simpler
- A list of any of the above things

Collectively we refer to the grouping objects as the keys. For example, consider the following DataFrame:

```python
In [1]: df = DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...:                   'foo', 'bar', 'foo', 'foo'],
...:                   'B' : ['one', 'one', 'two', 'three',
...:                    'two', 'two', 'one', 'three'],
...:                   'C' : randn(8), 'D' : randn(8)})
```

```python
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:

```python
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)
```
Starting with 0.8, pandas Index objects now supports 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:

```
In [7]: lst = [1, 2, 3, 1, 2, 3]
In [8]: s = 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.

### 17.1.1 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 [13]: df.groupby('A').groups
Out[13]: {'bar': [1L, 3L, 5L], 'foo': [0L, 2L, 4L, 6L, 7L]}
In [14]: df.groupby(get_letter_type, axis=1).groups
Out[14]: {'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [15]: grouped = df.groupby(['A', 'B'])
In [16]: grouped.groups
Out[16]:
{(‘bar’, ‘one’): [1L],
 (‘bar’, ‘three’): [3L],
```
('bar', 'two'): [5L],
('foo', 'one'): [0L, 6L],
('foo', 'three'): [7L],
('foo', 'two'): [2L, 4L])

In [17]: len(grouped)
Out[17]: 6

By default the group keys are sorted during the groupby operation. You may however pass sort=False for potential speedups:

In [18]: df2 = DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [19]: df2.groupby(['X'], sort=True).sum()
Out[19]:
   Y
X  
A  7
B  3

In [20]: df2.groupby(['X'], sort=False).sum()
Out[20]:
   Y
X  
B  3
A  7

GroupBy will tab complete column names (and other attributes)

In [21]: df
Out[21]:
   gender  height  weight
2000-01-01  male  42.849980  157.500553
2000-01-02  male  49.607315  177.340407
2000-01-03  male  56.293531  171.524640
2000-01-04  female  48.421077  144.251986
2000-01-05  male  46.556882  152.526206
2000-01-06  female  68.448851  168.272953
2000-01-07  male  70.757698  136.431469
2000-01-08  female  58.909500  168.272953
2000-01-09  female  76.435631  174.094104
2000-01-10  male  45.306120  177.540920

In [22]: gb = df.groupby('gender')

In [23]: gb.

17.1.2 GroupBy with MultiIndex

With hierarchically-indexed data, it’s quite natural to group by one of the levels of the hierarchy.

In [24]: s
Out[24]:
   first  second
bar  one  -0.575247
two  0.254161
baz  one  -1.143704
two  0.215897
foo  one  1.193555
two  -0.077118
qux  one  -0.408530
two  -0.862495
dtype: float64

In [25]: grouped = s.groupby(level=0)

In [26]: grouped.sum()
Out[26]:
   first
bar  -0.321085
baz  -0.927807
foo   1.116437
qux  -1.271025
dtype: float64

If the MultiIndex has names specified, these can be passed instead of the level number:

In [27]: s.groupby(level='second').sum()
Out[27]:
   second
one  -0.933926
two  -0.469555
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 [28]: s.sum(level='second')
Out[28]:
   second
one  -0.933926
two  -0.469555
dtype: float64

Also as of v0.6, grouping with multiple levels is supported.

In [29]: s
Out[29]:
   first second third
bar  doo one  1.346061
two  1.511763
baz  bee one  1.627081
two  -0.990582
foo  bop one -0.441652
two  1.211526
qux  bop one  0.268520
two  0.024580
dtype: float64

In [30]: s.groupby(level=['first','second']).sum()
Out[30]:
   first second
bar  doo    2.857824

17.1. Splitting an object into groups
More on the `sum` function and aggregation later.

### 17.1.3 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 [31]: grouped = df.groupby(['A'])
In [32]: grouped_C = grouped['C']
In [33]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```python
In [34]: df['C'].groupby(df['A'])
Out[34]: <pandas.core.groupby.SeriesGroupBy object at 0xa0c4de8c>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

### 17.2 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 [35]: grouped = df.groupby('A')
In [36]: for name, group in grouped:
.....:     print(name)
.....:     print(group)
.....:
bar
   A     B     C     D
  1 bar one -0.042379 -0.089329
  3 bar three -0.009920 -0.945867
  5 bar two  0.495767  1.956030
foo
   A     B     C     D
  0 foo one -0.919854 -1.131345
  2 foo two  1.247642  0.337863
  4 foo two  0.290213 -0.932132
  6 foo one  0.362949  0.017587
  7 foo three  1.548106 -0.016692
```

In the case of grouping by multiple keys, the group name will be a tuple:

```python
In [37]: for name, group in df.groupby(['A', 'B']):
.....:     print(name)
.....:     print(group)
.....:
('bar', 'one')
```
It's standard Python-fu but remember you can unpack the tuple in the for loop statement if you wish:

```python
for (k1, k2), group in grouped:
```

### 17.3 Selecting a group

A single group can be selected using `GroupBy.get_group()`:

```python
In [38]: grouped.get_group('bar')
Out[38]:
   A   B    C    D
1  bar one 0.042379 0.089329
3  bar three -0.00992 -0.945867
5  bar two 0.495767 1.95603
```

Or for an object grouped on multiple columns:

```python
In [39]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[39]:
   A   B    C    D
1  bar one 0.042379 0.089329
```

### 17.4 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data.

An obvious one is aggregation via the `aggregate` or equivalently `agg` method:

```python
In [40]: grouped = df.groupby('A')

In [41]: grouped.aggregate(np.sum)
Out[41]:
   C    D
A
bar 0.443469 0.920834
```
foo 2.529056 -1.724719

In [42]: grouped = df.groupby(['A', 'B'])

In [43]: grouped.aggregate(np.sum)
Out[43]:

    C    D
   --- ---
   A B  ---
   bar one -0.042379 -0.089329
          three -0.009920 -0.945867
          two  0.495767  1.956030
   foo one -0.556905 -1.113758
          three  1.548106 -0.016692
          two  1.537855 -0.594269

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:

In [44]: grouped = df.groupby(['A', 'B'], as_index=False)

In [45]: grouped.aggregate(np.sum)
Out[45]:

   A B C    D
   --- --- --- ---
   0 bar one -0.042379 -0.089329
   1 bar three -0.009920 -0.945867
   2 bar two  0.495767  1.956030
   3 foo one -0.556905 -1.113758
   4 foo three  1.548106 -0.016692
   5 foo two  1.537855 -0.594269

In [46]: df.groupby('A', as_index=False).sum()
Out[46]:

   A C    D
   --- --- ---
   0 bar 0.443469 0.920834
   1 foo 2.529056 -1.724719

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 [47]: df.groupby(['A', 'B']).sum().reset_index()
Out[47]:

   A   B   C   D
   --- --- --- ---
   0 bar one -0.042379 -0.089329
   1 bar three -0.009920 -0.945867
   2 bar two  0.495767  1.956030
   3 foo one -0.556905 -1.113758
   4 foo three  1.548106 -0.016692
   5 foo two  1.537855 -0.594269

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 [48]: grouped.size()
Out[48]:

   A B
   --- ---
   bar one 1
          three 1
          two 1

494 Chapter 17. Group By: split-apply-combine
foo one 2
three 1
two 2
dtype: int64

In [49]: grouped.describe()
Out[49]:
          C     D
0  count 1.000000 1.000000
  mean -0.042379 -0.089329
  std  NaN         NaN
  min -0.042379 -0.089329
  25% -0.042379 -0.089329
  50% -0.042379 -0.089329
  75% -0.042379 -0.089329
     ...     ...     ...
  5  mean 0.768928  0.297134
  std 0.677005 0.898022
  min 0.290213 0.932132
  25% 0.529570 0.614633
  50% 0.768928 0.297134
  75% 1.008285 0.020364
  max 1.247642 0.337863

[48 rows x 2 columns]

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

17.4.1 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:

In [50]: grouped = df.groupby('A')

In [51]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[51]:
        sum      mean     std
A         bar  0.443469  0.147823  0.301765
         foo  2.529056  0.505811  0.966450

If a dict is passed, the keys will be used to name the columns. Otherwise the function’s name (stored in the function object) will be used.

In [52]: grouped['D'].agg({'result1' : np.sum,
   ...:                      'result2' : np.mean})
   ...:
Out[52]:

17.4. Aggregation
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:

```
In [53]: grouped.agg([np.sum, np.mean, np.std])
Out[53]:
    C     D
  sum  mean   std  sum  mean   std
A  bar  0.443469  0.147823  0.301765  0.920834  0.306945  1.490982
    foo  2.529056  0.505811  0.966450 -1.724719 -0.344944  0.645875
```

Passing a dict of functions has different behavior by default, see the next section.

### 17.4.2 Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```
In [54]: grouped.agg({'C' : np.sum, 'D' : lambda x: np.std(x, ddof=1)})
Out[54]:
    C     D
A  bar  0.443469  1.490982
    foo  2.529056  0.645875
```

The function names can also be strings. In order for a string to be valid it must be either implemented on GroupBy or available via dispatching:

```
In [55]: grouped.agg({'C' : 'sum', 'D' : 'std'})
Out[55]:
    C     D
A  bar  0.443469  1.490982
    foo  2.529056  0.645875
```

### 17.4.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, `std`, and `sem`, have optimized Cython implementations:

```
In [56]: df.groupby('A').sum()
Out[56]:
    C     D
A  bar  0.443469  0.920834
    foo  2.529056 -1.724719
```

```
In [57]: df.groupby(['A', 'B']).mean()
Out[57]:
    C     D
A  B   bar one  -0.042379 -0.089329
```
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).

### 17.5 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. Thus, the passed transform function should return a result that is the same size as the group chunk. For example, suppose we wished to standardize the data within each group:

```python
# Original Data
In [66]: grouped = ts.groupby(key)
In [67]: grouped.mean()
Out[67]:
2000  0.442441
2001  0.526246
2002  0.459365
```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```python
# Original Data
In [66]: grouped = ts.groupby(key)
In [67]: grouped.mean()
Out[67]:
2000  0.442441
2001  0.526246
2002  0.459365
```
In [68]: grouped.std()
Out[68]:
2000 0.131752
2001 0.210945
2002 0.128753
dtype: float64

# Transformed Data
In [69]: grouped_trans = transformed.groupby(key)

In [70]: grouped_trans.mean()
Out[70]:
2000 -1.250934e-16
2001 -4.291848e-16
2002 2.404815e-17
dtype: float64

In [71]: grouped_trans.std()
Out[71]:
2000 1
2001 1
2002 1
dtype: float64

We can also visually compare the original and transformed data sets.

In [72]: compare = DataFrame({'Original': ts, 'Transformed': transformed})

In [73]: compare.plot()
Out[73]: <matplotlib.axes._subplots.AxesSubplot at 0xa46880ec>
Another common data transform is to replace missing data with the group mean.

In [74]: data_df
Out [74]:

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  1.924296  0.396437  0.812436
5  0.815643  0.376816 -0.469478
6 -0.030651  1.376106 -0.645129
... ...
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 [75]: countries = np.array(['US', 'UK', 'GR', 'JP'])
In [76]: key = countries[np.random.randint(0, 4, 1000)]
In [77]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [78]: grouped.count()
Out[78]:
     A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [79]: f = lambda x: x.fillna(x.mean())
In [80]: transformed = grouped.transform(f)

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.
In [81]: grouped_trans = transformed.groupby(key)
In [82]: grouped.mean()  # original group means
Out[82]:
     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 [83]: grouped_trans.mean()  # transformation did not change group means
Out[83]:
     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 [84]: grouped.count()  # original has some missing data points
Out[84]:
     A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [85]: grouped_trans.count()  # counts after transformation
Out[85]:
     A    B    C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258

In [86]: grouped_trans.size()  # Verify non-NA count equals group size
Out[86]:
500

Chapter 17. Group By: split-apply-combine
GR  228
JP  267
UK  247
US  258
dtype: int64

**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`.

```python
In [87]: grouped.ffill()
Out[87]:
   A     B     C
0 1.539708 -1.166480 0.533026
1 1.302092 -0.505754 0.533026
2 -0.371983 1.104803 -1.616657
3 -1.309622 1.116897 -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]
```

## 17.6 Filtration

New in version 0.12.

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.

```python
In [88]: sf = Series([1, 1, 2, 3, 3, 3])

In [89]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[89]:
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.

```python
In [90]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc'))

In [91]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[91]:
```

### 17.6. Filtration
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 [92]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
```

```
Out[92]:
   A  B
0  NaN  NaN
1  NaN  NaN
2  2  b
3  3  b
4  4  b
5  5  b
6  NaN  NaN
7  NaN  NaN
```

For dataframes with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [93]: dff['C'] = np.arange(8)
```

```
In [94]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
```

```
Out[94]:
     A  B  C
0  NaN  NaN  NaN
1  NaN  NaN  NaN
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 [95]: dff.groupby('B').head(2)
```

```
Out[95]:
   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
```

## 17.7 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 [96]: grouped = df.groupby('A')
```
In [97]: grouped.agg(lambda x: x.std())
Out[97]:
      C     D
A  bar  0.301765  1.490982
    foo  0.966450  0.645875

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 [98]: grouped.std()
Out[98]:
      C     D
A  bar  0.301765  1.490982
    foo  0.966450  0.645875

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 [99]: tsdf = DataFrame(randn(1000, 3),
      index=date_range('1/1/2000', periods=1000),
      columns=['A', 'B', 'C'])

In [100]: tsdf.ix[::2] = np.nan

In [101]: grouped = tsdf.groupby(lambda x: x.year)

In [102]: grouped.fillna(method='pad')
Out[102]:
    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

[1000 rows x 3 columns]

In this example, we chopped the collection of time series into yearly chunks then independently called fillna on the
groups.

New in version 0.14.1.

The nlargest and nsmallest methods work on Series style groupbys:
In [103]: s = Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])
In [104]: g = Series(list('abababab'))
In [105]: gb = s.groupby(g)
In [106]: gb.nlargest(3)
Out[106]:
   a 4  19.0
      0  9.0
      2  7.0
   b 1  8.0
      3  5.0
      7  3.3
dtype: float64
In [107]: gb.nsmallest(3)
Out[107]:
   a 6  4.2
      2  7.0
      0  9.0
   b 5  1.0
      7  3.3
      3  5.0
dtype: float64

17.8 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:

In [108]: df
Out[108]:
   A   B     C     D
0  foo  one -0.919854 -1.131345
1  bar  one -0.042379 -0.089329
2  foo  two  1.247642  0.337863
3  bar  three -0.009920  0.945867
4  foo  two  0.290213  0.932132
5  bar  two  0.495767  1.956030
6  foo  one  0.362949  0.017587
7  foo  three  1.548106 -0.016692

In [109]: grouped = df.groupby('A')

# could also just call .describe()
In [110]: grouped['C'].apply(lambda x: x.describe())
Out[110]:
   A
bar   count 3.000000
      mean 0.147823
      std 0.301765
      min -0.042379
     25% -0.026149
50%  -0.009920
75%   0.242924
...
foo  mean  0.505811
std   0.966450
min  -0.919854
25%  0.290213
50%  0.362949
75%  1.247642
max  1.548106
dtype: float64

The dimension of the returned result can also change:

In [111]: grouped = df.groupby('A')['C']
In [112]: def f(group):
       .....:     return DataFrame({'original' : group,
       .....:                      'demeaned' : group - group.mean()})
       .....:
In [113]: grouped.apply(f)
Out[113]:
   demeaned  original
0 -1.425665 -0.919854
1 -0.190202 -0.042379
2  0.741831  1.247642
3 -0.157743 -0.009920
4 -0.215598  0.290213
5  0.347944  0.495767
6 -0.142862  0.362949
7  1.042295  1.548106

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

In [114]: def f(x):
       .....:     return Series([ x, x**2 ], index = ['x', 'x^s'])
       .....:
In [115]: s
Out[115]:
   0 9.0
   1 8.0
   2 7.0
   3 5.0
   4 19.0
   5 1.0
   6 4.2
   7 3.3
dtype: float64
In [116]: s.apply(f)
Out[116]:
   x  x^s
   0 9.0   81.00
   1 8.0   64.00
   2 7.0   49.00
   3 5.0   25.00

17.8. Flexible apply
Note: apply can act as a reducer, transformer, or filter function, depending on exactly what is passed to apply. 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.

17.9 Other useful features

17.9.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:

Supposed we wished 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 [121]: df.groupby('A').std()
Out[121]:
   C   D
A
bar 0.301765 1.490982
foo 0.966450 0.645875
```

### 17.9.2 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).

### 17.9.3 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 [122]: data = Series(np.random.randn(100))
In [123]: factor = qcut(data, [0, .25, .5, .75, 1.])
In [124]: data.groupby(factor).mean()
Out[124]:
[[-2.617, -0.684]  -1.331461
 (-0.684, -0.0232] -0.272816
 (-0.0232, 0.541]  0.263607
 (0.541, 2.369]   1.166038
dtype: float64
```

### 17.9.4 Grouping with a Grouper specification

Your may need to specify a bit more data to properly group. You can use the `pd.Grouper` to provide this local control.

```
In [125]: import datetime as DT
In [126]: df = DataFrame({'Branch': 'A A A A A A A B'.split(),
                     'Buyer': 'Carl Mark Carl Carl Joe Joe Joe Carl'.split(),
                     'Quantity': [1,3,5,1,8,1,9,3],
                     'Date': [DT.datetime(2013,1,1,13,0),
                              DT.datetime(2013,1,1,13,5),
                              DT.datetime(2013,10,1,20,0),
                              DT.datetime(2013,10,2,10,0),
                              DT.datetime(2013,10,1,20,0),
                              DT.datetime(2013,10,2,10,0),
                              DT.datetime(2013,12,2,12,0),
                              DT.datetime(2013,12,2,14,0),]})
```

17.9. Other useful features 507
In [127]: df
Out[127]:

<table>
<thead>
<tr>
<th>Branch</th>
<th>Buyer</th>
<th>Date</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2013-01-01 13:00:00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>2013-01-01 13:05:00</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>2013-10-01 20:00:00</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>2013-10-02 10:00:00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>2013-10-01 20:00:00</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>2013-10-02 10:00:00</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>2013-12-02 12:00:00</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>2013-12-02 14:00:00</td>
<td>3</td>
</tr>
</tbody>
</table>

Groupby a specific column with the desired frequency. This is like resampling.

In [128]: df.groupby([pd.Grouper(freq='1M',key='Date'),'Buyer']).sum()
Out[128]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-31</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2013-10-31</td>
<td>Carl</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>9</td>
</tr>
<tr>
<td>2013-12-31</td>
<td>Carl</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>9</td>
</tr>
</tbody>
</table>

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

In [129]: df = df.set_index('Date')
In [130]: df['Date'] = df.index + pd.offsets.MonthEnd(2)

In [131]: df.groupby([pd.Grouper(freq='6M',key='Date'),'Buyer']).sum()
Out[131]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-02-28</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2014-02-28</td>
<td>Carl</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>18</td>
</tr>
</tbody>
</table>

In [132]: df.groupby([pd.Grouper(freq='6M',level='Date'),'Buyer']).sum()
Out[132]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-31</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2014-01-31</td>
<td>Carl</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>18</td>
</tr>
</tbody>
</table>

17.9.5 Taking the first rows of each group

Just like for a DataFrame or Series you can call head and tail on a groupby:

In [133]: df = DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
In [134]: df
Out[134]:
            A  B
    0  1  2
    1  1  4
    2  5  6

In [135]: g = df.groupby('A')

In [136]: g.head(1)
Out[136]:
            A  B
    0  1  2  
    2  5  6  

In [137]: g.tail(1)
Out[137]:
            A  B
    1  1  4  
    2  5  6  

This shows the first or last n rows from each group.

**Warning:** Before 0.14.0 this was implemented with a fall-through apply, so the result would incorrectly respect the as_index flag:

```python
>>> g.head(1):  # was equivalent to g.apply(lambda x: x.head(1))
    A  B
    A
    1  0  1  2
    5  2  5  6
```

### 17.9.6 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:

```
In [138]: df = DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [139]: g = df.groupby('A')

In [140]: g.nth(0)
Out[140]:
            B
    A
    1  NaN
    5  6

In [141]: g.nth(-1)
Out[141]:
            B
    A
    1  4
    5  6

In [142]: g.nth(1)
Out[142]:
```
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`, for a Series this just needs to be truthy.

```
# nth(0) is the same as g.first()
In [143]: g.nth(0, dropna='any')
Out[143]:
B
A  
  1 4
  5 6

In [144]: g.first()
Out[144]:
B
A  
  1 4
  5 6

# nth(-1) is the same as g.last()
In [145]: g.nth(-1, dropna='any')  # NaNs denote group exhausted when using dropna
Out[145]:
B
A  
  1 4
  5 6

In [146]: g.last()
Out[146]:
B
A  
  1 4
  5 6

In [147]: g.B.nth(0, dropna=True)
Out[147]:
A  
  1 4
  5 6
Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```
In [148]: df = DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [149]: g = df.groupby('A', as_index=False)

In [150]: g.nth(0)
Out[150]:
A  
  0 1 NaN
  2 5 6

In [151]: g.nth(-1)
Out[151]:
A  
  0 1 NaN
  2 5 6
```
You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

```python
In [152]: business_dates = date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [153]: df = DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [154]: df.groupby((df.index.year, df.index.month)).nth([0, 3, -1])
Out[154]:
        a   b
2014-04-01  1   1
2014-04-04  1   1
2014-04-30  1   1
2014-05-01  1   1
2014-05-30  1   1
2014-06-02  1   1
2014-06-05  1   1
2014-06-30  1   1
```

### 17.9.7 Enumerate group items

New in version 0.13.0.

To see the order in which each row appears within its group, use the `cumcount` method:

```python
In [155]: df = pd.DataFrame(list('aaabba'), columns=['A'])

In [156]: df
Out[156]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [157]: df.groupby('A').cumcount()
Out[157]:
     0  1  2  3  4  5
0  0  1  1  2  3  4
1  1  1  2  3  4  5
dtype: int64

In [158]: df.groupby('A').cumcount(ascending=False)  # kwarg only
Out[158]:
     0  1  2  3  4  5
0  1  2  3  4  5  0
1  4  3  2  1  0  5
```

17.9. Other useful features
17.9.8 Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame differ by group, in this case, the values in column 1 where the group is “B” are 3 higher on average.

In [159]: np.random.seed(1234)

In [160]: df = DataFrame(np.random.randn(50, 2))

In [161]: df['g'] = np.random.choice(['A', 'B'], size=50)

In [162]: df.loc[df['g'] == 'B', 1] += 3

We can easily visualize this with a boxplot:

In [163]: df.groupby('g').boxplot()
Out[163]: OrderedDict([('A', {'boxes': [<matplotlib.lines.Line2D object at 0xa0f5b04c>, <matplotlib.lines.Line2D object at 0xa0f39c2c>, <matplotlib.lines.Line2D object at 0xa4628a4c>, <matplotlib.lines.Line2D object at 0xa10a7e8c>]}))

The result of calling boxplot is a dictionary whose keys are the values of our grouping column g (“A” and “B”). The values of the resulting dictionary can be controlled by the return_type keyword of boxplot. See the visualization documentation for more.
Warning: For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See here for an explanation.

17.10 Examples

17.10.1 Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

In [164]: df = pd.DataFrame({'a':[1,0,0], 'b':[0,1,0], 'c':[1,0,0], 'd':[2,3,4]})

In [165]: df
Out[165]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4

In [166]: df.groupby(df.sum(), axis=1).sum()
Out[166]:
   1  9
   0  2  2
   1  1  3
   2  0  4

17.10.2 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 [167]: def compute_metrics(x):
   ....:     result = {'b_sum': x['b'].sum(), 'c_mean': x['c'].mean()}
   ....:     return pd.Series(result, name='metrics')

In [168]: result = df.groupby('a').apply(compute_metrics)

In [170]: result
Out[170]:
   metrics  b_sum  c_mean
   a        
   0        2      0.5
   1        2      0.5
   2        2      0.5
In [171]: result.stack()
Out[171]:
a metrics
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
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.

### 18.1 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]:
```python
df1 = 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]:
```python
df2 = 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]:
```python
df3 = 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]:
```python
frames = [df1, df2, df3]
```

In [5]:
```python
result = 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
concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
keys=None, levels=None, names=None, verify_integrity=False)
```

- **objs**: list or dict 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).
- **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.
- **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. If keys passed, specific levels to use for the resulting 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.
- **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.

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:
In [6]: result = 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:

In [7]: result.ix['y']

Out[7]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>A4</td>
<td>B4</td>
<td>C4</td>
<td>D4</td>
</tr>
<tr>
<td>5</td>
<td>A5</td>
<td>B5</td>
<td>C5</td>
<td>D5</td>
</tr>
<tr>
<td>6</td>
<td>A6</td>
<td>B6</td>
<td>C6</td>
<td>D6</td>
</tr>
<tr>
<td>7</td>
<td>A7</td>
<td>B7</td>
<td>C7</td>
<td>D7</td>
</tr>
</tbody>
</table>

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.

frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)

### 18.1.1 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 = DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                        'C': ['C1', 'C2', 'C6', 'C7'],
                        'D': ['D2', 'D3', 'D6', 'D7'],
                        'F': ['F2', 'F3', 'F6', 'F7']},
                        index=[2, 3, 6, 7])

In [9]: result = concat([df1, df4], axis=1)
```

```
<table>
<thead>
<tr>
<th></th>
<th>df1</th>
<th>df4</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>df1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>df4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>A0</td>
<td>B0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A1</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A2</td>
<td>E2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A3</td>
<td>E3</td>
</tr>
<tr>
<td>note</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```python
In [10]: result = concat([df1, df4], axis=1, join='inner')
```

```
<table>
<thead>
<tr>
<th></th>
<th>df1</th>
<th>df4</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>df1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>df4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>A0</td>
<td>B0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A1</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A2</td>
<td>E2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A3</td>
<td>E3</td>
</tr>
</tbody>
</table>
```

Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:

```python
In [11]: result = concat([df1, df4], axis=1, join_axes=[df1.index])
```

```
<table>
<thead>
<tr>
<th></th>
<th>df1</th>
<th>df4</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>df1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>df4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>A0</td>
<td>B0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A1</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A2</td>
<td>E2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A3</td>
<td>E3</td>
</tr>
</tbody>
</table>
```
18.1.2 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

append may take multiple objects to concatenate:

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.

18.1.3 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:

```python
In [15]: result = concat([df1, df4], ignore_index=True)
```

This is also a valid argument to `DataFrame.append`:

```python
In [16]: result = df1.append(df4, ignore_index=True)
```

18.1.4 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]: sl = Series(['X0', 'X1', 'X2', 'X3'], name='X')
In [18]: result = concat([df1, sl], axis=1)
```
If unnamed Series are passed they will be numbered consecutively.

In [19]: s2 = Series(['_0', '_1', '_2', '_3'])

In [20]: result = concat([df1, s2, s2, s2], axis=1)

Passing `ignore_index=True` will drop all name references.

In [21]: result = concat([df1, s1], axis=1, ignore_index=True)

18.1.5 More concatenating with group keys

Let’s consider a variation on the first example presented:

In [22]: result = 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):

```
In [23]: pieces = {'x': df1, 'y': df2, 'z': df3}
In [24]: result = concat(pieces)
```
In [25]: result = concat(pieces, keys=['z', 'y'])

The MultiIndex created has levels that are constructed from the passed keys and the index of the DataFrame pieces:

In [26]: result.index.levels
Out[26]: FrozenList([[u'z', u'y'], [4, 5, 6, 7, 8, 9, 10, 11]])

If you wish to specify other levels (as will occasionally be the case), you can do so using the levels argument:

In [27]: result = concat(pieces, keys=['x', 'y', 'z'],
                      levels=[['z', 'y', 'x', 'w']],
                      names=['group_key'])
In [28]: result.index.levels
Out[28]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])

Yes, this is fairly esoteric, but is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

### 18.1.6 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.

In [29]: s2 = Series({'X0': 'X0', 'X1': 'X1', 'X2': 'X2', 'X3': 'X3'}, index=['A', 'B', 'C', 'D'])

In [30]: 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:

```python
In [31]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
           {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]

In [32]: result = df1.append(dicts, ignore_index=True)
```

### 18.2 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: powerful Python data analysis toolkit, Release 0.16.2

pandas provides a single function, \texttt{merge}, as the entry point for all standard database join operations between DataFrame objects:

\begin{verbatim}
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)
\end{verbatim}

Here’s a description of what each argument is for:

- \texttt{left}: A DataFrame object
- \texttt{right}: Another DataFrame object
- \texttt{on}: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and \texttt{left_index} and \texttt{right_index} are \texttt{False}, the intersection of the columns in the DataFrames will be inferred to be the join keys
- \texttt{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
- \texttt{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
- \texttt{left_index}: If \texttt{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
- \texttt{right_index}: Same usage as \texttt{left_index} for the right DataFrame
- \texttt{how}: One of \texttt{'left'}, \texttt{'right'}, \texttt{'outer'}, \texttt{'inner'}. Defaults to \texttt{inner}. See below for more detailed description of each method
- \texttt{sort}: Sort the result DataFrame by the join keys in lexicographical order. Defaults to \texttt{True}, setting to \texttt{False} will improve performance substantially in many cases
- \texttt{suffixes}: A tuple of string suffixes to apply to overlapping columns. Defaults to \texttt{('_x', '_y')}.
- \texttt{copy}: Always copy data (default \texttt{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.

The return type will be the same as \texttt{left}. If \texttt{left} is a \texttt{DataFrame} and \texttt{right} is a subclass of \texttt{DataFrame}, the return type will still be \texttt{DataFrame}.

\texttt{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 \texttt{DataFrame.join} method, uses \texttt{merge} internally for the index-on-index and index-on-column(s) joins, but \texttt{joins on indexes} by default rather than trying to join on common columns (the default behavior for \texttt{merge}). If you are joining on index, you may wish to use \texttt{DataFrame.join} to save yourself some typing.

\subsection*{18.2.1 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:

- \textbf{one-to-one} joins: for example when joining two DataFrame objects on their indexes (which must contain unique values)
- \textbf{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:

```
In [33]: left = DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                               'A': ['A0', 'A1', 'A2', 'A3'],
                               'B': ['B0', 'B1', 'B2', 'B3']})

In [34]: right = DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                               'C': ['C0', 'C1', 'C2', 'C3'],
                               'D': ['D0', 'D1', 'D2', 'D3']})

In [35]: result = merge(left, right, on='key')
```

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>A0</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>A0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K3</td>
</tr>
</tbody>
</table>

Here is a more complicated example with multiple join keys:

```
In [36]: left = DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                               'key2': ['K0', 'K1', 'K0', 'K1'],
                               'A': ['A0', 'A1', 'A2', 'A3'],
                               'B': ['B0', 'B1', 'B2', 'B3']})

In [37]: right = DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                               'key2': ['K0', 'K0', 'K0', 'K0'],
                               'C': ['C0', 'C1', 'C2', 'C3'],
                               'D': ['D0', 'D1', 'D2', 'D3']})

In [38]: result = 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>

In [39]: result = merge(left, right, how='left', on=['key1', 'key2'])

In [40]: result = merge(left, right, how='right', on=['key1', 'key2'])

In [41]: result = merge(left, right, how='outer', on=['key1', 'key2'])
18.2.2 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:

In [43]: left = DataFrame({'A': ['A0', 'A1', 'A2'],
                        'B': ['B0', 'B1', 'B2'],
                        index=['K0', 'K1', 'K2'])

In [44]: right = DataFrame({'C': ['C0', 'C2', 'C3'],
                        'D': ['D0', 'D2', 'D3'],
                        index=['K0', 'K2', 'K3'])

In [45]: result = left.join(right)
In [46]: result = left.join(right, how='outer')

In [47]: result = left.join(right, how='inner')

The data alignment here is on the indexes (row labels). This same behavior can be achieved using `merge` plus additional arguments instructing it to use the indexes:

In [48]: result = merge(left, right, left_index=True, right_index=True, how='outer')

In [49]: result = merge(left, right, left_index=True, right_index=True, how='inner');
18.2.3 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:

```
left.join(right, on=key_or_keys)
merge(left, right, left_on=key_or_keys, right_index=True,
how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the DataFrame’s is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [50]: left = DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
....:                   'B': ['B0', 'B1', 'B2', 'B3'],
....:                   'key': ['K0', 'K1', 'K0', 'K1']})
....:
In [51]: right = DataFrame({'C': ['C0', 'C1'],
....:                       'D': ['D0', 'D1']},
....:                       index=['K0', 'K1'])
....:
In [52]: result = left.join(right, on='key')
```

```
left  |   right  |  Result
---   |         |  ---
A     |   B     |   key   |   C   |   D
0 A0  |   B0    |   K0    |   C0  |   D0
1 A1  |   B1    |   K1    |   C1  |   D1
2 A2  |   B2    |   K0    |   C0  |   D0
3 A3  |   B3    |   K1    |   C1  |   D1
```

```
In [53]: result = merge(left, right, left_on='key', right_index=True,
....:                    how='left', sort=False);
....:
```

```
left  |   right  |  Result
---   |         |  ---
A     |   B     |   key   |   C   |   D
0 A0  |   B0    |   K0    |   C0  |   D0
1 A1  |   B1    |   K1    |   C1  |   D1
2 A2  |   B2    |   K0    |   C0  |   D0
3 A3  |   B3    |   K1    |   C1  |   D1
```

To join on multiple keys, the passed DataFrame must have a MultiIndex:

```
In [54]: left = DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
....:                   'B': ['B0', 'B1', 'B2', 'B3'],
....:                   'key1': ['K0', 'K0', 'K1', 'K2'],
....:
```
In [55]:
index = MultiIndex.from_tuples([('K0', 'K0'), ('K1', 'K0'), ('K2', 'K0'), ('K2', 'K1')])

In [56]:
right = 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 [57]:
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 [58]:
result = left.join(right, on=['key1', 'key2'], how='inner')

As you can see, this drops any rows where there was no match.

### 18.2.4 Joining a single Index to a Multi-index

New in version 0.14.0.

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 [59]:
left = DataFrame({'A': ['A0', 'A1', 'A2'],
                 'B': ['B0', 'B1', 'B2']},
                index=MultiIndex.from_tuples([('A0', 'B0'), ('A1', 'B1'), ('A2', 'B2')]))

As you can see, this drops any rows where there was no match.
In [60]:
    
    index = MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'), ('K2', 'Y2'), ('K2', 'Y3')], names=['key', 'Y'])
    
In [61]:
    
    right = DataFrame({'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']},
                    index=index)
    
In [62]:
    
    result = left.join(right, how='inner')

This is equivalent but less verbose and more memory efficient / faster than this.

In [63]:
    
    result = merge(left.reset_index(), right.reset_index(), on=['key'], how='inner').set_index(['key', 'Y'])

18.2.5 Joining with two multi-indexes

This is not Implemented via join at-the-moment, however it can be done using the following.

In [64]:
    
    index = MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'), ('K1', 'X2')], names=['key', 'X'])
    
In [65]:
    
    left = DataFrame({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']},
                     index=index)
In [66]: result = merge(left.reset_index(), right.reset_index(),
     on=['key'], how='inner').set_index(['key', 'X', 'Y'])

18.2.6 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 [67]: left = DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})

In [68]: right = DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})

In [69]: result = merge(left, right, on='k')

In [70]: result = merge(left, right, on='k', suffixes=['_l', '_r'])

DataFrame.join has lsuffix and rsuffix arguments which behave similarly.
In [71]: left = left.set_index('k')

In [72]: right = right.set_index('k')

In [73]: result = left.join(right, lsuffix='_l', rsuffix='_r')

18.2.7 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.

In [74]: right2 = DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])

In [75]: result = left.join([right, right2])

18.2.8 Merging Ordered Data

New in v0.8.0 is the ordered_merge function for combining time series and other ordered data. In particular it has an optional fill_method keyword to fill/interpolate missing data:

In [76]: left = DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
                   ....:        'lv': [1, 2, 3, 4],
                   ....:        's': ['a', 'b', 'c', 'd']})

In [77]: right = DataFrame({'k': ['K1', 'K2', 'K4'],
                   ....:        'rv': [1, 2, 3]})
18.2.9 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 [79]: df1 = DataFrame([[nan, 3., 5.], [-4.6, np.nan, nan],
                       [nan, 7., nan]],
                      index=[1, 2])

In [80]: df2 = DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4.]],
                        index=[1, 2])
```

For this, use the `combine_first` method:

```python
In [81]: result = df1.combine_first(df2)
```

<table>
<thead>
<tr>
<th>k</th>
<th>lv</th>
<th>s</th>
<th>k</th>
<th>lv</th>
<th>s</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>K0</td>
<td>1</td>
<td>a</td>
<td>0</td>
<td>K0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>K1</td>
<td>2</td>
<td>b</td>
<td>1</td>
<td>K1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>K1</td>
<td>3</td>
<td>c</td>
<td>2</td>
<td>K2</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>K2</td>
<td>d</td>
<td>d</td>
<td>3</td>
<td>K4</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>K1</td>
<td>2</td>
<td>b</td>
<td>4</td>
<td>K1</td>
<td>2.0</td>
</tr>
<tr>
<td>5</td>
<td>K2</td>
<td>2</td>
<td>b</td>
<td>5</td>
<td>K2</td>
<td>2.0</td>
</tr>
<tr>
<td>6</td>
<td>K4</td>
<td>2</td>
<td>b</td>
<td>6</td>
<td>K4</td>
<td>2.0</td>
</tr>
<tr>
<td>7</td>
<td>K1</td>
<td>3</td>
<td>c</td>
<td>7</td>
<td>K1</td>
<td>3.0</td>
</tr>
<tr>
<td>8</td>
<td>K2</td>
<td>3</td>
<td>c</td>
<td>8</td>
<td>K2</td>
<td>3.0</td>
</tr>
<tr>
<td>9</td>
<td>K4</td>
<td>3</td>
<td>c</td>
<td>9</td>
<td>K4</td>
<td>3.0</td>
</tr>
<tr>
<td>10</td>
<td>K1</td>
<td>NaN</td>
<td>d</td>
<td>10</td>
<td>K1</td>
<td>NaN</td>
</tr>
<tr>
<td>11</td>
<td>K2</td>
<td>4</td>
<td>d</td>
<td>11</td>
<td>K2</td>
<td>4.0</td>
</tr>
<tr>
<td>12</td>
<td>K4</td>
<td>4</td>
<td>d</td>
<td>12</td>
<td>K4</td>
<td>4.0</td>
</tr>
</tbody>
</table>

18.2. Database-style DataFrame joining/merging 537
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 [82]: df1.update(df2)

<table>
<thead>
<tr>
<th>df1</th>
<th>df2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4.6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>5.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

| 0   | 3.0 | 5.0 |
| 1   | NaN | NaN |
| 2   | 7.0 | NaN |
19.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```python
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
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:

```python
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:
In [3]: df.pivot(index='date', columns='variable', values='value')

Out[3]:

<table>
<thead>
<tr>
<th>variable</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</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>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.282863</td>
<td>1.212112</td>
<td>-1.044236</td>
<td>-0.494929</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.509059</td>
<td>-0.173215</td>
<td>-0.861849</td>
<td>1.071804</td>
</tr>
</tbody>
</table>

If the values argument is omitted, and the input DataFrame has more than one column of values which are not used as column or index inputs to pivot, then the resulting “pivoted” DataFrame will have hierarchical columns whose topmost level indicates the respective value column:

In [4]: df['value2'] = df['value'] * 2

In [5]: pivoted = df.pivot('date', 'variable')

In [6]: pivoted

Out[6]:

<table>
<thead>
<tr>
<th>value</th>
<th>value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>A</td>
</tr>
<tr>
<td>date</td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.938225</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.565727</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-3.018117</td>
</tr>
</tbody>
</table>

You of course can then select subsets from the pivoted DataFrame:

In [7]: pivoted['value2']

Out[7]:

<table>
<thead>
<tr>
<th>variable</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.938225</td>
<td>-2.271265</td>
<td>0.238417</td>
<td>-4.209138</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.565727</td>
<td>2.424224</td>
<td>-0.346429</td>
<td>2.143608</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-3.018117</td>
<td>-0.346429</td>
<td>0.238417</td>
<td>-4.209138</td>
</tr>
</tbody>
</table>

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

## 19.2 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:
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:

In [13]: stacked = df2.stack()

In [14]: stacked
Out[14]:
     first   second
    bar one     A  0.721555
           B -0.706771
           two A -1.039575
                  B  0.271860
    baz one     A -0.424972
                  B  0.567020
                  two A  0.276232
                         B -1.087401
dtype: float64

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of `stack` is `unstack`, which by default unstacks the last level:

In [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

In [16]: stacked.unstack(1)
   Out[16]:
     second
    one   two
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())
Out[22]: True
```

while the above code will raise a `TypeError` if the call to `sort` is removed.

### 19.2.1 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  A    B  A    B
animal  cat  cat  dog  dog
hair_length  long  long  short  short
0  1.075770 -0.109050  1.643563 -1.469388
1  0.357021 -0.674600 -1.776904 -0.968914
2 -1.294524  0.413738  0.276662 -0.472035
3 -0.013960 -0.362543  0.006154  0.923061

In [26]: df.stack(level=['animal', 'hair_length'])
Out[26]:
exp  A  B
animal hair_length
0  cat  long  1.075770 -0.109050
   dog  short  1.643563 -1.469388
1  cat  long  0.357021 -0.674600
   dog  short -1.776904 -0.968914
2  cat  long -1.294524  0.413738
   dog  short  0.276662 -0.472035
3  cat  long -0.013960 -0.362543
   dog  short -0.006154 -0.923061

The list of levels can contain either level names or level numbers (but not a mixture of the two).

# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [27]: df.stack(level=[1, 2])
Out[27]:
exp  A  B
animal hair_length
0  cat  long  1.075770 -0.109050
   dog  short  1.643563 -1.469388
1  cat  long  0.357021 -0.674600
   dog  short -1.776904 -0.968914
2  cat  long -1.294524  0.413738
   dog  short  0.276662 -0.472035
3  cat  long -0.013960 -0.362543
   dog  short -0.006154 -0.923061

19.2.2 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 sortlevel, of course). Here is a more complex example:

In [28]: columns = pd.MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
....:                                     ('B', 'cat'), ('A', 'dog')],
....:                                     names=['exp', 'animal'])

In [29]: index = pd.MultiIndex.from_product([('bar', 'baz', 'foo', 'qux'),
....:                                         ('one', 'two')],
....:                                         names=['first', 'second'])
As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [33]: df2.stack('exp')
Out[33]:
animal   exp
first   second
bar     one  0.895717
         A  0.805244
         B -1.206412
         A  2.565646
         B -0.226169
         A  0.805244
         B -1.206412
         A  2.565646
         B -0.226169
baz     one  0.410835
         A  0.132003
         B -0.827317
         A  0.132003
         B -0.827317
foo     one -1.413681
         A  1.607920
         B  1.024180
         A  0.569605
         B  0.569605
         A  1.607920
         B  1.024180
         A  0.569605
         B  0.569605
qux     two -1.226825
         A  0.769804
         B -1.281247
         A -0.727707
         B -0.727707
```

```
In [34]: df2.stack('animal')
Out[34]:
exp  animal
first   second
bar    one  cat
       A  0.895717
       B -1.206412
       A  2.565646
       B -0.226169
       A  2.565646
       B -0.226169
baz    one  cat
       A  0.410835
       B  0.132003
       A -0.827317
       B -0.827317
foo    one  cat
       A -1.413681
       B  1.024180
       A  0.569605
       B  0.569605
       A  1.024180
       B  1.024180
       A  0.569605
       B  0.569605
       A  1.024180
       B  1.024180
qux    two  cat
       A -1.226825
       B -1.281247
       A -0.727707
       B -0.727707
```

### 19.2.3 With a MultiIndex

Unstacking when the columns are a `MultiIndex` is also careful about doing the right thing:
In [35]: df[:3].unstack(0)
Out[35]:
exp  A    B    A
    animal cat  dog  cat  dog
    first  bar  baz  bar  baz  bar
second
one  0.895717 0.410835 0.81385 -1.206412 0.132003 2.565646
two 1.431256 NaN 1.340309 NaN -1.170299 NaN -0.226169

exp
animal
first  baz
second
one  -0.827317
two  NaN

In [36]: df2.unstack(1)
Out[36]:
exp  A    B    A
    animal cat  dog  cat  dog
    second one  two  one  two  one  two  one
first  bar    0.895717 1.431256 0.805244 1.340309 -1.206412 -1.170299 2.565646
baz    0.410835 NaN 0.813850 NaN 0.132003 NaN -0.827317
foo  -1.413681 0.875906 1.607920 -2.211372 1.024180 0.974466 0.569605
qux   NaN -1.226825 NaN 0.769804 NaN -1.281247 NaN

exp
animal
second  two
first
bar  -0.226169
baz   NaN
foo  -2.006747
qux  -0.727707

19.3 Reshaping by Melt

The melt() function is 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,

In [37]: cheese = pd.DataFrame({'first':['John', 'Mary'],
                           'last': ['Doe', 'Bo'],
                           'height': [5.5, 6.0],
                           'weight': [130, 150]})

In [38]: cheese
Out[38]:
   first  height last  weight
0     John   5.5  Doe     130
1     Mary   6.0   Bo     150
In [39]: pd.melt(cheese, id_vars=['first', 'last'])
Out[39]:
  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 [40]: pd.melt(cheese, id_vars=['first', 'last'], var_name='quantity')
Out[40]:
  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 [41]: 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 [42]: dft["id"] = dft.index

In [43]: dft
Out[43]:
0     a      d    2.5    3.2 -0.121306     0
1     b      e    1.2    1.3 -0.097883     1
2     c      f    0.7    0.1   0.695775     2

In [44]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")
Out[44]:
      X  A  B
id year
0  1970 -0.121306  a  2.5
1  1970 -0.097883  b  1.2
2  1970  0.695775  c  0.7
0  1980 -0.121306  d  3.2
1  1980 -0.097883  e  1.3
2  1980  0.695775  f  0.1

19.4 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 [45]: df
Out[45]:
   exp  A  B  A
animal  cat  dog  cat  dog
first  second

546  Chapter 19.  Reshaping and Pivot Tables
In [46]: `df.stack().mean(1).unstack()
Out[46]:

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 [47]: `df.groupby(level=1, axis=1).mean()
Out[47]:

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

In [48]: `df.stack().groupby(level=1).mean()
Out[48]:

exp  A  B
second
one  0.071448  0.455513
    two -0.424186 -0.204486

In [49]: `df.mean().unstack(0)
Out[49]:

exp  A  B
animal
cat  0.060843  0.018596
dog -0.413580  0.232430

### 19.5 Pivot tables and cross-tabulations

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 [50]: import datetime

In [51]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 6,
                      'B': ['A', 'B', 'C'] * 8,
                      'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
                      'D': np.random.randn(24),
                      'E': np.random.randn(24),
                      'F': [datetime.datetime(2013, i, 1)
                            for i in range(1, 13)] +
                            [datetime.datetime(2013, i, 15)
                             for i in range(1, 13)]})

In [52]: df
Out[52]:
   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  0.169796  -0.487602  2013-04-01
4   one B  bar  0.149748  -0.082240  2013-05-01
5   one C  bar  0.732339  -2.189373  2013-06-01
6   two A  foo  0.687738   0.380396  2013-07-01
7   ...   ...   ...          ...          ...          ...
17  one C  bar  0.345352  0.206053  2013-06-15
18  two A  foo  1.314232  -0.251905  2013-07-15
19  three B  foo  0.690579  -2.213588  2013-08-15
20   one C  foo  0.995761  1.063327  2013-09-15
21   one A  bar  2.396780  1.266143  2013-10-15
22  two B  bar  0.014871  0.299368  2013-11-15
23  three C  bar  3.357427  0.863838  2013-12-15
[24 rows x 6 columns]
```

We can produce pivot tables from this data very easily:

```
In [53]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[53]:
   C       bar       foo
A
one A    1.120915 -0.514058
   B    -0.338421  0.002759
   C    -0.538846  0.699535
three A  -1.181568   NaN
   B     NaN  0.433512
   C  0.588783   NaN
two A     NaN  1.000985
   B  0.158248   NaN
   C NaN  0.176180
```

548 Chapter 19. Reshaping and Pivot Tables
In [54]: pd.pivot_table(df, values='D', index=['B'], columns=['A', 'C'], aggfunc=np.sum)
Out[54]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>one</th>
<th>three</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2.241830</td>
<td>-1.028115</td>
<td>-2.363137</td>
<td>NaN</td>
</tr>
<tr>
<td>B</td>
<td>-0.676843</td>
<td>0.005518</td>
<td>NaN</td>
<td>0.867024</td>
</tr>
<tr>
<td>C</td>
<td>-1.077692</td>
<td>1.399070</td>
<td>1.177566</td>
<td>NaN</td>
</tr>
</tbody>
</table>

In [55]: pd.pivot_table(df, values=['D','E'], index=['B'], columns=['A', 'C'], aggfunc=np.sum)
Out[55]:

<table>
<thead>
<tr>
<th></th>
<th>D E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></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 [56]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])
Out[56]:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></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.

In [57]: pd.pivot_table(df, values='D', index=Grouper(freq='M', key='F'), columns='C')
Out[57]:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

19.5. Pivot tables and cross-tabulations
You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```python
In [58]: table = pd.pivot_table(df, index=['A', 'B'], columns=['C'])
In [59]: print(table.to_string(na_rep=''))
```

```
<table>
<thead>
<tr>
<th></th>
<th>bar</th>
<th>foo</th>
<th></th>
<th>bar</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.120915</td>
<td>-0.514058</td>
<td>B</td>
<td>0.002759</td>
<td>0.684140</td>
</tr>
<tr>
<td></td>
<td>1.393057</td>
<td>-0.021605</td>
<td></td>
<td>0.908442</td>
<td>0.747859</td>
</tr>
<tr>
<td>C</td>
<td>-0.538846</td>
<td>0.699535</td>
<td></td>
<td>0.699535</td>
<td>0.747859</td>
</tr>
<tr>
<td>three</td>
<td>-1.181568</td>
<td>0.961289</td>
<td></td>
<td>0.699535</td>
<td>0.747859</td>
</tr>
<tr>
<td></td>
<td>0.433512</td>
<td>-1.064372</td>
<td></td>
<td>-0.131830</td>
<td></td>
</tr>
<tr>
<td>two</td>
<td>0.158248</td>
<td>-0.097147</td>
<td></td>
<td>0.176180</td>
<td>0.436241</td>
</tr>
<tr>
<td></td>
<td>0.588783</td>
<td>0.588783</td>
<td></td>
<td>0.176180</td>
<td>0.436241</td>
</tr>
</tbody>
</table>
```

Note that `pivot_table` is also available as an instance method on DataFrame.

### 19.5.1 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)

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

For example:

```python
In [60]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'
In [61]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)
In [62]: b = np.array([one, one, two, one, two, one], dtype=object)
In [63]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)
```
In [64]: pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
Out[64]:
   b    one two
  c    dull  shiny  dull  shiny
     a  
    bar  1   0   0   1
    foo  2   1   1   0

19.5.2 Adding margins (partial aggregates)

If you pass margins=True to pivot_table, special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

In [65]: df.pivot_table(index=['A', 'B'], columns='C', margins=True, aggfunc=np.std)
Out[65]:
   D  E
  C    bar    foo  All   bar    foo  All
  A  B
    one
      A  1.804346  1.210272  1.569879  0.179483  0.418374  0.858005
      B  0.690376  1.353355  0.898998  1.083825  0.968138  1.101401
      C  0.273641  0.418926  0.771139  1.689271  0.446140  1.422136
    three A  0.794212  1.049040  1.422136
      B  NaN  0.690376  1.353355  0.898998  1.083825  0.968138  1.101401
      C  3.915454  1.035215  1.625237  1.625237
    two A  NaN   0.202765  0.560757  0.202765   0.560757
      B  0.202765  0.560757  NaN   0.202765  0.560757
      C  NaN   0.650439  0.650439  NaN   0.650439  0.650439
    All  1.556686  0.952552  1.246608  1.250924  0.899904  1.059389

19.6 Tiling

The cut function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

In [66]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [67]: pd.cut(ages, bins=3)
Out[67]:

[(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[

Categories (3, object): [(9.95, 26.667] < (26.667, 43.333] < (43.333, 60[

If the bins keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

In [68]: pd.cut(ages, bins=[0, 18, 35, 70])
Out[68]:

[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]

Categories (3, object): [(0, 18] < (18, 35] < (35, 70]

19.7 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:
In [69]: df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})

In [70]: pd.get_dummies(df['key'])
Out[70]:
    a  b  c
0  0  1  0
1  1  0  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:

In [71]: dummies = pd.get_dummies(df['key'], prefix='key')

In [72]: dummies
Out[72]:
    key_a  key_b  key_c
0     0     1     0
1     1     0     0
2     1     0     0
3     0     0     1
4     1     0     0
5     0     1     0

In [73]: df[['data1']].join(dummies)
Out[73]:
     data1  key_a  key_b  key_c
0     0.0     0     1     0
1     1.0     0     1     0
2     2.0     1     0     0
3     3.0     0     0     1
4     4.0     1     0     0
5     5.0     0     1     0

This function is often used along with discretization functions like \texttt{cut}:

In [74]: values = np.random.randn(10)

In [75]: values
Out[75]:
array([-1.0481, -0.9884, 0.0941, 1.2627, 1.29 , 0.0824, -0.0558, 0.5366])

In [76]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [77]: pd.get_dummies(pd.cut(values, bins))
Out[77]:
    (0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1]
0      0      0      1      0      0
1      0      0      0      0      0
2      0      0      0      0      0
3      0      0      0      0      0
4      1      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      1      0      0      0      0
8      0      0      0      0      0
See also `Series.str.get_dummies`.

New in version 0.15.0.

`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 [78]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
                      'C': [1, 2, 3]})

In [79]: pd.get_dummies(df)
Out[79]:
   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 [80]: pd.get_dummies(df, columns=['A'])
Out[80]:
    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 [81]: simple = pd.get_dummies(df, prefix='new_prefix')

In [82]: simple
Out[82]:
   C  new_prefix_a  new_prefix_b  new_prefix_b  new_prefix_c
0  1                0                0                1
1  2                0                1                0
2  3                1                0                0
```

```python
In [83]: from_list = pd.get_dummies(df, prefix=['from_A', 'from_B'])

In [84]: from_list
Out[84]:
    C  from_A_a  from_A_b  from_B_b  from_B_c
0  1                1                0                0    1
1  2                0                1                0    1
2  3                1                0                1    0
```

```python
In [85]: from_dict = pd.get_dummies(df, prefix={'B': 'from_B', 'A': 'from_A'})
```
In [86]: from_dict
Out[86]:
     C  from_A_a  from_A_b  from_B_b  from_B_c
0   1          1          0          0         1
1   2          0          1          0         1
2   3          1          0          1         0

19.8 Factorizing values

To encode 1-d values as an enumerated type use factorize:

In [87]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])

In [88]: x
Out[88]:
0   A
1   A
2  NaN
3   B
4  3.14
5   inf
dtype: object

In [89]: labels, uniques = pd.factorize(x)

In [90]: labels
Out[90]: array([ 0,  0, -1,  1,  2,  3])

In [91]: uniques
Out[91]: Index([u'A', u'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 [92]: pd.factorize(x, sort=True)
Out[92]: (array([ 2,  2, -1,  3,  0,  1]),
         Index([3.14, inf, u'A', u'B'], dtype='object'))

In [93]: np.unique(x, return_inverse=True)[:-1]
Out[93]: (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. This feature was introduced in version 0.15.
pandas has proven very successful as a tool for working with time series data, especially in the financial data analysis space. With the 0.8 release, we have further improved the time series API in pandas by leaps and bounds. Using the new NumPy `datetime64` dtype, 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 = 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', tz=None)
```

Index pandas objects with dates:

```python
In [3]: ts = Series(randn(len(rng)), index=rng)
```

```python
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', how='mean')
Out[7]:
2011-01-01 -0.319569
2011-01-02 -0.337703
2011-01-03  0.117258
Freq: D, dtype: float64

20.1 Time Stamps vs. Time Spans

Time-stamped data is the most basic type of timeseries data that associates values with points in time. For pandas objects it means using the points in time to create the index.

In [8]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]

In [9]: ts = Series(np.random.randn(3), dates)

In [10]: type(ts.index)
Out[10]: pandas.tseries.index.DatetimeIndex

In [11]: ts
Out[11]:
2012-05-01 -0.410001
2012-05-02 -0.078638
2012-05-03  0.545952
dtype: float64

However, in many cases it is more natural to associate things like change variables with a time span instead.

For example:

In [12]: periods = PeriodIndex([Period('2012-01'), Period('2012-02'),
                           Period('2012-03')])

In [13]: ts = Series(np.random.randn(3), periods)

In [14]: type(ts.index)
Out[14]: pandas.tseries.period.PeriodIndex

In [15]: ts
Out[15]:
2012-01  -1.219217
2012-02  -1.226825
2012-03   0.769804
Freq: M, dtype: float64
Starting with 0.8, 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 forthcoming in future releases.

## 20.2 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 [16]: to_datetime(Series(['Jul 31, 2009', '2010-01-10', None]))
Out[16]:
0  2009-07-31
1  2010-01-10
2  NaT
   dtype: datetime64[ns]
```

```python
In [17]: to_datetime(['2005/11/23', '2010.12.31'])
Out[17]: DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None, tz=None)
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```python
In [18]: to_datetime(['04-01-2012 10:00'], dayfirst=True)
Out[18]: DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=None, tz=None)
```

```python
In [19]: to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
Out[19]: DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]', freq=None, tz=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.

**Note:** Specifying a format argument will potentially speed up the conversion considerably and on versions later than 0.13.0 explicitly specifying a format string of ‘%Y%m%d’ takes a faster path still.

### 20.2.1 Invalid Data

Pass `coerce=True` to convert invalid data to NaT (not a time):

```python
In [20]: to_datetime(['2009-07-31', 'asd'], coerce=True)
Out[20]: array(['2009-07-31', 'NaT'], dtype=object)
```

```python
In [21]: to_datetime(['2009-07-31', 'asd'], coerce=True)
Out[21]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None, tz=None)
```

Take care, `to_datetime` may not act as you expect on mixed data:

```python
In [22]: to_datetime([1, '1'])
Out[22]: array([1, '1'], dtype=object)
```
20.2.2 Epoch Timestamps

It's also possible to convert integer or float epoch times. The default unit for these is nanoseconds (since these are how Timestamps are stored). However, often epochs are stored in another unit which can be specified:

Typical epoch stored units

```python
In [23]: to_datetime([1349720105, 1349806505, 1349892905,
                  ....: 1349979305, 1350065705], unit='s')
Out[23]: 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, tz=None)

In [24]: to_datetime([1349720105100, 1349720105200, 1349720105300,
                  ....: 1349720105400, 1349720105500], unit='ms')
Out[24]: 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, tz=None)
```

These work, but the results may be unexpected.

```python
In [25]: to_datetime([1])
Out[25]: DatetimeIndex(['1970-01-01 00:00:00.000000001'], dtype='datetime64[ns]', freq=None, tz=None)

In [26]: to_datetime([1, 3.14], unit='s')
Out[26]: DatetimeIndex(['1970-01-01 00:00:01', '1970-01-01 00:00:03.140000'], dtype='datetime64[ns]',
```

Note: Epoch times will be rounded to the nearest nanosecond.

20.3 Generating Ranges of Timestamps

To generate an index with time stamps, you can use either the DatetimeIndex or Index constructor and pass in a list of datetime objects:

```python
In [27]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2),
                  datetime(2012, 5, 3)]

In [28]: index = DatetimeIndex(dates)

In [29]: index # Note the frequency information
Out[29]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'],
```

Practically, 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 pandas functions `date_range` and `bdate_range` to create timestamp indexes.
In [32]: index = date_range('2000-1-1', periods=1000, freq='M')

In [33]: index
Out[33]:
dtype='datetime64[ns]', length=1000, freq='M', tz=None)

In [34]: index = bdate_range('2012-1-1', periods=250)

In [35]: index
Out[35]:
   '2012-12-03', '2012-12-04', '2012-12-05', '2012-12-06', '2012-12-07', '2012-12-10', '2012-12-11', '2012-12-12', '2012-12-13', '2012-12-14'],
dtype='datetime64[ns]', length=250, freq='B', tz=None)

Convenience functions like date_range and bdate_range utilize a variety of frequency aliases. The default frequency for date_range is a calendar day while the default for bdate_range is a business day.

In [36]: start = datetime(2011, 1, 1)

In [37]: end = datetime(2012, 1, 1)

In [38]: rng = date_range(start, end)

In [39]: rng
Out[39]:
dtype='datetime64[ns]', length=366, freq='D', tz=None)

In [40]: rng = bdate_range(start, end)

In [41]: rng
Out[41]:
dtype='datetime64[ns]', length=260, freq='B', tz=None)
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:

```python
In [42]: date_range(start, end, freq='BM')
Out[42]:
       dtype='datetime64[ns]', freq='BM', tz=None)
```

```python
In [43]: date_range(start, end, freq='W')
Out[43]:
              '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', tz=None)
```

```python
In [44]: bdate_range(end=end, periods=20)
Out[44]:
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', tz=None)
```

```python
In [45]: bdate_range(start=start, periods=20)
Out[45]:
              '2011-01-27', '2011-01-28'],
       dtype='datetime64[ns]', freq='B', tz=None)
```

The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

## 20.4 DatetimeIndex

One of the main uses for DatetimeIndex is as an index for pandas objects. The DatetimeIndex class contains many timeseries 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

DatetimIndex objects has all the basic functionality of regular Index objects and a smorgasbord of advanced timeseries-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. So please be careful.

DatetimIndex can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```python
In [46]: rng = date_range(start, end, freq='BM')
In [47]: ts = Series(randn(len(rng)), index=rng)
In [48]: ts.index
                        dtype='datetime64[ns]', freq='BM', tz=None)
In [49]: ts[:5].index
                    '2011-05-31'],
                        dtype='datetime64[ns]', freq='BM', tz=None)
In [50]: ts[::2].index
                     '2011-09-30', '2011-11-30'],
                        dtype='datetime64[ns]', freq='2BM', tz=None)
```

### 20.4.1 DatetimIndex Partial String Indexing

You can pass in dates and strings that parse to dates as indexing parameters:

```python
In [51]: ts['1/31/2011']
Out[51]: -1.2812473076599529
In [52]: ts[datetime(2011, 12, 25):]
Out[52]: 2011-12-30  0.687738
Freq: BM, dtype: float64
In [53]: ts['10/31/2011':'12/31/2011']
Out[53]: 2011-10-31  0.149748
```

20.4. DatetimIndex
To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

```
In [54]: ts['2011']
Out[54]:
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 [55]: ts['2011-6']
Out[55]:
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. Here’s an example:

```
In [56]: dft = DataFrame(randn(100000,1),columns=['A'],index=date_range('20130101',periods=100000,freq='T'))

In [57]: dft
Out[57]:
         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
      ...        ...
2013-03-11 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]
```

```
In [58]: dft['2013']
Out[58]:
         A
2013-01-01 00:00:00  0.176444
2013-01-01 00:01:00  0.403310
```

This starts on the very first time in the month, and includes the last date & time for the month

```
In [59]: dft['2013-1':'2013-2']
Out[59]:
          0
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
...           ...
2013-02-28 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 [60]: dft['2013-1':'2013-2-28']
Out[60]:
          0
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
...           ...
2013-02-28 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
[100000 rows x 1 columns]
```
This specifies an exact stop time (and is not the same as the above)

```
In [61]: dft['2013-1':'2013-2-28 00:00:00']
Out[61]:

       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

```
In [62]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[62]:

       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]
```
Warning: The following selection will raise a KeyError; otherwise this selection methodology would be inconsistent with other selection methods in pandas (as this is not a slice, nor does it resolve to one)

dft['2013-1-15 12:30:00']

To select a single row, use .loc

In [63]: dft.loc['2013-1-15 12:30:00']
Out[63]:
   A
0  0.193284
Name: 2013-01-15 12:30:00, dtype: float64

20.4.2 Datetime Indexing

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 frequency of the index. In contrast, indexing with datetime objects is exact, because the objects have exact meaning. These also follow the semantics of including both endpoints.

These datetime objects are specific hours, minutes, and seconds even though they were not explicitly specified (they are 0).

In [64]: dft[datetime(2013, 1, 1):datetime(2013, 2, 28)]
Out[64]:
   A
0  0.176444
1  0.403310
2 -0.154951
3  0.301624
4 -2.179861
5 -1.369849
6 -0.954208
... ... 
83519 0.897051
83520 -0.309230
83521 1.944713
83522 0.369849
83523 0.053071
83524 -0.019734
83525 1.388189

[83521 rows x 1 columns]

With no defaults.

In [65]: dft[datetime(2013, 1, 1, 10, 12, 0):datetime(2013, 2, 28, 10, 12, 0)]
Out[65]:
   A
0  -0.246733
1  -1.429225
2  -1.265339
3   0.710986
4  -0.818200
5   0.543542
6   1.577713
... ... 
83519 0.897051
83520 -0.309230
83521 1.944713
83522 0.369849
83523 0.053071
83524 -0.019734
83525 1.388189

20.4. DatetimeIndex
20.4.3 Truncating & Fancy Indexing

A `truncate` convenience function is provided that is equivalent to slicing:

```python
In [66]: ts.truncate(before='10/31/2011', after='12/31/2011')
Out[66]:
2011-10-31  0.149748
2011-11-30 -0.732339
2011-12-30  0.687738
Freq: BM, dtype: float64
```

Even complicated fancy indexing that breaks the DatetimeIndex’s frequency regularity will result in a DatetimeIndex (but frequency is lost):

```python
In [67]: ts[[0, 2, 6]].index
Out[67]: DatetimeIndex(['2011-01-31', '2011-03-31', '2011-07-29'], dtype='datetime64[ns]', freq=None, tz=None)
```

20.4.4 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</td>
</tr>
<tr>
<td>time</td>
<td>Returns datetime.time</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 day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>weekday</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>quarter</td>
<td>Quarter of the date: Jan=Mar = 1, Apr-Jun = 2, etc.</td>
</tr>
<tr>
<td>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>
</tbody>
</table>
Furthermore, if you have a Series with datetimelike values, then you can access these properties via the .dt accessor, see the docs

20.5 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 (experimental)</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>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>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 [68]: d = datetime(2008, 8, 18, 9, 0)
In [69]: d + relativedelta(months=4, days=5)
Out[69]: datetime.datetime(2008, 12, 23, 9, 0)
```

We could have done the same thing with DateOffset:

```
In [70]: from pandas.tseries.offsets import *
In [71]: d + Offset(months=4, days=5)
Out[71]: Timestamp('2008-12-23 09:00:00')
```
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):
        ...
```

```
In [72]: d - 5 * BDay()
Out[72]: Timestamp('2008-08-11 09:00:00')

In [73]: d + BMonthEnd()
Out[73]: Timestamp('2008-08-29 09:00:00')
```

The `rollforward` and `rollback` methods do exactly what you would expect:

```
In [74]: d
Out[74]: datetime.datetime(2008, 8, 18, 9, 0)

In [75]: offset = BMonthEnd()

In [76]: offset.rollforward(d)
Out[76]: Timestamp('2008-08-29 09:00:00')

In [77]: offset.rollback(d)
Out[77]: Timestamp('2008-07-31 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.

```
In [78]: day = Day()

In [79]: day.apply(Timestamp('2014-01-01 09:00'))
Out[79]: Timestamp('2014-01-02 09:00:00')

In [80]: day = Day(normalize=True)

In [81]: day.apply(Timestamp('2014-01-01 09:00'))
Out[81]: Timestamp('2014-01-02 00:00:00')

In [82]: hour = Hour()

In [83]: hour.apply(Timestamp('2014-01-01 22:00'))
Out[83]: Timestamp('2014-01-01 23:00:00')

In [84]: hour = Hour(normalize=True)

In [85]: hour.apply(Timestamp('2014-01-01 22:00'))
Out[85]: Timestamp('2014-01-01 00:00:00')
```
20.5.1 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:

```python
In [87]: d
Out[87]: datetime.datetime(2008, 8, 18, 9, 0)

In [88]: d + Week()
Out[88]: Timestamp('2008-08-25 09:00:00')

In [89]: d + Week(weekday=4)
Out[89]: Timestamp('2008-08-22 09:00:00')

In [90]: (d + Week(weekday=4)).weekday()
Out[90]: 4

In [91]: d - Week()
Out[91]: Timestamp('2008-08-11 09:00:00')
```

The normalize option will be effective for addition and subtraction.

```python
In [92]: d + Week(normalize=True)
Out[92]: Timestamp('2008-08-25 00:00:00')

In [93]: d - Week(normalize=True)
Out[93]: Timestamp('2008-08-11 00:00:00')
```

Another example is parameterizing YearEnd with the specific ending month:

```python
In [94]: d + YearEnd()
Out[94]: Timestamp('2008-12-31 09:00:00')

In [95]: d + YearEnd(month=6)
Out[95]: Timestamp('2009-06-30 09:00:00')
```

20.5.2 Custom Business Days (Experimental)

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.

```python
In [96]: from pandas.tseries.offsets import CustomBusinessDay

# As an interesting example, let's look at Egypt where
# a Friday–Saturday weekend is observed.
In [97]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers’ Day so let’s
# add that for a couple of years
In [98]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]
```
In [99]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [100]: dt = datetime(2013, 4, 30)

In [101]: dt + 2 * bday_egypt
Out[101]: Timestamp('2013-05-05 00:00:00')

In [102]: dts = date_range(dt, periods=5, freq=bday_egypt)

In [103]: Series(dts.weekday, dts).map(Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
Out[103]:
2013-04-30 Tue
2013-05-02 Thu
2013-05-05 Sun
2013-05-06 Mon
2013-05-07 Tue
Freq: C, dtype: object

As of v0.14 holiday calendars can be used to provide the list of holidays. See the holiday calendar section for more information.

In [104]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [105]: bday_us = CustomBusinessDay(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [106]: dt = datetime(2014, 1, 17)

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [107]: dt + bday_us
Out[107]: Timestamp('2014-01-21 00:00:00')

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.

In [108]: from pandas.tseries.offsets import CustomBusinessMonthBegin

In [109]: bmth_us = CustomBusinessMonthBegin(calendar=USFederalHolidayCalendar())

# Skip new years
In [110]: dt = datetime(2013, 12, 17)

In [111]: dt + bmth_us
Out[111]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [112]: from pandas import DatetimeIndex

In [113]: DatetimeIndex(start='20100101', end='20120101', freq=bmth_us)
Out[113]:
DatetimeIndex(['2010-01-04', '2010-02-01', '2010-03-01', '2010-04-01',
               '2010-05-03', '2010-06-01', '2010-07-01', '2010-08-02',
               '2010-09-01', '2010-10-01', '2010-11-01', '2010-12-01',
               '2011-01-03', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-09-01', '2011-10-03', '2011-11-01', '2011-12-01'],
              dtype='datetime64[ns]', freq='CBMS', tz=None)

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.

**Note:** This uses the `numpy.busdaycalendar` API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.

**Warning:** There are known problems with the timezone handling in Numpy 1.7 and users should therefore use this experimental(!) feature with caution and at their own risk. To the extent that the `datetime64` and `busdaycalendar` APIs in Numpy have to change to fix the timezone issues, the behaviour of the `CustomBusinessDay` class may have to change in future versions.

### 20.5.3 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 [114]: bh = BusinessHour()

In [115]: bh
Out[115]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [116]: Timestamp('2014-08-01 10:00').weekday()
Out[116]: 4

In [117]: Timestamp('2014-08-01 10:00') + bh
Out[117]: Timestamp('2014-08-01 11:00:00')

# Below example is the same as Timestamp('2014-08-01 09:00') + bh
In [118]: Timestamp('2014-08-01 08:00') + bh
Out[118]: Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
In [119]: Timestamp('2014-08-01 16:00') + bh
Out[119]: Timestamp('2014-08-04 09:00:00')

# Remainings are added to the next day
In [120]: Timestamp('2014-08-01 16:30') + bh
Out[120]: Timestamp('2014-08-04 09:30:00')

# Adding 2 business hours
In [121]: Timestamp('2014-08-01 10:00') + BusinessHour(2)
Out[121]: Timestamp('2014-08-01 12:00:00')

# Subtracting 3 business hours
In [122]: Timestamp('2014-08-01 10:00') + BusinessHour(-3)
Out[122]: Timestamp('2014-07-31 15:00:00')
```

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`.

---

#### 20.5. DateOffset objects
In [123]: bh = BusinessHour(start='11:00', end=time(20, 0))

In [124]: bh
Out[124]: <BusinessHour: BH=11:00-20:00>

In [125]: Timestamp('2014-08-01 13:00') + bh
Out[125]: Timestamp('2014-08-01 14:00:00')

In [126]: Timestamp('2014-08-01 09:00') + bh
Out[126]: Timestamp('2014-08-01 12:00:00')

In [127]: Timestamp('2014-08-01 18:00') + bh
Out[127]: Timestamp('2014-08-01 19:00:00')

Passing start time later than end represents midnight business hour. In this case, business hour exceeds midnight and overlap to the next day. Valid business hours are distinguished by whether it started from valid BusinessDay.

In [128]: bh = BusinessHour(start='17:00', end='09:00')

In [129]: bh
Out[129]: <BusinessHour: BH=17:00-09:00>

In [130]: Timestamp('2014-08-01 17:00') + bh
Out[130]: Timestamp('2014-08-01 18:00:00')

In [131]: Timestamp('2014-08-01 23:00') + bh
Out[131]: Timestamp('2014-08-02 00:00:00')

# Although 2014-08-02 is Saturday,
# it is valid because it starts from 08-01 (Friday).
In [132]: Timestamp('2014-08-02 04:00') + bh
Out[132]: Timestamp('2014-08-02 05:00:00')

# Although 2014-08-04 is Monday,
# it is out of business hours because it starts from 08-03 (Sunday).
In [133]: Timestamp('2014-08-04 04:00') + bh
Out[133]: Timestamp('2014-08-04 18:00:00')

Applying BusinessHour.rollforward and rollback to out of business hours results in the next business hour start or previous day’s end. Different from other offsets, BusinessHour.rollforward may output different results from apply by definition.

This is because one day’s business hour end is equal to next day’s business hour start. For example, under the default business hours (9:00 - 17:00), there is no gap (0 minutes) between 2014-08-01 17:00 and 2014-08-04 09:00.

# This adjusts a Timestamp to business hour edge
In [134]: BusinessHour().rollback(Timestamp('2014-08-02 15:00'))
Out[134]: Timestamp('2014-08-01 17:00:00')

In [135]: BusinessHour().rollforward(Timestamp('2014-08-02 15:00'))
Out[135]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessHour().apply(Timestamp('2014-08-01 17:00')).
# And it is the same as BusinessHour().apply(Timestamp('2014-08-04 09:00'))
In [136]: BusinessHour().apply(Timestamp('2014-08-02 15:00'))
Out[136]: Timestamp('2014-08-04 10:00:00')

# BusinessDay results (for reference)
In [137]: BusinessHour().rollforward(Timestamp('2014-08-02'))
Out[137]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessDay().apply(Timestamp('2014-08-01'))
# The result is the same as rollforward because BusinessDay never overlap.
In [138]: BusinessHour().apply(Timestamp('2014-08-02'))
Out[138]: Timestamp('2014-08-04 10:00:00')

20.5.4 Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as offset aliases (referred to as time rules prior to v0.8.0).

<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 (experimental)</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>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>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 endfrequency</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</td>
<td>year end frequency</td>
</tr>
<tr>
<td>BA</td>
<td>business year end frequency</td>
</tr>
<tr>
<td>AS</td>
<td>year start frequency</td>
</tr>
<tr>
<td>BAS</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</td>
<td>minutely frequency</td>
</tr>
<tr>
<td>S</td>
<td>secondly frequency</td>
</tr>
<tr>
<td>L</td>
<td>milliseonds</td>
</tr>
<tr>
<td>U</td>
<td>microseconds</td>
</tr>
<tr>
<td>N</td>
<td>nanoseconds</td>
</tr>
</tbody>
</table>

20.5.5 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

In [139]: date_range(start, periods=5, freq='B')
Out[139]:
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B', tz=None)

In [140]: date_range(start, periods=5, freq=BDay())
Out[140]:
               ...:  ...: '2011-01-07'],
              dtype='datetime64[ns]', freq='B', tz=None)
You can combine together day and intraday offsets:

In [141]: date_range(start, periods=10, freq='2h20min')
Out[141]:
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', tz=None)

In [142]: date_range(start, periods=10, freq='1D10U')
Out[142]:
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-02 00:00:00.000010',
              '2011-01-03 00:00:00.000020', '2011-01-04 00:00:00.000030',
              '2011-01-05 00:00:00.000040', '2011-01-06 00:00:00.000050',
              '2011-01-07 00:00:00.000060', '2011-01-08 00:00:00.000070',
              '2011-01-09 00:00:00.000080', '2011-01-10 00:00:00.000090'],
dtype='datetime64[ns]', freq='86400000010U', tz=None)

## 20.5.6 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W-SUN</td>
<td>weekly frequency (sundays). Same as ‘W’</td>
</tr>
<tr>
<td>W-MON</td>
<td>weekly frequency (mondays)</td>
</tr>
<tr>
<td>W-TUE</td>
<td>weekly frequency (tuesdays)</td>
</tr>
<tr>
<td>W-WED</td>
<td>weekly frequency (wednesdays)</td>
</tr>
<tr>
<td>W-THU</td>
<td>weekly frequency (thursdays)</td>
</tr>
<tr>
<td>W-FRI</td>
<td>weekly frequency (fridays)</td>
</tr>
<tr>
<td>W-SAT</td>
<td>weekly frequency (saturdays)</td>
</tr>
<tr>
<td>(B)Q(S)-DEC</td>
<td>quarterly frequency, year ends in December. Same as ‘Q’</td>
</tr>
<tr>
<td>(B)Q(S)-JAN</td>
<td>quarterly frequency, year ends in January</td>
</tr>
<tr>
<td>(B)Q(S)-FEB</td>
<td>quarterly frequency, year ends in February</td>
</tr>
<tr>
<td>(B)Q(S)-MAR</td>
<td>quarterly frequency, year ends in March</td>
</tr>
<tr>
<td>(B)Q(S)-APR</td>
<td>quarterly frequency, year ends in April</td>
</tr>
<tr>
<td>(B)Q(S)-MAY</td>
<td>quarterly frequency, year ends in May</td>
</tr>
<tr>
<td>(B)Q(S)-JUN</td>
<td>quarterly frequency, year ends in June</td>
</tr>
<tr>
<td>(B)Q(S)-JUL</td>
<td>quarterly frequency, year ends in July</td>
</tr>
<tr>
<td>(B)Q(S)-AUG</td>
<td>quarterly frequency, year ends in August</td>
</tr>
<tr>
<td>(B)Q(S)-SEP</td>
<td>quarterly frequency, year ends in September</td>
</tr>
<tr>
<td>(B)Q(S)-OCT</td>
<td>quarterly frequency, year ends in October</td>
</tr>
<tr>
<td>(B)Q(S)-NOV</td>
<td>quarterly frequency, year ends in November</td>
</tr>
<tr>
<td>(B)A(S)-DEC</td>
<td>annual frequency, anchored end of December. Same as ‘A’</td>
</tr>
<tr>
<td>(B)A(S)-JAN</td>
<td>annual frequency, anchored end of January</td>
</tr>
<tr>
<td>(B)A(S)-FEB</td>
<td>annual frequency, anchored end of February</td>
</tr>
<tr>
<td>(B)A(S)-MAR</td>
<td>annual frequency, anchored end of March</td>
</tr>
<tr>
<td>(B)A(S)-APR</td>
<td>annual frequency, anchored end of April</td>
</tr>
<tr>
<td>(B)A(S)-MAY</td>
<td>annual frequency, anchored end of May</td>
</tr>
<tr>
<td>(B)A(S)-JUN</td>
<td>annual frequency, anchored end of June</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.1 – continued from previous page

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.

### 20.5.7 Legacy Aliases

Note that prior to v0.8.0, time rules had a slightly different look. pandas will continue to support the legacy time rules for the time being but it is strongly recommended that you switch to using the new offset aliases.

<table>
<thead>
<tr>
<th>Legacy Time Rule</th>
<th>Offset Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEKDAY</td>
<td>B</td>
</tr>
<tr>
<td>EOM</td>
<td>BM</td>
</tr>
<tr>
<td>W@MON</td>
<td>W-MON</td>
</tr>
<tr>
<td>W@TUE</td>
<td>W-TUE</td>
</tr>
<tr>
<td>W@WED</td>
<td>W-WED</td>
</tr>
<tr>
<td>W@THU</td>
<td>W-THU</td>
</tr>
<tr>
<td>W@FRI</td>
<td>W-FRI</td>
</tr>
<tr>
<td>W@SAT</td>
<td>W-SAT</td>
</tr>
<tr>
<td>W@SUN</td>
<td>W-SUN</td>
</tr>
<tr>
<td>Q@JAN</td>
<td>BQ-JAN</td>
</tr>
<tr>
<td>Q@FEB</td>
<td>BQ-FEB</td>
</tr>
<tr>
<td>Q@MAR</td>
<td>BQ-MAR</td>
</tr>
<tr>
<td>A@JAN</td>
<td>BA-JAN</td>
</tr>
<tr>
<td>A@FEB</td>
<td>BA-FEB</td>
</tr>
<tr>
<td>A@MAR</td>
<td>BA-MAR</td>
</tr>
<tr>
<td>A@APR</td>
<td>BA-APR</td>
</tr>
<tr>
<td>A@MAY</td>
<td>BA-MAY</td>
</tr>
<tr>
<td>A@JUN</td>
<td>BA-JUN</td>
</tr>
<tr>
<td>A@JUL</td>
<td>BA-JUL</td>
</tr>
<tr>
<td>A@AUG</td>
<td>BA-AUG</td>
</tr>
<tr>
<td>A@SEP</td>
<td>BA-SEP</td>
</tr>
<tr>
<td>A@OCT</td>
<td>BA-OCT</td>
</tr>
<tr>
<td>A@NOV</td>
<td>BA-NOV</td>
</tr>
<tr>
<td>A@DEC</td>
<td>BA-DEC</td>
</tr>
<tr>
<td>min</td>
<td>T</td>
</tr>
<tr>
<td>ms</td>
<td>L</td>
</tr>
<tr>
<td>us</td>
<td>U</td>
</tr>
</tbody>
</table>

As you can see, legacy quarterly and annual frequencies are business quarters and business year ends. Please also note the legacy time rule for milliseconds `ms` versus the new offset alias for month start `MS`. This means that offset alias parsing is case sensitive.

### 20.5.8 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, \texttt{start\_date} and \texttt{end\_date} class attributes determine over what date range holidays are generated. These should be overwritten on the \texttt{AbstractHolidayCalendar} class to have the range apply to all calendar subclasses. \texttt{USFederalHolidayCalendar} is the only calendar that exists and primarily serves as an example for developing other calendars.

For holidays that occur on fixed dates (e.g., US Memorial Day or July 4th) an observance rule determines when that holiday is observed if it falls on a weekend or some other non-observed day. Defined observance rules are:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nearest_workday</td>
<td>move Saturday to Friday and Sunday to Monday</td>
</tr>
<tr>
<td>sunday_to_monday</td>
<td>move Sunday to following Monday</td>
</tr>
<tr>
<td>next_monday_or_tuesday</td>
<td>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:

In [143]: from pandas.tseries.holiday import Holiday, USMemorialDay, AbstractHolidayCalendar, nearest_workday, MO

In [144]: 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 [145]: cal = ExampleCalendar()

In [146]: cal.holidays(datetime(2012, 1, 1), datetime(2012, 12, 31))
Out[146]: DatetimeIndex(
[2012-05-28, 2012-07-04, 2012-10-08], dtype='datetime64[ns]', freq=None, tz=None)

Using this calendar, creating an index or doing offset arithmetic skips weekends and holidays (i.e., Memorial Day/July 4th).

In [147]: DatetimeIndex(start='7/1/2012', end='7/10/2012',
                  freq=CDay(calendar=cal)).to_pydatetime()
Out[147]:
array(
    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 [148]: offset = CustomBusinessDay(calendar=cal)

In [149]: datetime(2012, 5, 25) + offset
Out[149]: Timestamp('2012-05-29 00:00:00')

In [150]: datetime(2012, 7, 3) + offset
Out[150]: Timestamp('2012-07-05 00:00:00')

In [151]: datetime(2012, 7, 3) + 2 * offset
Ranges are defined by the `start_date` and `end_date` class attributes of `AbstractHolidayCalendar`. The defaults are below.

```python
In [153]: AbstractHolidayCalendar.start_date
Out[153]: Timestamp('1970-01-01 00:00:00')

In [154]: AbstractHolidayCalendar.end_date
Out[154]: Timestamp('2030-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as `datetime/Timestamp/string`.

```python
In [155]: AbstractHolidayCalendar.start_date = datetime(2012, 1, 1)

In [156]: 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 [158]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory,
       USLaborDay

In [159]: cal = get_calendar('ExampleCalendar')

In [160]: cal.rules
Out[160]:
[Holiday: MemorialDay (month=5, day=24, offset=<DateOffset: kwds={'weekday': MO(+1)}>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x9c164294>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: kwds={'weekday': MO(+2)}>)]

In [161]: new_cal = HolidayCalendarFactory('NewExampleCalendar', cal, USLaborDay)

In [162]: new_cal.rules
Out[162]:
[Holiday: Labor Day (month=9, day=1, offset=<DateOffset: kwds={'weekday': MO(+1)}>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: kwds={'weekday': MO(+2)}>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x9c164294>),
 Holiday: Memorial Day (month=5, day=24, offset=<DateOffset: kwds={'weekday': MO(+1)}>)]
```

## 20.6 Time series-related instance methods

### 20.6.1 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 [163]: ts = ts[:5]

In [164]: ts.shift(1)
Out[164]:
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 a freq argument which can accept a `DateOffset` class or other `timedelta`-like object or also a offset alias:

In [165]: ts.shift(5, freq=datetools.bday)
Out[165]:
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 [166]: ts.shift(5, freq='BM')
Out[166]:
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 [167]: ts.tshift(5, freq='D')
Out[167]:
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.

### 20.6.2 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 [168]: dr = date_range('1/1/2010', periods=3, freq=3 * datetools.bday)

In [169]: ts = Series(randn(3), index=dr)

In [170]: ts
Out[170]:
2010-01-01   -0.659574
2010-01-06  1.494522
2010-01-11  -0.778425
Freq: 3B, dtype: float64

In [171]: ts.asfreq(BDay())
Out[171]:
2010-01-01  -0.659574
2010-01-04  NaN
2010-01-05  NaN
2010-01-06  1.494522
2010-01-07  NaN
2010-01-08  NaN
2010-01-11  -0.778425
Freq: B, dtype: float64

asfreq provides a further convenience so you can specify an interpolation method for any gaps that may appear after
the frequency conversion

In [172]: ts.asfreq(BDay(), method='pad')
Out[172]:
2010-01-01  -0.659574
2010-01-04  -0.659574
2010-01-05  -0.659574
2010-01-06  1.494522
2010-01-07  1.494522
2010-01-08  1.494522
2010-01-11  -0.778425
Freq: B, dtype: float64

20.6.3 Filling forward / backward

Related to asfreq and reindex is the fillna function documented in the missing data section.

20.6.4 Converting to Python datetimes

DatetimeIndex can be converted to an array of Python native datetime.datetime objects using the
to_pydatetime method.

20.7 Up- and downsampling

With 0.8, pandas introduces 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 some cookbook examples for some advanced strategies

In [173]: rng = date_range('1/1/2012', periods=100, freq='S')
In [174]: ts = Series(randint(0, 500, len(rng)), index=rng)
In [175]: ts.resample('5Min', how='sum')
Out[175]:
2012-01-01   25103
Freq: 5T, dtype: int32
The `resample` function is very flexible and allows you to specify many different parameters to control the frequency conversion and resampling operation.

The `how` parameter can be a function name or numpy array function that takes an array and produces aggregated values:

```
In [176]: ts.resample('5Min')  # default is mean
Out[176]:
2012-01-01  251.03
Freq: 5T, dtype: float64
```

```
In [177]: ts.resample('5Min', how='ohlc')
Out[177]:
<table>
<thead>
<tr>
<th></th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-01-01</td>
<td>308</td>
<td>460</td>
<td>9</td>
<td>205</td>
</tr>
</tbody>
</table>
```

```
In [178]: ts.resample('5Min', how=np.max)
Out[178]:
2012-01-01  460
Freq: 5T, dtype: int32
```

Any function available via *dispatching* can be given to the `how` parameter by name, including `sum`, `mean`, `std`, `sem`, `max`, `min`, `median`, `first`, `last`, `ohlc`.

For downsampling, `closed` can be set to ‘left’ or ‘right’ to specify which end of the interval is closed:

```
In [179]: ts.resample('5Min', closed='right')
Out[179]:
2011-12-31 23:55:00  308.000000
2012-01-01 00:00:00  250.454545
Freq: 5T, dtype: float64
```

```
In [180]: ts.resample('5Min', closed='left')
Out[180]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

For upsampling, the `fill_method` and `limit` parameters can be specified to interpolate over the gaps that are created:

```
# from secondly to every 250 milliseconds
In [181]: ts[:2].resample('250L')
Out[181]:
2012-01-01 00:00:00.000  308
2012-01-01 00:00:00.250  NaN
2012-01-01 00:00:00.500  NaN
2012-01-01 00:00:00.750  NaN
2012-01-01 00:00:01.000  204
Freq: 250L, dtype: float64
```

```
In [182]: ts[:2].resample('250L', fill_method='pad')
Out[182]:
2012-01-01 00:00:00.000  308
2012-01-01 00:00:00.250  308
2012-01-01 00:00:00.500  308
2012-01-01 00:00:00.750  308
2012-01-01 00:00:01.000  204
Freq: 250L, dtype: int32
```

```
In [183]: ts[:2].resample('250L', fill_method='pad', limit=2)
```

---

580 Chapter 20. Time Series / Date functionality
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.

In [184]: ts.resample('5Min') # by default label='right'
Out[184]:
2012-01-01 251.03
Freq: 5T, dtype: float64

In [185]: ts.resample('5Min', label='left')
Out[185]:
2012-01-01 251.03
Freq: 5T, dtype: float64

In [186]: ts.resample('5Min', label='left', loffset='1s')
Out[186]:
2012-01-01 00:00:01 251.03
dtype: float64

The axis parameter can be set to 0 or 1 and allows you to resample the specified axis for a DataFrame.

kind can be set to ‘timestamp’ or ‘period’ to convert the resulting index to/from time-stamp and time-span representations. By default resample retains the input representation.

convention can be set to ‘start’ or ‘end’ when resampling period data (detail below). It specifies how low frequency periods are converted to higher frequency periods.

Note that 0.8 marks a watershed in the timeseries functionality in pandas. In previous versions, resampling had to be done using a combination of date_range, groupby with asof, and then calling an aggregation function on the grouped object. This was not nearly as convenient or performant as the new pandas timeseries API.

## 20.8 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.

### 20.8.1 Period

A Period represents a span of time (e.g., a day, a month, a quarter, etc). It can be created using a frequency alias:

In [187]: Period('2012', freq='A-DEC')
Out[187]: Period('2012', 'A-DEC')

In [188]: Period('2012-1-1', freq='D')
Out[188]: Period('2012-01-01', 'D')

In [189]: Period('2012-1-1 19:00', freq='H')
Out[189]: Period('2012-01-01 19:00', 'H')
Unlike time stamped data, pandas does not support frequencies at multiples of DateOffsets (e.g., ‘3Min’) for periods. Adding and subtracting integers from periods shifts the period by its own frequency.

```
In [190]: p = Period('2012', freq='A-DEC')
In [191]: p + 1
Out[191]: Period('2013', 'A-DEC')
In [192]: p - 3
Out[192]: Period('2009', 'A-DEC')
```

If `Period` freq is daily or higher (`D`, `H`, `T`, `S`, `L`, `U`, `N`), offsets and `timedelta`-like can be added if the result can have the same freq. Otherwise, `ValueError` will be raised.

```
In [193]: p = Period('2014-07-01 09:00', freq='H')
In [194]: p + Hour(2)
Out[194]: Period('2014-07-01 11:00', 'H')
In [195]: p + timedelta(minutes=120)
Out[195]: Period('2014-07-01 11:00', 'H')
In [196]: p + np.timedelta64(7200, 's')
Out[196]: 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.

```
In [197]: p = Period('2014-07', freq='M')
In [198]: p + MonthEnd(3)
Out[198]: 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:

```
In [199]: Period('2012', freq='A-DEC') - Period('2002', freq='A-DEC')
Out[199]: 10L
```

### 20.8.2 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:

```
In [200]: prng = period_range('1/1/2011', '1/1/2012', freq='M')
In [201]: prng
Out[201]:
```

The PeriodIndex constructor can also be used directly:

In [202]: PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[202]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='int64', freq='M')

Just like DatetimeIndex, a PeriodIndex can also be used to index pandas objects:

In [203]: ps = Series(randn(len(prng)), prng)
In [204]: ps
Out[204]:
2011-01 -0.253355
2011-02 -1.426908
2011-03  1.548971
2011-04 -0.088718
2011-05 -1.771348
2011-06 -0.989328
2011-07 -1.584789
2011-08 -0.288786
2011-09 -2.029806
2011-10 -0.761200
2011-11 -1.603608
2011-12  1.756171
2012-01  0.256502
Freq: M, dtype: float64

PeriodIndex supports addition and subtraction with the same rule as Period.

In [205]: idx = period_range('2014-07-01 09:00', periods=5, freq='H')
In [206]: idx
Out[206]:
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='int64', freq='H')
In [207]: idx + Hour(2)
Out[207]:
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='int64', freq='H')
In [208]: idx = period_range('2014-07', periods=5, freq='M')
In [209]: idx
Out[209]:
In [210]: idx + MonthEnd(3)
Out[210]:
PeriodIndex(['2014-10', '2014-11', '2014-12', '2015-01', '2015-02'], dtype='int64', freq='M')
20.8.3 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 [211]: ps['2011-01']
Out[211]: -0.25335528290092818

In [212]: ps[datetime(2011, 12, 25):]
Out[212]:
2011-12  1.756171
2012-01  0.256502
Freq: M, dtype: float64

In [213]: ps['10/31/2011':'12/31/2011']
Out[213]:
2011-10 -0.761200
2011-11 -1.603608
2011-12  1.756171
Freq: M, dtype: float64

Passing a string representing a lower frequency than PeriodIndex returns partial sliced data.

In [214]: ps['2011']
Out[214]:
2011-01  -0.253355
2011-02  -1.426908
2011-03   1.548971
2011-04  -0.088718
2011-05  -1.771348
2011-06  -0.989328
2011-07  -1.584789
2011-08  -0.288786
2011-09  -0.029806
2011-10  -0.761200
2011-11  -1.603608
2011-12   1.756171
Freq: M, dtype: float64

In [215]: dfp = DataFrame(randn(600,1), columns=['A'],
                   index=period_range('2013-01-01 9:00', periods=600, freq='T'))

In [216]: dfp
Out[216]:
       A
2013-01-01 09:00  0.020601
2013-01-01 09:01 -0.411719
2013-01-01 09:02  2.079413
2013-01-01 09:03 -1.077911
2013-01-01 09:04  0.999258
2013-01-01 09:05 -0.089851
2013-01-01 09:06  0.711329
... ... ... ...
2013-01-01 18:53 -1.340038
2013-01-01 18:54  1.315461
2013-01-01 18:55  2.396188
2013-01-01 18:56 -0.501527
2013-01-01 18:57  3.171938
```
2013-01-01 18:58  0.142019
2013-01-01 18:59  0.606998

[600 rows x 1 columns]
```

```
In [217]: dfp['2013-01-01 10H']
Out[217]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 10:00</td>
<td>-0.745396</td>
</tr>
<tr>
<td>2013-01-01 10:01</td>
<td>0.141880</td>
</tr>
<tr>
<td>2013-01-01 10:02</td>
<td>-1.077754</td>
</tr>
<tr>
<td>2013-01-01 10:03</td>
<td>-1.301174</td>
</tr>
<tr>
<td>2013-01-01 10:04</td>
<td>-0.269628</td>
</tr>
<tr>
<td>2013-01-01 10:05</td>
<td>-0.456347</td>
</tr>
<tr>
<td>2013-01-01 10:06</td>
<td>0.157766</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013-01-01 10:53</td>
<td>0.168057</td>
</tr>
<tr>
<td>2013-01-01 10:54</td>
<td>-0.214306</td>
</tr>
<tr>
<td>2013-01-01 10:55</td>
<td>-0.069739</td>
</tr>
<tr>
<td>2013-01-01 10:56</td>
<td>-1.511809</td>
</tr>
<tr>
<td>2013-01-01 10:57</td>
<td>0.307021</td>
</tr>
<tr>
<td>2013-01-01 10:58</td>
<td>1.449776</td>
</tr>
<tr>
<td>2013-01-01 10:59</td>
<td>0.782537</td>
</tr>
</tbody>
</table>

[60 rows x 1 columns]
```

As with `DatetimeIndex`, the endpoints will be included in the result. The example below slices data starting from 10:00 to 11:59.

```
In [218]: dfp['2013-01-01 10H':'2013-01-01 11H']
Out[218]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 10:00</td>
<td>-0.745396</td>
</tr>
<tr>
<td>2013-01-01 10:01</td>
<td>0.141880</td>
</tr>
<tr>
<td>2013-01-01 10:02</td>
<td>-1.077754</td>
</tr>
<tr>
<td>2013-01-01 10:03</td>
<td>-1.301174</td>
</tr>
<tr>
<td>2013-01-01 10:04</td>
<td>-0.269628</td>
</tr>
<tr>
<td>2013-01-01 10:05</td>
<td>-0.456347</td>
</tr>
<tr>
<td>2013-01-01 10:06</td>
<td>0.157766</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013-01-01 11:53</td>
<td>-0.064395</td>
</tr>
<tr>
<td>2013-01-01 11:54</td>
<td>0.350193</td>
</tr>
<tr>
<td>2013-01-01 11:55</td>
<td>1.336433</td>
</tr>
<tr>
<td>2013-01-01 11:56</td>
<td>-0.438701</td>
</tr>
<tr>
<td>2013-01-01 11:57</td>
<td>-0.915841</td>
</tr>
<tr>
<td>2013-01-01 11:58</td>
<td>0.294215</td>
</tr>
<tr>
<td>2013-01-01 11:59</td>
<td>0.040959</td>
</tr>
</tbody>
</table>

[120 rows x 1 columns]
```

### 20.8.4 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:

```
In [219]: p = Period('2011', freq='A-DEC')
```
In [220]: p
Out[220]: Period('2011', 'A-DEC')

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or
ending month:

In [221]: p.asfreq('M', how='start')
Out[221]: Period('2011-01', 'M')

In [222]: p.asfreq('M', how='end')
Out[222]: Period('2011-12', 'M')

The shorthands ‘s’ and ‘e’ are provided for convenience:

In [223]: p.asfreq('M', 's')
Out[223]: Period('2011-01', 'M')

In [224]: p.asfreq('M', 'e')
Out[224]: 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:

In [225]: p = Period('2011-12', freq='M')

In [226]: p.asfreq('A-NOV')
Out[226]: 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:

In [227]: p = Period('2012Q1', freq='Q-DEC')

In [228]: p.asfreq('D', 's')
Out[228]: Period('2012-01-01', 'D')

In [229]: p.asfreq('D', 'e')
Out[229]: Period('2012-03-31', 'D')

Q-MAR defines fiscal year end in March:

In [230]: p = Period('2011Q4', freq='Q-MAR')

In [231]: p.asfreq('D', 's')
Out[231]: Period('2011-01-01', 'D')

In [232]: p.asfreq('D', 'e')
Out[232]: Period('2011-03-31', 'D')
20.9 Converting between Representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```python
In [233]: rng = date_range('1/1/2012', periods=5, freq='M')

In [234]: ts = Series(randn(len(rng)), index=rng)

In [235]: ts
Out[235]:
2012-01-31 -0.016142
2012-02-29  0.865782
2012-03-31  0.246439
2012-04-30 -1.199736
2012-05-31  0.407620
Freq: M, dtype: float64

In [236]: ps = ts.to_period()

In [237]: ps
Out[237]:
2012-01  -0.016142
2012-02   0.865782
2012-03   0.246439
2012-04  -1.199736
2012-05   0.407620
Freq: M, dtype: float64

In [238]: ps.to_timestamp()
Out[238]:
2012-01-01 -0.016142
2012-02-01  0.865782
2012-03-01  0.246439
2012-04-01 -1.199736
2012-05-01  0.407620
Freq: MS, dtype: float64

In [239]: ps.to_timestamp('D', how='s')
Out[239]:
2012-01-01 -0.016142
2012-02-01  0.865782
2012-03-01  0.246439
2012-04-01 -1.199736
2012-05-01  0.407620
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 [239]: ps.to_timestamp('D', how='s')
Out[239]:
2012-01-01 -0.016142
2012-02-01  0.865782
2012-03-01  0.246439
2012-04-01 -1.199736
2012-05-01  0.407620
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 [240]: prng = period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [241]: ts = Series(randn(len(prng)), prng)

In [242]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9
```

20.9. Converting between Representations
In [243]: ts.head()
Out[243]:
1990-03-01 09:00 -2.470970
1990-06-01 09:00 -0.929915
1990-09-01 09:00  1.385889
1990-12-01 09:00 -1.830966
1991-03-01 09:00 -0.328505
Freq: H, dtype: float64

20.10 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 [244]: span = period_range('1215-01-01', '1381-01-01', freq='D')

In [245]: span
Out[245]:
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='int64', length=60632, freq='D')

To convert from a int64 based YYYYMMDD representation.

In [246]: s = Series([20121231, 20141130, 99991231])

In [247]: s
Out[247]:
0  20121231
1  20141130
2  99991231
dtype: int64

In [248]: def conv(x):
       .....:    return Period(year = x // 10000, month = x//100 % 100, day = x%100, freq='D')
       .....:

In [249]: s.apply(conv)
Out[249]:
0  2012-12-31
1  2014-11-30
2  9999-12-31
dtype: object

In [250]: s.apply(conv)[2]
Out[250]: Period('9999-12-31', 'D')

These can easily be converted to a PeriodIndex

In [251]: span = PeriodIndex(s.apply(conv))
In [252]: span
Out[252]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='int64', freq='D')

20.11 Time Zone Handling

Pandas provides rich support for working with timestamps in different time zones using pytz and dateutil libraries. dateutil support is new in 0.14.1 and currently 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.

20.11.1 Working with Time Zones

By default, pandas objects are time zone unaware:

In [253]: rng = date_range('3/6/2012 00:00', periods=15, freq='D')

In [254]: rng.tz is None
Out[254]: 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.

# pytz
In [255]: rng_pytz = date_range('3/6/2012 00:00', periods=10, freq='D',
...:                    tz='Europe/London')

In [256]: rng_pytz.tz
Out[256]: <DstTzInfo 'Europe/London' LMT-1 day, 23:59:00 STD>

# dateutil
In [257]: rng_dateutil = date_range('3/6/2012 00:00', periods=10, freq='D',
...:                     tz='dateutil/Europe/London')

In [258]: rng_dateutil.tz
Out[258]: tzfile('/usr/share/zoneinfo/Europe/London')

# dateutil - utc special case
In [259]: rng_utc = date_range('3/6/2012 00:00', periods=10, freq='D',
...:                      tz=dateutil.tz.tzutc())

In [260]: rng_utc.tz
Out[260]: 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.
# pytz
In [261]: tz_pytz = pytz.timezone('Europe/London')

In [262]: rng_pytz = date_range('3/6/2012 00:00', periods=10, freq='D',
                           tz=tz_pytz)

In [263]: rng_pytz.tz == tz_pytz
Out[263]: True

# dateutil
In [264]: tz_dateutil = dateutil.tz.gettz('Europe/London')

In [265]: rng_dateutil = date_range('3/6/2012 00:00', periods=10, freq='D',
                              tz=tz_dateutil)

In [266]: rng_dateutil.tz == tz_dateutil
Out[266]: 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`:

In [267]: ts = Series(randn(len(rng)), rng)

In [268]: ts_utc = ts.tz_localize('UTC')

In [269]: ts_utc
Out[269]:
2012-03-06 00:00:00+00:00 0.758606
2012-03-07 00:00:00+00:00 2.190827
2012-03-08 00:00:00+00:00 0.706087
2012-03-09 00:00:00+00:00 1.798831
2012-03-10 00:00:00+00:00 1.228481
2012-03-11 00:00:00+00:00 -0.179494
2012-03-12 00:00:00+00:00 0.634073
2012-03-13 00:00:00+00:00 0.262123
2012-03-14 00:00:00+00:00 1.928233
2012-03-15 00:00:00+00:00 0.322573
2012-03-16 00:00:00+00:00 -0.711113
2012-03-17 00:00:00+00:00 1.444272
2012-03-18 00:00:00+00:00 -0.352268
2012-03-19 00:00:00+00:00 0.213008
2012-03-20 00:00:00+00:00 -0.619340
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:

In [270]: ts_utc.tz_convert('US/Eastern')
Out[270]:
2012-03-05 19:00:00-05:00 0.758606
2012-03-06 19:00:00-05:00 2.190827
2012-03-07 19:00:00-05:00 0.706087
2012-03-08 19:00:00-05:00 1.798831
2012-03-09 19:00:00-05:00 1.228481
2012-03-10 19:00:00-05:00 -0.179494
2012-03-11 20:00:00-04:00 0.634073
2012-03-12 20:00:00-04:00 0.262123
2012-03-13 20:00:00-04:00 1.928233
2012-03-14 20:00:00-04:00 0.322573
2012-03-15 20:00:00-04:00 -0.711113
2012-03-16 20:00:00-04:00 1.444272
2012-03-17 20:00:00-04:00 -0.352268
2012-03-18 20:00:00-04:00 0.213008
2012-03-19 20:00:00-04:00 -0.619340
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 [271]: rng_eastern = rng_utc.tz_convert('US/Eastern')
In [272]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [273]: rng_eastern[5]  
Out[273]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', offset='D')

In [274]: rng_berlin[5]  
Out[274]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', offset='D')

Out[275]: True
```

Like Series, DataFrame, and DatetimeIndex, Timestamps can be converted to other time zones using `tz_convert`:

```python
In [276]: rng_eastern[5]  
Out[276]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', offset='D')

In [277]: rng_berlin[5]  
Out[277]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', offset='D')

In [278]: rng_eastern[5].tz_convert('Europe/Berlin')  
Out[278]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of Timestamps functions just like DatetimeIndex and Series:

```python
In [279]: rng[5]  
Out[279]: Timestamp('2012-03-11 00:00:00', offset='D')

In [280]: rng[5].tz_localize('Asia/Shanghai')  
Out[280]: Timestamp('2012-03-11 08:00:00+0800', tz='Asia/Shanghai')
```

Operations between Series in different time zones will yield UTC Series, aligning the data on the UTC timestamps:
In [281]: eastern = ts_utc.tz_convert('US/Eastern')
In [282]: berlin = ts_utc.tz_convert('Europe/Berlin')
In [283]: result = eastern + berlin
In [284]: result
Out[284]:
2012-03-06 00:00:00+00:00 1.517212
2012-03-07 00:00:00+00:00 4.381654
2012-03-08 00:00:00+00:00 1.412174
2012-03-09 00:00:00+00:00 3.597662
2012-03-10 00:00:00+00:00 2.456962
2012-03-11 00:00:00+00:00 -0.358988
2012-03-12 00:00:00+00:00 1.268146
2012-03-13 00:00:00+00:00 0.524245
2012-03-14 00:00:00+00:00 3.856466
2012-03-15 00:00:00+00:00 0.645146
2012-03-16 00:00:00+00:00 -1.422226
2012-03-17 00:00:00+00:00 2.888544
2012-03-18 00:00:00+00:00 -0.704537
2012-03-19 00:00:00+00:00 0.426017
2012-03-20 00:00:00+00:00 -1.238679
Freq: D, dtype: float64

In [285]: result.index
Out[285]:
DatetimeIndex(['2012-03-06', '2012-03-07', '2012-03-08', '2012-03-09',
              '2012-03-10', '2012-03-11', '2012-03-12', '2012-03-13',
              '2012-03-14', '2012-03-15', '2012-03-16', '2012-03-17',
              '2012-03-18', '2012-03-19', '2012-03-20'],
             dtype='datetime64[ns]', freq='D', tz='UTC')

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 [286]: didx = DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')

In [287]: didx
Out[287]:
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]', freq='H', tz='US/Eastern')

In [288]: didx.tz_localize(None)
Out[288]:
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=None)

In [289]: didx.tz_convert(None)
```python
Out[289]:
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=None)
# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [290]:
didx.tz_convert('UTC').tz_localize(None)
Out[290]:
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=None)
```

### 20.11.2 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.

```python
In [291]: rng_hourly = DatetimeIndex(['11/06/2011 00:00', '11/06/2011 01:00',
                              ....:     '11/06/2011 03:00'])
......:     '11/06/2011 03:00'])
......:

# This will fail as there are ambiguous times
In [292]: rng_hourly.tz_localize('US/Eastern')
---------------------------------------------------------------------------
AmbiguousTimeError Traceback (most recent call last)
<ipython-input-292-8c5fa6a37f5b> in <module>()
    ---> 1 rng_hourly.tz_localize('US/Eastern')

/home/joris/scipy/pandas/pandas/util/decorators.pyc in wrapper(*args, **kwargs)
    88                           return func(*args, **kwargs)
    89                      return _deprecate_kwarg
--> 90                    else:
    91                         kwargs[new_arg_name] = new_arg_value
    92
/home/joris/scipy/pandas/pandas/tseries/index.pyc in tz_localize(self, tz, ambiguous)
   1619                   new_dates = tslib.tz_localize_to_utc(self.asi8, tz,
   1620                       ambiguous=ambiguous)
 --> 1621                   new_dates = new_dates.view(_NS_DTYPE)
   1622            return self._shallow_copy(new_dates, tz=tz)

/home/joris/scipy/pandas/pandas/tslib.so in pandas.tzlib.tz_localize_to_utc (pandas/tslib.c:47148)()  # pandas.tzlib.tz_localize_to_utc

AmbiguousTimeError: Cannot infer dst time from Timestamp('2011-11-06 01:00:00'), try using the 'ambiguous' argument

In [293]: rng_hourly_eastern = rng_hourly.tz_localize('US/Eastern', ambiguous='infer')
```

---

**20.11. Time Zone Handling**

593
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 [295]: rng_hourly_dst = np.array([1, 1, 0, 0, 0])

In [296]: rng_hourly.tz_localize('US/Eastern', ambiguous=rng_hourly_dst).tolist()
Out[296]:
[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 [297]: rng_hourly.tz_localize('US/Eastern', ambiguous='NaT').tolist()
Out[297]:
[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 [298]: didx = DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')

In [299]: didx
Out[299]:
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', '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='US/Eastern')

In [300]: didx.tz_localize(None)
Out[300]:
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=None)

In [301]: didx.tz_convert(None)
Out[301]:
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=None)
In [302]: didx.tz_convert('UTC').tz_localize(None)
Out[302]:
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=None)
CHAPTER TWENTYONE

TIME DELTAS

Note: Starting in v0.15.0, 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.

Timedeltas are differences in times, expressed in difference units, e.g. days, hours, minutes, seconds. They can be both positive and negative.

21.1 Parsing

You can construct a Timedelta scalar through various arguments:

# strings
In [1]: Timedelta('1 days')
Out[1]: Timedelta('1 days 00:00:00')

In [2]: Timedelta('1 days 00:00:00')
Out[2]: Timedelta('1 days 00:00:00')

In [3]: Timedelta('1 days 2 hours')
Out[3]: Timedelta('1 days 02:00:00')

In [4]: 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]: Timedelta(days=1,seconds=1)
Out[5]: Timedelta('1 days 00:00:01')

# integers with a unit
In [6]: Timedelta(1,unit='d')
Out[6]: Timedelta('1 days 00:00:00')

# from a timedelta/np.timedelta64
In [7]: Timedelta(timedelta(days=1,seconds=1))
Out[7]: Timedelta('1 days 00:00:01')

In [8]: 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

```python
In [9]: Timedelta('-1us')
Out[9]: Timedelta('-1 days +23:59:59.999999')
```

A NaT

```python
In [10]: Timedelta('nan')
Out[10]: NaT

In [11]: Timedelta('nat')
Out[11]: NaT
```

`DateOffsets` (`Day`, `Hour`, `Minute`, `Second`, `Milli`, `Micro`, `Nano`) can also be used in construction.

```python
In [12]: Timedelta(Second(2))
Out[12]: Timedelta('0 days 00:00:02')
```

Further, operations among the scalars yield another scalar `Timedelta`

```python
In [13]: Timedelta(Day(2)) + Timedelta(Second(2)) + Timedelta('00:00:00.000123')
Out[13]: Timedelta('2 days 00:00:02.000123')
```

### 21.1.1 `to_timedelta`

**Warning:** Prior to 0.15.0 `pd.to_timedelta` would return a `Series` for list-like/Series input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for `Series` input, and `Timedelta` for scalar input.

The arguments to `pd.to_timedelta` are now `(arg, unit='ns', box=True)`, previously were `(arg, box=True, unit='ns')` as these are more logical.

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 will output a `TimedeltaIndex`

```python
In [14]: to_timedelta('1 days 06:05:01.00003')
Out[14]: Timedelta('1 days 06:05:01.000030')

In [15]: to_timedelta('15.5us')
Out[15]: Timedelta('0 days 00:00:00.000015')

In [16]: to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
Out[16]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT], dtype='timedelta64[ns]', freq=None)

In [17]: to_timedelta(np.arange(5), unit='s')
Out[17]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'], dtype='timedelta64[ns]', freq=None)

In [18]: to_timedelta(np.arange(5), unit='d')
Out[18]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype='timedelta64[ns]', freq=None)
```

### 21.2 Operations

You can operate on `Series/DataFrames` and construct `timedelta64[ns]` `Series` through subtraction operations on `datetime64[ns]` `Series`, or `Timestamps`.
In [19]: s = Series(date_range('2012-1-1', periods=3, freq='D'))

In [20]: td = Series([Timedelta(days=i) for i in range(3)])

In [21]: df = DataFrame(dict(A = s, B = td))

In [22]: df
Out[22]:
   A        B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [23]: df['C'] = df['A'] + df['B']

In [24]: df
Out[24]:
   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 [25]: df.dtypes
Out[25]:
A  datetime64[ns]
B  timedelta64[ns]
C  datetime64[ns]
dtype: object

In [26]: s - s.max()
Out[26]:
0  -2 days
1  -1 days
2   0 days
dtype: timedelta64[ns]

In [27]: s - datetime(2011,1,1,3,5)
Out[27]:
0  364 days 20:55:00
1  365 days 20:55:00
2  366 days 20:55:00
dtype: timedelta64[ns]

In [28]: s + timedelta(minutes=5)
Out[28]:
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 [29]: s + Minute(5)
Out[29]:
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 [30]: s + Minute(5) + Milli(5)
Out[30]:
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 [31]: y = s - s[0]

In [32]: y
Out[32]:
0    0 days
1    1 days
2    2 days
dtype: timedelta64[ns]

Series of timedeltas with NaT values are supported

In [33]: y = s - s.shift()

In [34]: y
Out[34]:
0     NaT
1    1 days
2    1 days
dtype: timedelta64[ns]

Elements can be set to NaT using np.nan analogously to datetimes

In [35]: y[1] = np.nan

In [36]: y
Out[36]:
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 [37]: s.max() - s
Out[37]:
0     2 days
1     1 days
2     0 days
dtype: timedelta64[ns]

In [38]: datetime(2011,1,1,3,5) - s
Out[38]:
0   -365 days 03:05:00
1   -366 days 03:05:00
2   -367 days 03:05:00
dtype: timedelta64[ns]

In [39]: timedelta(minutes=5) + s
Out[39]:
0  2012-01-01 00:05:00
1  2012-01-02 00:05:00
min, max and the corresponding idxmin, idxmax operations are supported on frames

In [40]: A = s - Timestamp('20120101') - Timedelta('00:05:05')

In [41]: B = s - Series(date_range('2012-1-2', periods=3, freq='D'))

In [42]: df = DataFrame(dict(A=A, B=B))

In [43]: df
Out[43]:
   A            B
0 -1 days +23:54:55 -1 days
1  0 days  23:54:55 -1 days
2  1 days  23:54:55 -1 days

In [44]: df.min()
Out[44]:
A -1 days +23:54:55
B -1 days +00:00:00
dtype: timedelta64[ns]

In [45]: df.min(axis=1)
Out[45]:
0 -1 days
1 -1 days
2 -1 days
dtype: timedelta64[ns]

In [46]: df.idxmin()
Out[46]:
A 0
B 0
dtype: int64

In [47]: df.idxmax()
Out[47]:
A 2
B 0
dtype: int64

min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

In [48]: df.min().max()
Out[48]: Timedelta('-1 days +23:54:55')

In [49]: df.min(axis=1).min()
Out[49]: Timedelta('-1 days +00:00:00')

In [50]: df.min().idxmax()
Out[50]: 'A'

In [51]: df.min(axis=1).idxmin()
Out[51]: 0

You can fillna on timedeltas. Integers will be interpreted as seconds. You can pass a timedelta to get a particular value.
In [52]: y.fillna(0)
Out[52]:
0  0 days
1  0 days
2  1 days
dtype: timedelta64[ns]

In [53]: y.fillna(10)
Out[53]:
0  0 days 00:00:10
1  0 days 00:00:10
2  1 days 00:00:00
dtype: timedelta64[ns]

In [54]: y.fillna(Timedelta('-1 days, 00:00:05'))
Out[54]:
0  -1 days +00:00:05
1  -1 days +00:00:05
2  1 days 00:00:00
dtype: timedelta64[ns]

You can also negate, multiply and use abs on Timedeltas

In [55]: td1 = Timedelta('1 days 2 hours 3 seconds')
In [56]: td1
Out[56]: Timedelta('-2 days +21:59:57')

In [57]: -1 * td1
Out[57]: Timedelta('1 days 02:00:03')

In [58]: - td1
Out[58]: Timedelta('1 days 02:00:03')

In [59]: abs(td1)
Out[59]: Timedelta('1 days 02:00:03')

21.3 Reductions

Numeric reduction operation for timedelta64[ns] will return Timedelta objects. As usual NaT are skipped during evaluation.

In [60]: y2 = Series(to_timedelta(['-1 days +00:00:05','nat','-1 days +00:00:05','1 days']))
In [61]: y2
Out[61]:
0  -1 days +00:00:05
1  NaT
2  -1 days +00:00:05
3  1 days 00:00:00
dtype: timedelta64[ns]

In [62]: y2.mean()
Out[62]: Timedelta('-1 days +16:00:03.333333')

In [63]: y2.median()
Out[63]: Timedelta('-1 days +00:00:05')

In [64]: y2.quantile(.1)
Out[64]: Timedelta('-1 days +00:00:05')

In [65]: y2.sum()
Out[65]: Timedelta('-1 days +00:00:10')

21.4 Frequency Conversion

New in version 0.13.

Timedelta Series, TimedeltaIndex, and Timedelta scalars can be converted to other ‘frequencies’ by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield Series and propagate NaT -> nan. Note that division by the numpy scalar is true division, while astyping is equivalent of floor division.

In [66]: td = Series(date_range('20130101',periods=4)) - Series(date_range('20121201',periods=4))

In [67]: td[2] += timedelta(minutes=5,seconds=3)

In [68]: td[3] = np.nan

In [69]: td
Out[69]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 NaN
dtype: timedelta64[ns]

# to days
In [70]: td / np.timedelta64(1,'D')
Out[70]:
0 31.000000
1 31.000000
2 31.003507
3 NaN
dtype: float64

In [71]: td.astype('timedelta64[D]')
Out[71]:
0 31
1 31
2 31
3 NaN
dtype: float64

# to seconds
In [72]: td / np.timedelta64(1,'s')
Out[72]:
0 2678400
1 2678400
2 2678703
3 NaN
dtype: float64

In [73]: td.astype('timedelta64[s]')
Out[73]:
0  2678400
1  2678400
2  2678703
3  NaN

dtype: float64

# to months (these are constant months)
In [74]: td / np.timedelta64(1,'M')
Out[74]:
0  1.018501
1  1.018501
2  1.018617
3  NaN

dtype: float64

Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series yields another timedelta64[ns] dtypes Series.

In [75]: td * -1
Out[75]:
0  -31 days 00:00:00
1  -31 days 00:00:00
2  -32 days 23:54:57
3  NaN

dtype: timedelta64[ns]

In [76]: td * Series([1,2,3,4])
Out[76]:
0   31 days 00:00:00
1   62 days 00:00:00
2   93 days 00:15:09
3  NaN

dtype: timedelta64[ns]

21.5 Attributes

You can access various components of the Timedelta or TimedeltaIndex directly using the attributes days,seconds,microseconds, nanoseconds. These are identical to the values returned by datetime.timedelta, in that, for example, the .seconds attribute represents the number of seconds >= 0 and < 1 day. These are signed according to whether the Timedelta is signed.

These operations can also be directly accessed via the .dt property of the Series as well.

Note: Note that the attributes are NOT the displayed values of the Timedelta. Use .components to retrieve the displayed values.

For a Series

In [77]: td.dt.days
Out[77]:
0   31
1   31

604 Chapter 21. Time Deltas
You can access the value of the fields for a scalar `Timedelta` directly.

```python
In [79]: tds = Timedelta('31 days 5 min 3 sec')
In [80]: tds.days
Out[80]: 31L
In [81]: tds.seconds
Out[81]: 303L
In [82]: (-tds).seconds
Out[82]: 86097L
```

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 [83]: td.dt.components
Out[83]:
          days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0        31     0        0         0            0              0          0
1        31     0        0         0            0              0          0
2        31     0        5         3            0              0          0
3         NaN   NaN       NaN       NaN            NaN            NaN          NaN
In [84]: td.dt.components.seconds
Out[84]:
          0  1  2  3
Name: seconds, dtype: float64
```

### 21.6 TimedeltaIndex

New in version 0.15.0.

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 [85]: TimedeltaIndex(['1 days','1 days, 00:00:05',
                   ....: np.timedelta64(2,'D'),timedelta(days=2,seconds=2))
                   ....:
```

21.6. TimedeltaIndex 605
Similarly to `date_range`, you can construct regular ranges of a `TimedeltaIndex`:

```python
In [86]: timedelta_range(start='1 days', periods=5, freq='D')
Out[86]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'],
                        dtype='timedelta64[ns]', freq='D')
```

```python
In [87]: timedelta_range(start='1 days', end='2 days', freq='30T')
Out[87]: 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')
```

### 21.6.1 Using the TimedeltaIndex

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.

```python
In [88]: s = Series(np.arange(100),
               index=timedelta_range('1 days', periods=100, freq='h'))
```

```python
In [89]: s
Out[89]:
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
5 days 00:00:00   96
5 days 01:00:00   97
5 days 02:00:00   98
5 days 03:00:00   99
```

21.6.1 Using the TimedeltaIndex

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.

```python
In [88]: s = Series(np.arange(100),
               index=timedelta_range('1 days', periods=100, freq='h'))
```

```python
In [89]: s
Out[89]:
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
5 days 00:00:00   96
5 days 01:00:00   97
5 days 02:00:00   98
5 days 03:00:00   99
```
Freq: H, dtype: int32

Selections work similarly, with coercion on string-likes and slices:

```
In [90]: s['1 day':'2 day']
Out[90]:
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
   ...
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, dtype: int32

In [91]: s['1 day 01:00:00']
Out[91]: 1

In [92]: s[Timedelta('1 day 1h')]
Out[92]: 1
```

Furthermore you can use partial string selection and the range will be inferred:

```
In [93]: s['1 day':'1 day 5 hours']
Out[93]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
Freq: H, dtype: int32
```

### 21.6.2 Operations

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

```
In [94]: tdi = TimedeltaIndex(['1 days',pd.NaT,'2 days'])

In [95]: tdi.tolist()
Out[95]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [96]: dti = date_range('20130101',periods=3)

In [97]: dti.tolist()
Out[97]:
[Timestamp('2013-01-01 00:00:00', offset='D'),
 Timestamp('2013-01-02 00:00:00', offset='D'),
 Timestamp('2013-01-03 00:00:00', offset='D')]
```
In [98]: (dti + tdi).tolist()
Out[98]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [99]: (dti - tdi).tolist()
Out[99]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]

### 21.6.3 Conversions

Similarly to frequency conversion on a Series above, you can convert these indices to yield another Index.

In [100]: tdi / np.timedelta64(1,'s')
Out[100]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

In [101]: tdi.astype('timedelta64[s]')
Out[101]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

Scalars type ops work as well. These can potentially return a different type of index.

# adding or timedelta and date -> datelike
In [102]: tdi + Timestamp('20130101')
Out[102]: DatetimeIndex(['2013-01-02', 'NaT', '2013-01-03'], dtype='datetime64[ns]', freq=None, tz=None)

# subtraction of a date and a timedelta -> datelike
# note that trying to subtract a date from a Timedelta will raise an exception
In [103]: (Timestamp('20130101') - tdi).tolist()
Out[103]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')]

# timedelta + timedelta -> timedelta
In [104]: tdi + Timedelta('10 days')
Out[104]: TimedeltaIndex(['11 days', NaT, '12 days'], dtype='timedelta64[ns]', freq=None)

# division can result in a Timedelta if the divisor is an integer
In [105]: tdi / 2
Out[105]: 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 [106]: tdi / tdi[0]
Out[106]: Float64Index([1.0, nan, 2.0], dtype='float64')

### 21.7 Resampling

Similar to timeseries resampling, we can resample with a TimedeltaIndex.

In [107]: s.resample('D')
Out[107]:
1 days 11.5
2 days 35.5
3 days 59.5
4 days 83.5
5 days 97.5
Freq: D, dtype: float64
New in version 0.15.

**Note:** While there was `pandas.Categorical` in earlier versions, the ability to use categorical data in `Series` and `DataFrame` is new.

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](#).

### 22.1 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]:**

```python
Out[2]:
0   a
1   b
```

**In [3]:**

```python
Out[3]:
0   a
1   b
```

**In [4]:**

```python
Out[4]:
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    8  0 - 9
```

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

In [15]: df
Out[15]:
   A  B
0  a  NaN
1  b  b
2  c  c
3  a  NaN

You can also specify differently ordered categories or make the resulting data ordered, by passing these arguments to `astype()`:

In [16]: s = pd.Series(['a', 'b', 'c', 'a'])

In [17]: s_cat = s.astype("category", categories=['b', 'c', 'd'], ordered=False)

In [18]: s_cat
Out[18]:
0  NaN
1  b
2  c
3  NaN
dtype: category
Categories (3, object): [b, c, d]

Categorical data has a specific category `dtype`:

In [19]: df.dtypes
Out[19]:
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)`:

In [20]: s = pd.Series(['a', 'b', 'c', 'a'])

In [21]: s
Out[21]:
 0  a
 1  b
 2  c
 3  a
dtype: object

In [22]: s2 = s.astype('category')
In [23]: s2
Out[23]:
0  a
1  b
2  c
3  a
dtype: category
Categories (3, object): [a, b, c]

In [24]: s3 = s2.astype('string')

In [25]: s3
Out[25]:
0  a
1  b
2  c
3  a
dtype: object

In [26]: np.asarray(s2)
Out[26]: array(['a', 'b', 'c', 'a'], dtype=object)

If you have already codes and categories, you can use the from_codes() constructor to save the factorize step during normal constructor mode:

In [27]: splitter = np.random.choice([0,1], 5, p=[0.5,0.5])

In [28]: s = pd.Series(pd.Categorical.from_codes(splitter, categories=["train", "test"]))

22.2 Description

Using .describe() on categorical data will produce similar output to a Series or DataFrame of type string.

In [29]: cat = pd.Categorical(["a","c","c",np.nan], categories=["b","a","c",np.nan] )

In [30]: df = pd.DataFrame({"cat":cat, "s":["a","c","c",np.nan]})

In [31]: df.describe()
Out[31]:
          cat  s
count     3  3
unique    2  2
top       c  c
freq      2  2

In [32]: df["cat"].describe()
Out[32]:
count     3
unique    2
top       c
freq      2
Name: cat, dtype: object
22.3 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.

```
In [33]: s = pd.Series(['a', 'b', 'c', 'a'], dtype='category')
```

```
In [34]: s.cat.categories
Out[34]: Index(['a', 'b', 'c'], dtype='object')
```

```
In [35]: s.cat.ordered
Out[35]: False
```

It’s also possible to pass in the categories in a specific order:

```
In [36]: s = pd.Series(pd.Categorical(['a', 'b', 'c', 'a'], categories=['c', 'b', 'a']))
```

```
In [37]: s.cat.categories
Out[37]: Index(['c', 'b', 'a'], dtype='object')
```

```
In [38]: s.cat.ordered
Out[38]: False
```

**Note:** New categorical data are NOT automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered `Categorical`.

22.3.1 Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `Categorical.rename_categories()` method:

```
In [39]: s = pd.Series(['a', 'b', 'c', 'a'], dtype='category')
```

```
In [40]: s
Out[40]:
0   a
1   b
2   c
3   a
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

```
In [41]: s.cat.categories = ['Group %s' % g for g in s.cat.categories]
```

```
In [42]: s
Out[42]:
0  Group a
1  Group b
2  Group c
3  Group a
dtype: category
Categories (3, object): ['Group a', 'Group b', 'Group c']
```

```
In [43]: s.cat.rename_categories([1, 2, 3])
Out[43]:
0  Group a
1  Group b
2  Group c
3  Group a
dtype: category
Categories (3, object): ['Group a', 'Group b', 'Group c']
```
0 1
1 2
2 3
3 1
dtype: category
Categories (3, int64): [1, 2, 3]

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 operations, while most other operation under `Series.cat` per default return a new Series of dtype `category`.

Categories must be unique or a `ValueError` is raised:

```
In [44]: try:
    ....:     s.cat.categories = [1,1,1]
    ....:     except ValueError as e:
    ....:         print("ValueError: " + str(e))
    ....:
ValueError: Categorical categories must be unique
```

### 22.3.2 Appending new categories

Appending categories can be done by using the `Categorical.add_categories()` method:

```
In [45]: s = s.cat.add_categories([4])
```

```
In [46]: s.cat.categories
Out[46]: Index(['Group a', 'Group b', 'Group c', 4], dtype='object')
```

```
In [47]: s
Out[47]:
0  Group a
1  Group b
2  Group c
3  Group a
dtype: category
Categories (4, object): [Group a, Group b, Group c, 4]
```

### 22.3.3 Removing categories

Removing categories can be done by using the `Categorical.remove_categories()` method. Values which are removed are replaced by `np.nan`:

```
In [48]: s = s.cat.remove_categories([4])
```

```
In [49]: s
Out[49]:
0  Group a
1  Group b
2  Group c
3  Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]```
22.3.4 Removing unused categories

Removing unused categories can also be done:

In [50]: s = pd.Series(pd.Categorical(["a","b","a"], categories=["a","b","c","d"]))

In [51]: s
Out[51]:
0  a
1  b
2  a
dtype: category
Categories (4, object): [a, b, c, d]

In [52]: s.cat.remove_unused_categories()
Out[52]:
0  a
1  b
2  a
dtype: category
Categories (2, object): [a, b]

22.3.5 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()`.

In [53]: s = pd.Series(["one","two","four", ","], dtype="category")

In [54]: s
Out[54]:
0  one
1  two
2  four
3  
dtype: category
Categories (4, object): [-, four, one, two]

In [55]: s = s.cat.set_categories(["one","two","three","four"])

In [56]: s
Out[56]:
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!
22.4 Sorting and Order

Warning: The default for construction has changed in v0.16.0 to ordered=False, from the prior implicit ordered=True

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.

In [57]: s = pd.Series(pd.Categorical(["a","b","c","a"], ordered=False))

In [58]: s.sort()

In [59]: s = pd.Series(["a","b","c","a"]).astype('category', ordered=True)

In [60]: s.sort()

In [61]: s
Out[61]:
0  a
3  a
1  b
2  c
dtype: category
Categories (3, object): [a < b < c]

In [62]: s.min(), s.max()
Out[62]: ('a', 'c')

You can set categorical data to be ordered by using as_ordered() or unordered by using as_unordered(). These will by default return a new object.

In [63]: s.cat.as_ordered()
Out[63]:
0  a
3  a
1  b
2  c
dtype: category
Categories (3, object): [a < b < c]

In [64]: s.cat.as_unordered()
Out[64]:
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 [65]: s = pd.Series([1,2,3,1], dtype="category")

In [66]: s = s.cat.set_categories([2,3,1], ordered=True)

In [67]: s
22.4.1 Reordering

Reordering the categories is possible via the `Categorical.reorder_categories()` and the `Categorical.set_categories()` methods. For `Categorical.reorder_categories()`, all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [71]: s = pd.Series([1,2,3,1], dtype="category")

In [72]: s = s.cat.reorder_categories([2,3,1], ordered=True)

In [73]: s
Out[73]:
0  1
1  2
2  3
3  1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [74]: s.sort()

In [75]: s
Out[75]:
1  2
2  3
0  1
3  1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [76]: s.min(), s.max()
Out[76]: (2, 1)
```
Note: Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the Series, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the Series are changed.

Note: If the Categorical is not ordered, Series.min() and Series.max() will raise TypeError. Numeric operations like +, -, *, / and operations based on them (e.g. ‘Series.median()’, which would need to compute the mean between two values if the length of an array is even) do not work and raise a TypeError.

22.4.2 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 [77]: dfs = pd.DataFrame({'A' : pd.Categorical(list('bbeebbaa'), categories=['e','a','b'], ordered=True), 'B' : [1,2,1,2,2,1,2,1] })

    ....:
    ....:

In [78]: dfs.sort(['A', 'B'])

Out[78]:
       A  B
0    e  1
1    e  2
2    a  1
3    a  2
4    b  1
5    b  1
6    b  2
7    b  2

Reordering the categories changes a future sort.

In [79]: dfs['A'] = dfs['A'].cat.reorder_categories(['a','b','e'])

In [80]: dfs.sort(['A', 'B'])

Out[80]:
       A  B
0    a  1
1    a  2
2    b  1
3    b  1
4    b  2
5    b  2
6    e  1
7    e  2

22.5 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 an TypeError because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [81]: cat = pd.Series([1,2,3]).astype("category", categories=[3,2,1], ordered=True)
In [82]: cat_base = pd.Series([2,2,2]).astype("category", categories=[3,2,1], ordered=True)
In [83]: cat_base2 = pd.Series([2,2,2]).astype("category", ordered=True)
In [84]: cat
Out[84]:
0  1
1  2
2  3
dtype: category
Categories (3, int64): [3 < 2 < 1]
In [85]: cat_base
Out[85]:
0  2
1  2
2  2
dtype: category
Categories (3, int64): [3 < 2 < 1]
In [86]: cat_base2
Out[86]:
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:

```
In [87]: cat > cat_base
Out[87]:
0   True
1  False
2  False
dtype: bool
In [88]: cat > 2
Out[88]:
0   True
1  False
2  False
dtype: bool
```

Equality comparisons work with any list-like object of same length and scalars:

22.5. Comparisons
```python
In [89]: cat == cat_base
Out[89]:
0   False
1    True
2   False
dtype: bool

In [90]: cat == np.array([1,2,3])
Out[90]:
0    True
1    True
2    True
dtype: bool

In [91]: cat == 2
Out[91]:
0   False
1    True
2   False
dtype: bool

This doesn't work because the categories are not the same:
```
In [92]: try:
....:    cat > cat_base2
....: except TypeError as e:
....:    print("TypeError: ", str(e))
....:
TypeError: Categoricals can only be compared if 'categories' are the same

If you want to do a “non-equality” comparison of a categorical series with a list-like object which is not categorical data, you need to be explicit and convert the categorical data back to the original values:
```
In [93]: base = np.array([1,2,3])

In [94]: try:
....:    cat > base
....: except TypeError as e:
....:    print("TypeError: ", str(e))
....:
TypeError: Cannot compare a Categorical for op __gt__ with type <type 'numpy.ndarray'>. If you want to compare values, use 'np.asarray(cat) <op> other'.

In [95]: np.asarray(cat) > base
Out[95]: array([False, False, False], dtype=bool)
```

### 22.6 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:

```
In [96]: s = pd.Series(pd.Categorical(["a","b","c","c"], categories=["c","a","b","d"])

In [97]: s.value_counts()
```

```
Out[97]:
```

---

620 Chapter 22. Categorical Data
Groupby will also show “unused” categories:

```python
In [98]: cats = pd.Categorical(["a","b","b","b","c","c","c"], categories=["a","b","c","d"])  
In [99]: df = pd.DataFrame({'cats':cats,'values':[1,2,2,2,3,4,5]})  
In [100]: df.groupby('cats').mean()  
Out[100]:

<table>
<thead>
<tr>
<th>cats</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

```python
In [101]: cats2 = pd.Categorical(["a","a","b","b"], categories=["a","b","c"])  
In [102]: df2 = pd.DataFrame({'cats':cats2,'B':['c','d','c','d'],'values':[1,2,3,4]})  
In [103]: df2.groupby(["cats","B"]).mean()  
Out[103]:

<table>
<thead>
<tr>
<th>cats</th>
<th>B</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

Pivot tables:

```python
In [104]: raw_cat = pd.Categorical(["a","a","b","b"], categories=["a","b","c"])  
In [105]: df = pd.DataFrame({'A':raw_cat,'B':['c','d','c','d'],'values':[1,2,3,4]})  
In [106]: df.pivot_table(df, values='values', index=['A', 'B'])  
Out[106]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

22.7 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.ix`, `.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.
22.7.1 Getting

If the slicing operation returns either a DataFrame or a column of type Series, the category dtype is preserved.

```python
In [107]: idx = pd.Index(["h","i","j","k","l","m","n",])
```

```python
In [108]: cats = pd.Series(["a","b","b","b","c","c","c"], dtype="category", index=idx)
```

```python
In [109]: values= [1,2,2,2,3,4,5]
```

```python
In [110]: df = pd.DataFrame({"cats":cats,"values":values}, index=idx)
```

```python
In [111]: df.iloc[2:4,:
Out[111]:
cats  values
j  b  2
k  b  2
```

```python
In [112]: df.iloc[2:4,].dtypes
Out[112]:
cats  category
values  int64
dtype: object
```

```python
In [113]: df.loc["h":"j","cats"
Out[113]:
h  a
i  b
j  b
Name: cats, dtype: category
Categories (3, object): [a, b, c]
```

```python
In [114]: df.ix["h":"j",0:1]
Out[114]:
cats
h  a
i  b
j  b
```

```python
In [115]: df[df["cats"] == "b"]
Out[115]:
cats  values
i  b  2
j  b  2
k  b  2
```

An example where the category type is not preserved is if you take one single row: the resulting Series is of dtype object:

```python
# get the complete "h" row as a Series
In [116]: df.loc["h", :]
Out[116]:
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”.
In [117]: df.iat[0,0]
Out[117]: 'a'

In [118]: df["cats"].cat.categories = ["x","y","z"]

In [119]: df.at["h","cats"] # returns a string
Out[119]: '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:

In [120]: df.loc["h","cats"]
Out[120]:
  h   x
Name: cats, dtype: category
Categories (3, object): [x, y, z]

### 22.7.2 Setting

Setting values in a categorical column (or `Series`) works as long as the value is included in the `categories`:

In [121]: idx = pd.Index(["h","i","j","k","l","m","n"])

In [122]: cats = pd.Categorical(["a","a","a","a","a","a","a"], categories=["a","b"])

In [123]: values = [1,1,1,1,1,1,1]

In [124]: df = pd.DataFrame({"cats":cats,"values":values}, index=idx)

In [125]: df.iloc[2:4,:] = [["b",2],["b",2]]

In [126]: df
Out[126]:
   cats  values
  h    a     1
  i    a     1
  j    b     2
  k    b     2
  l    a     1
  m    a     1
  n    a     1

In [127]: 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 [128]: df.loc["j":"k","cats"] = pd.Categorical(["a","a"], categories=["a","b"])

In [129]: df
Out[129]:
   cats  values
    j    a     2
    k    a     2
    l    a     1
    m    a     0
    n    a     1

### 22.7. Data munging
In [130]: try:  
.....:    df.loc["j":"k","cats"] = pd.Categorical(["b","b"], categories=["a","b","c"])  
.....: except ValueError as e:  
.....:    print("ValueError: " + str(e))  
.....:  
ValueError: Cannot set a Categorical with another, without identical categories

Assigning a Categorical to parts of a column of other types will use the values:

In [131]: df = pd.DataFrame({"a": [1,1,1,1,1], "b": ["a","a","a","a","a"]})
In [132]: df.loc[1:2,"a"] = pd.Categorical(["b","b"], categories=["a","b"])
In [133]: df.loc[2:3,"b"] = pd.Categorical(["b","b"], categories=["a","b"])

In [134]: df  
Out[134]:  
a  b  
0  1  a  
1  b  a  
2  b  b  
3  l  b  
4  l  a

In [135]: df.dtypes  
Out[135]:  
a object  
b object  
dtype: object

### 22.7.3 Merging

You can concat two DataFrames containing categorical data together, but the categories of these categoricals need to be the same:

In [136]: cat = pd.Series(["a","b"], dtype=category)  
In [137]: vals = [1,2]  
In [138]: df = pd.DataFrame({"cats":cat, "vals":vals})  
In [139]: res = pd.concat([df,df])

In [140]: res  
Out[140]:  
cats  vals  
0  a  1  
1  b  2  
0  a  1
In this case the categories are not the same and so an error is raised:

```python
In [142]: df_different = df.copy()
In [143]: df_different["cats"].cat.categories = ["c","d"]
In [144]: try:
   .....:     pd.concat([df,df_different])
   .......: except ValueError as e:
   .......:     print("ValueError: " + str(e))
   .......
ValueError: incompatible categories in categorical concat
```

The same applies to `df.append(df_different)`.

### 22.8 Getting Data In/Out

New in version 0.15.2.

Writing data (`Series`, `Frames`) to a HDF store that contains a `category` dtype was implemented in 0.15.2. See [here](#) for an example and caveats.

Writing data to and reading data from `Stata` format files was implemented in 0.15.2. See [here](#) for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to `category` and assign the right categories and categories ordering.

```python
In [145]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))
   # rename the categories
In [146]: s.cat.categories = ["very good", "good", "bad"]
   # reorder the categories and add missing categories
In [147]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])
In [148]: df = pd.DataFrame({"cats":s, "vals":[1,2,3,4,5,6]})
In [149]: csv = StringIO()
In [150]: df.to_csv(csv)
In [151]: df2 = pd.read_csv(StringIO(csv.getvalue()))
In [152]: df2.dtypes
Out[152]:
      Unnamed: 0  int64
      cats    object
```

```
vals    int64
dtype: object

In [153]: df2["cats"]
Out[153]:
0  very good
1    good
2    good
3  very good
4  very good
5     bad
Name: cats, dtype: object

# Redo the category
In [154]: df2("cats") = df2["cats"].astype("category")

In [155]: df2["cats"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"],
                              inplace=True)

In [156]: df2.dtypes
Out[156]:
         0         cats  vals
dtype: object

In [157]: df2["cats"]
Out[157]:
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.

22.9 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

There are two ways a np.nan can be represented in categorical data: either the value is not available (“missing value”) or np.nan is a valid category.

In [158]: s = pd.Series(["a","b",np.nan,"a"], dtype="category")

# only two categories
In [159]: s
Out[159]:
0    a
1    b
2  NaN
3    a
dtype: category
Categories (2, object): [a, b]

In [160]: s2 = pd.Series(["a","b","c","a"], dtype="category")

In [161]: s2.cat.categories = [1,2, np.nan]

# three categories, np.nan included
In [162]: s2
Out[162]:
0  1
1  2
2  NaN
3  1
dtype: category
Categories (3, object): [1, 2, NaN]

Note: As integer Series can’t include NaN, the categories were converted to object.

Note: Missing value methods like isnull and fillna will take both missing values as well as np.nan categories into account:

In [163]: c = pd.Series(["a","b", np.nan], dtype="category")

In [164]: c.cat.set_categories(["a","b", np.nan], inplace=True)

# will be inserted as a NA category:
In [165]: c[0] = np.nan

In [166]: s = pd.Series(c)

In [167]: s
Out[167]:
0  NaN
1  b
2  NaN
dtype: category
Categories (3, object): [a, b, NaN]

In [168]: pd.isnull(s)
Out[168]:
0   True
1  False
2   True
dtype: bool

In [169]: s.fillna("a")
Out[169]:
0  a
1  b
2  a
dtype: category
Categories (3, object): [a, b, NaN]
22.9.1 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!

22.10 Gotchas

22.10.1 Memory Usage

The memory usage of a *Categorical* is proportional to the number of categories times the length of the data. In contrast, an *object* dtype is a constant times the length of the data.

```python
In [170]: s = pd.Series(['foo','bar']*1000)

# object dtype
In [171]: s.nbytes
Out[171]: 8000

# category dtype
In [172]: s.astype('category').nbytes
Out[172]: 2008
```

*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 [173]: s = pd.Series(['foo%04d' % i for i in range(2000)])

# object dtype
In [174]: s.nbytes
Out[174]: 8000

# category dtype
In [175]: s.astype('category').nbytes
Out[175]: 12000
```

22.10.2 Old style constructor usage

In earlier versions than pandas 0.15, a *Categorical* could be constructed by passing in precomputed *codes* (called then *labels*) instead of values with categories. The *codes* were interpreted as pointers to the categories with `-1 as NaN`. This type of constructor usage is replaced by the special constructor `Categorical.from_codes()`.

Unfortunately, in some special cases, using code which assumes the old style constructor usage will work with the current pandas version, resulting in subtle bugs:
```python
>>> cat = pd.Categorical([1, 2], [1, 2, 3])
>>> # old version
>>> cat.get_values()
array([2, 3], dtype=int64)
>>> # new version
>>> cat.get_values()
array([1, 2], dtype=int64)
```

**Warning:** If you used *Categoricals* with older versions of pandas, please audit your code before upgrading and change your code to use the *from_codes()* constructor.

### 22.10.3 *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*:

```python
In [176]: try:
   ....:     np.dtype("category")
   ....: except TypeError as e:
   ....:     print("TypeError: " + str(e))
   ....:
TypeError: data type "category" not understood
```

```python
In [177]: dtype = pd.Categorical(["a"]).dtype
```

```python
In [178]: try:
   ....:     np.dtype(dtype)
   ....: except TypeError as e:
   ....:     print("TypeError: " + str(e))
   ....:
TypeError: data type not understood
```

Dtype comparisons work:

```python
In [179]: dtype == np.str_
Out[179]: False
```

```python
In [180]: np.str_ == dtype
Out[180]: False
```

To check if a Series contains Categorical data, with pandas 0.16 or later, use `hasattr(s, 'cat')`:

```python
In [181]: hasattr(pd.Series(["a"], dtype='category'), 'cat')
Out[181]: True
```

```python
In [182]: hasattr(pd.Series(["a"], 'cat')
Out[182]: 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).

```python
In [183]: s = pd.Series(pd.Categorical([[1,2,3,4]]))
```

```python
In [184]: try:
   ....:     np.sum(s)
   ....:
```

---

22.10. Gotchas 629
......: 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/pydata/pandas!

22.10.4 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 [185]: df = pd.DataFrame({"a": [1, 2, 3, 4],
......:                     "b": ["a", "b", "c", "d"],
......:                     "cats": pd.Categorical([1, 2, 3, 2])})

In [186]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[186]:
0 <type 'long'>
1 <type 'long'>
2 <type 'long'>
3 <type 'long'>
dtype: object

In [187]: df.apply(lambda col: col.dtype, axis=0)
Out[187]:
a object
b object
cats object
dtype: object

22.10.5 Categorical Index

New in version 0.16.1.

A new CategoricalIndex index type is introduced in version 0.16.1. See the advanced indexing docs for a more detailed explanation.

Setting the index, will create create a CategoricalIndex

In [188]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])

In [189]: strings = ["a", "b", "c", "d"]

In [190]: values = [4, 2, 3, 1]

In [191]: df = pd.DataFrame({"strings": strings, "values": values}, index=cats)

In [192]: df.index
Out[192]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False, dtype='category')

# This now sorts by the categories order
In [193]: df.sort_index()
In previous versions (<0.16.1) there is no index of type `category`, so setting the index to categorical column will convert the categorical data to a “normal” dtype first and therefore remove any custom ordering of the categories.

### 22.10.6 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 [194]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])

In [195]: s = pd.Series(cat, name="cat")

In [196]: cat
Out[196]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [197]: s.iloc[0:2] = 10

In [198]: cat
Out[198]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [199]: df = pd.DataFrame(s)

In [200]: df["cat"].cat.categories = [1,2,3,4,5]

In [201]: cat
Out[201]:
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]

In [202]: df = pd.DataFrame(s)

In [203]: df["cat"].cat.categories = [1,2,3,4,5]

In [204]: cat
Out[204]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [205]: s.iloc[0:2] = 10

In [206]: cat
Out[206]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
```

Use `copy=True` to prevent such a behaviour or simply don’t reuse `Categoricals`:

```
In [202]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])

In [203]: s = pd.Series(cat, name="cat", copy=True)

In [204]: cat
Out[204]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [205]: s.iloc[0:2] = 10

In [206]: cat
Out[206]:
[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:

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

The plots in this document are made using matplotlib’s ggplot style (new in version 1.4):

```python
import matplotlib
matplotlib.style.use('ggplot')
```

If your version of matplotlib is 1.3 or lower, you can set `display.mpl_style` to 'default' with `pd.options.display.mpl_style = 'default'` to produce more appealing plots. When set, matplotlib’s `rcParams` are changed (globally!) to nicer-looking settings.

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.

### 23.1 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()`:

```python
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 0xaf7a5d6c>
```
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 `x` and `y` keywords in `plot()`:

```python
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')
```

Out[10]: `<matplotlib.axes._subplots.AxesSubplot at 0xaf67144c>`
23.2 Other Plots

The `kind` keyword argument of `plot()` accepts a handful of values for plots other than the default Line 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
In addition to these kinds, there are the `DataFrame.hist()` and `DataFrame.boxplot()` methods, which use a separate interface.

Finally, there are several plotting functions in `pandas.tools.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.

### 23.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

```python
In [11]: plt.figure();

In [12]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
```

```python
Out[12]: <matplotlib.lines.Line2D at 0xaf5d880c>
```
Calling a DataFrame's `plot()` method with `kind='bar'` produces a multiple bar plot:

```
In [13]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
In [14]: df2.plot(kind='bar');
```
To produce a stacked bar plot, pass `stacked=True`:

```python
In [15]: df2.plot(kind='bar', stacked=True);
```
To get horizontal bar plots, pass kind='barh':

```
In [16]: df2.plot(kind='barh', stacked=True);
```
23.2.2 Histograms

New in version 0.15.0.

Histogram can be drawn specifying kind='hist'.

In [17]: 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 [18]: plt.figure();

In [19]: df4.plot(kind='hist', alpha=0.5)
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0xaf218e4c>
Histogram can be stacked by `stacked=True`. Bin size can be changed by `bins` keyword.

```python
In [20]: plt.figure();

In [21]: df4.plot(kind='hist', stacked=True, bins=20)
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0xaf203dac>
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 [22]: plt.figure();

In [23]: df4['a'].plot(kind='hist', orientation='horizontal', cumulative=True)
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0xaf4d376c>
pandas: powerful Python data analysis toolkit, Release 0.16.2

See the `hist` method and the `matplotlib hist documentation` for more.

The existing interface `DataFrame.hist` to plot histogram still can be used.

```
In [24]: plt.figure();

In [25]: df['A'].diff().hist()
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0xaf54440c>
```
**DataFrames**.hist() plots the histograms of the columns on multiple subplots:

```python
In [26]: plt.figure()
Out[26]: <matplotlib.figure.Figure at 0xaf22c6ac>
```

```python
In [27]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[27]:
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
New in version 0.10.0.

The `by` keyword can be specified to plot grouped histograms:

```
In [28]: data = pd.Series(np.random.randn(1000))

In [29]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[29]:
array([[[<matplotlib.axes._subplots.AxesSubplot object at 0xaeb9e7ec>,
         <matplotlib.axes._subplots.AxesSubplot object at 0xae8208ac>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0xae7e092c>,
         <matplotlib.axes._subplots.AxesSubplot object at 0xae78ee6c>]]], dtype=object)
```
23.2.3 Box Plots

Boxplot can be drawn calling a Series and DataFrame.plot with kind='box', or DataFrame.boxplot to visualize the distribution of values within each column.

New in version 0.15.0.

plot method now supports kind='box' to draw boxplot.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

In [30]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])

In [31]: df.plot(kind='box')
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0xadf5cfcc>
Boxplot can be colorized by passing the `color` keyword. You can pass a dictionary whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the dictionary, default colors are used for the corresponding artists. Also, boxplot has the `sym` keyword to specify fliers style.

When you pass other type of arguments via the `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 the `return_type`.

In [32]:
    color = dict(boxes='DarkGreen', whiskers='DarkOrange',
                  medians='DarkBlue', caps='Gray')
    ...

In [33]:
    df.plot(kind='box', color=color, sym='r+')
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0xadf84f2c>
Also, you can pass other keywords supported by matplotlib's `boxplot`. For example, horizontal and custom-positioned boxplot can be drawn by `vert=False` and `positions` keywords.

```
In [34]: df.plot(kind='box', vert=False, positions=[1, 4, 5, 6, 8])
Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0xade254cc>
```
See the \texttt{boxplot} method and the \texttt{matplotlib} \texttt{boxplot} documenation for more.

The existing interface \texttt{DataFrame.boxplot} to plot boxplot still can be used.

\begin{verbatim}
In [35]: df = pd.DataFrame(np.random.rand(10,5))

In [36]: plt.figure();

In [37]: bp = df.boxplot()
\end{verbatim}
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```python
In [38]: df = pd.DataFrame(np.random.rand(10,2), columns=['Col1', 'Col2'] )

In [39]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])

In [40]: plt.figure();

In [41]: bp = df.boxplot(by='X')
```
You can also pass a subset of columns to plot, as well as group by multiple columns:

```python
In [42]: df = pd.DataFrame(np.random.rand(10,3), columns=['Col1', 'Col2', 'Col3'])
In [43]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [44]: df['Y'] = pd.Series(['A','B','A','B','A','B','A','B','A','B'])
In [45]: plt.figure();
In [46]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```
Basically, plot functions return `matplotlib Axes` as a return value. In `boxplot`, the return type can be changed by argument `return_type`, and whether the subplots is enabled (`subplots=True` in `plot` or `by` is specified in `boxplot`).

When `subplots=False / by` is `None`:

- if `return_type` is `'dict'`, a dictionary containing the `matplotlib Lines` is returned. The keys are “boxes”, “caps”.
  This is the default of `boxplot` in historical reason. Note that `plot(kind='box')` returns `Axes` as default as the same as other plots.

- if `return_type` is `'axes'`, a `matplotlib Axes` containing the boxplot is returned.

- if `return_type` is `'both'`, a namedtuple containing the `matplotlib Axes` and `matplotlib Lines` is returned

When `subplots=True / by` is some column of the DataFrame:

- A dict of `return_type` is returned, where the keys are the columns of the DataFrame. The plot has a facet for each column of the DataFrame, with a separate box for each value of `by`.

Finally, when calling `boxplot` on a `Groupby` object, a dict of `return_type` is returned, where the keys are the same as the `Groupby` object. The plot has a facet for each key, with each facet containing a box for each column of the DataFrame.

23.2. Other Plots 653
In [47]: np.random.seed(1234)

In [48]: df_box = pd.DataFrame(np.random.randn(50, 2))

In [49]: df_box['g'] = np.random.choice(['A', 'B'], size=50)

In [50]: df_box.loc[df_box['g'] == 'B', 1] += 3

In [51]: bp = df_box.boxplot(by='g')

Compare to:

In [52]: bp = df_box.groupby('g').boxplot()
23.2.4 Area Plot

New in version 0.14.

You can create area plots with Series.plot and DataFrame.plot by passing kind='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 [53]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [54]: df.plot(kind='area');
To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

```python
In [55]: df.plot(kind='area', stacked=False);
```
23.2.5 Scatter Plot

New in version 0.13.

You can create scatter plots with DataFrame.plot by passing `kind='scatter'`. Scatter plot requires numeric columns for x and y axis. These can be specified by x and y keywords each.

In [56]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])

In [57]: df.plot(kind='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 [58]: ax = df.plot(kind='scatter', x='a', y='b',
                  color='DarkBlue', label='Group 1');
```

```
In [59]: df.plot(kind='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 [60]:** `df.plot(kind='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 [61]: df.plot(kind='scatter', x='a', y='b', s=df['c']*200);
```
23.2.6 Hexagonal Bin Plot

New in version 0.14.

You can create hexagonal bin plots with `DataFrame.plot()` and `kind='hexbin'`. Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

```
In [62]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [63]: df['b'] = df['b'] + np.arange(1000)
In [64]: df.plot(kind='hexbin', x='a', y='b', gridsize=25)
Out[64]: <matplotlib.axes._subplots.AxesSubplot at 0xad40eccc>
```
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.

```
In [65]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [66]: df['b'] = df['b'] + np.arange(1000)
In [67]: df['z'] = np.random.uniform(0, 3, 1000)
In [68]: df.plot(kind='hexbin', x='a', y='b', C='z', reduce_C_function=np.max, gridsize=25)
```

Out[68]: `<matplotlib.axes._subplots.AxesSubplot at 0xadadb8c>`
23.2.7 Pie plot

New in version 0.14.

You can create a pie plot with `DataFrame.plot()` or `Series.plot()` with `kind='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 [69]: series = pd.Series(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], name='series')
In [70]: series.plot(kind='pie', figsize=(6, 6))
```

```
Out[70]: <matplotlib.axes._subplots.AxesSubplot at 0xacd2dc4c>
```
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.

```
In [71]: df = pd.DataFrame(3 * np.random.rand(4, 2), index=['a', 'b', 'c', 'd'], columns=['x', 'y'])
In [72]: df.plot(kind='pie', subplots=True, figsize=(8, 4))
Out[72]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0xadae15ec>,
       <matplotlib.axes._subplots.AxesSubplot object at 0xad46114c>], 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.

```python
In [73]: series.plot(kind='pie', labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
...:          autopct='%.2f', fontsize=20, figsize=(6, 6))
...:
Out[73]: <matplotlib.axes._subplots.AxesSubplot at 0xadd0558c>
```
If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

In [74]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')

In [75]: series.plot(kind='pie', figsize=(6, 6))
Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0xadcea3ac>
See the matplotlib pie documentation for more.

### 23.3 Plotting with Missing Data

Pandas tries to be pragmatic about plotting DataFrames or Series that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.

<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.
23.4 Plotting Tools

These functions can be imported from pandas.tools.plotting and take a Series or DataFrame as an argument.

23.4.1 Scatter Matrix Plot

New in version 0.7.3.

You can create a scatter plot matrix using the scatter_matrix method in pandas.tools.plotting:

```
In [76]: from pandas.tools.plotting import scatter_matrix

In [77]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [78]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
```

```
Out[78]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0xad5d5ccc>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xac44724c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xac41ce8c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xac3b5f6c>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0xac37c1ac>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xac329b2c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xac2e7a2c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad5504ec>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0xad4c044c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad329b2c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad3c2c2c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad278ec>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0xad44ebac>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad397f8c>,
        <matplotlib.axes._subplots.AxesSubplot object at 0xad34d5ec>]], dtype=object)
```
23.4.2 Density Plot

New in version 0.8.0.

You can create density plots using the Series/DataFrame.plot and setting `kind='kde'`:

In [79]: ser = pd.Series(np.random.randn(1000))

In [80]: ser.plot(kind='kde')
Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0xa9af7dac>
23.4.3 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.

In [81]: from pandas.tools.plotting import andrews_curves

In [82]: data = pd.read_csv('data/iris.data')

In [83]: plt.figure()
Out[83]: <matplotlib.figure.Figure at 0xa9c6f8cc>

In [84]: andrews_curves(data, 'Name')
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0xa9c6f96c>
23.4.4 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 [85]: from pandas.tools.plotting import parallel_coordinates

In [86]: data = pd.read_csv('data/iris.data')

In [87]: plt.figure()
Out[87]: <matplotlib.figure.Figure at 0xa99a0d2c>

In [88]: parallel_coordinates(data, 'Name')
Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0xa99a57cc>
23.4.5 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 [89]: from pandas.tools.plotting import lag_plot

In [90]: plt.figure()
Out[90]: <matplotlib.figure.Figure at 0xa9baf52c>

In [91]: data = pd.Series(0.1 * np.random.rand(1000) +
                      0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))

In [92]: lag_plot(data)
Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0xa9ba90ec>
23.4.6 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.

In [93]: from pandas.tools.plotting import autocorrelation_plot

In [94]: plt.figure()
Out[94]: <matplotlib.figure.Figure at 0xa9ba12cc>

In [95]: data = pd.Series(0.7 * np.random.rand(1000) +
        ....: 0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
        ....:

In [96]: autocorrelation_plot(data)
Out[96]: <matplotlib.axes._subplots.AxesSubplot at 0xa9b78c8c>
23.4.7 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 [97]: from pandas.tools.plotting import bootstrap_plot

In [98]: data = pd.Series(np.random.rand(1000))

In [99]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[99]: <matplotlib.figure.Figure at 0xa967668c>
23.4.8 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 [100]: from pandas.tools.plotting import radviz
In [101]: data = pd.read_csv('data/iris.data')
In [102]: plt.figure()
Out[102]: <matplotlib.figure.Figure at 0xa9c2e30c>
In [103]: radviz(data, 'Name')
Out[103]: <matplotlib.axes._subplots.AxesSubplot at 0xa9206c8c>
23.5 Plot Formatting

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```python
In [104]: 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.

### 23.5.1 Controlling the Legend

You may set the legend argument to False to hide the legend, which is shown by default.

In [105]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))

In [106]: df = df.cumsum()

In [107]: df.plot(legend=False)

Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0xa8d18ccc>
23.5.2 Scales

You may pass `logy` to get a log-scale Y axis.

```
In [108]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [109]: ts = np.exp(ts.cumsum())

In [110]: ts.plot(logy=True)
```

Out[110]: <matplotlib.axes._subplots.AxesSubplot at 0xa8a67e6c>
See also the logx and loglog keyword arguments.

### 23.5.3 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the secondary_y keyword:

```python
In [111]: df.A.plot()
Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0xa8dc572c>
```

```python
In [112]: df.B.plot(secondary_y=True, style='g')
Out[112]: <matplotlib.axes._subplots.AxesSubplot at 0xa85a228c>
```
To plot some columns in a DataFrame, give the column names to the \texttt{secondary\_y} keyword:

\begin{verbatim}
In [113]: plt.figure()
Out[113]: <matplotlib.figure.Figure at 0xa85a960c>

In [114]: ax = df.plot(secondary\_y=['A', 'B'])

In [115]: ax.set_ylabel('CD scale')
Out[115]: <matplotlib.text.Text at 0xa894a96c>

In [116]: ax.right_ax.set_ylabel('AB scale')
Out[116]: <matplotlib.text.Text at 0xa89450ac>
\end{verbatim}
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 [117]: plt.figure()
Out[117]: <matplotlib.figure.Figure at 0xa8d0d06c>

In [118]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[118]: <matplotlib.axes._subplots.AxesSubplot at 0xa847c18c>
```
23.5.4 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 [119]: plt.figure()
Out[119]: <matplotlib.figure.Figure at 0xa80a4a0c>

In [120]: df.A.plot()
Out[120]: <matplotlib.axes._subplots.AxesSubplot at 0xa809e74c>
```
Using the `x_compat` parameter, you can suppress this behavior:

```python
In [121]: plt.figure()
Out[121]: <matplotlib.figure.Figure at 0xa7fde38c>

In [122]: df.A.plot(x_compat=True)
Out[122]: <matplotlib.axes._subplots.AxesSubplot at 0xa7fe640c>
```
If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plot_params` can be used in a `with statement`:

```
In [123]: plt.figure()
Out[123]: <matplotlib.figure.Figure at 0xa7fc0e2c>

In [124]: with pd.plot_params.use('x_compat', True):
   ....:     df.A.plot(color='r')
   ....:     df.B.plot(color='g')
   ....:     df.C.plot(color='b')
   ....:
```
23.5.5 Subplots

Each Series in a DataFrame can be plotted on a different axis with the `subplots` keyword:

```
In [125]: df.plot(subplots=True, figsize=(6, 6));
```
23.5.6 Using Layout and Targetting 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.

In [126]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
The above example is identical to using

```
In [127]: 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.

```
In [128]: fig, axes = plt.subplots(4, 4, figsize=(6, 6));
In [129]: plt.subplots_adjust(wspace=0.5, hspace=0.5);
In [130]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
In [131]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]
In [132]: df.plot(subplots=True, ax=target1, legend=False, sharex=False, sharey=False);
In [133]: (-df).plot(subplots=True, ax=target2, legend=False, sharex=False, sharey=False);
```
Another option is passing an `ax` argument to `Series.plot()` to plot on a particular axis:

In [134]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [135]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A');

In [136]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B');

In [137]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C');

In [138]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D');
23.5.7 Plotting With Error Bars

New in version 0.14.

Plotting with error bars is now supported in the `DataFrame.plot()` and `Series.plot()` methods.

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 `Mx2` array should be provided indicating lower and upper (or left and right) errors. For a `MxN` `DataFrame`, asymmetrical errors should be in a `Mx2xN` array.

Here is an example of one way to easily plot group means with standard deviations from the raw data.
# Generate the data
In [139]: ix3 = pd.MultiIndex.from_arrays([['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'], ['foo', 'foo', 'bar', 'bar', 'foo', 'foo', 'bar', 'bar']], names=['letter', 'word'])

In [140]: 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
In [141]: gp3 = df3.groupby(level=('letter', 'word'))

In [142]: means = gp3.mean()

In [143]: errors = gp3.std()

In [144]: means
Out[144]:

<table>
<thead>
<tr>
<th>letter</th>
<th>word</th>
<th>data1</th>
<th>data2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bar</td>
<td>3.5</td>
<td>6.0</td>
</tr>
<tr>
<td></td>
<td>foo</td>
<td>2.5</td>
<td>5.5</td>
</tr>
<tr>
<td>b</td>
<td>bar</td>
<td>2.5</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>foo</td>
<td>3.0</td>
<td>4.5</td>
</tr>
</tbody>
</table>

In [145]: errors
Out[145]:

<table>
<thead>
<tr>
<th>letter</th>
<th>word</th>
<th>data1</th>
<th>data2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bar</td>
<td>0.707107</td>
<td>1.414214</td>
</tr>
<tr>
<td></td>
<td>foo</td>
<td>0.707107</td>
<td>0.707107</td>
</tr>
<tr>
<td>b</td>
<td>bar</td>
<td>0.707107</td>
<td>0.707107</td>
</tr>
<tr>
<td></td>
<td>foo</td>
<td>1.414214</td>
<td>0.707107</td>
</tr>
</tbody>
</table>

# Plot
In [146]: fig, ax = plt.subplots()

In [147]: means.plot(yerr=errors, ax=ax, kind='bar')

Out[147]: <matplotlib.axes._subplots.AxesSubplot at 0xa631450c>
23.5.8 Plotting Tables

New in version 0.14.

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.

```
In [148]: fig, ax = plt.subplots(1, 1)

In [149]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

In [150]: ax.get_xaxis().set_visible(False)  # Hide Ticks

In [151]: df.plot(table=True, ax=ax)
Out[151]: <matplotlib.axes._subplots.AxesSubplot at 0xa6294eec>
```
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.

In [152]: fig, ax = plt.subplots(1, 1)
In [153]: ax.get_xaxis().set_visible(False)  # Hide Ticks
In [154]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[154]: <matplotlib.axes._subplots.AxesSubplot at 0xa61a244c>
Finally, there is a helper function `pandas.tools.plotting.table` to create a table from `DataFrame` and `Series`, and add it to a `matplotlib.Axes`. This function can accept keywords which `matplotlib.table` has.

```python
In [155]: from pandas.tools.plotting import table

In [156]: fig, ax = plt.subplots(1, 1)

In [157]: table(ax, np.round(df.describe(), 2),
              loc='upper right', colWidths=[0.2, 0.2, 0.2])

Out[157]: <matplotlib.table.Table at 0xa61db5ac>

In [158]: df.plot(ax=ax, ylim=(0, 2), legend=None)
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0xa615c3ec>
```
Note: You can get table instances on the axes using axes.tables property for further decorations. See the matplotlib table documentation for more.

23.5.9 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=

In [159]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
In [160]: df = df.cumsum()
In [161]: plt.figure()
Out[161]: <matplotlib.figure.Figure at 0xa5f062cc>
In [162]: df.plot(colormap='cubehelix')
Out[162]: <matplotlib.axes._subplots.AxesSubplot at 0xa5f0b0ac>

or we can pass the colormap itself

In [163]: from matplotlib import cm

In [164]: plt.figure()
Out[164]: <matplotlib.figure.Figure at 0xa591ee2c>

In [165]: df.plot(colormap=cm.cubehelix)
Out[165]: <matplotlib.axes._subplots.AxesSubplot at 0xa5e152ec>
Colormaps can also be used other plot types, like bar charts:

```python
In [166]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
In [167]: dd = dd.cumsum()
In [168]: plt.figure()
Out[168]: <matplotlib.figure.Figure at 0xa584e1ec>
In [169]: dd.plot(kind='bar', colormap='Greens')
Out[169]: <matplotlib.axes._subplots.AxesSubplot at 0xa58355cc>
```
Parallel coordinates charts:

In [170]: plt.figure()
Out[170]: <matplotlib.figure.Figure at 0xa55c730c>

In [171]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
Out[171]: <matplotlib.axes._subplots.AxesSubplot at 0xa55b79cc>
Andrews curves charts:

```
In [172]: plt.figure()
Out[172]: <matplotlib.figure.Figure at 0xa4fb712c>
```

```
In [173]: andrews_curves(data, 'Name', colormap='winter')
Out[173]: <matplotlib.axes._subplots.AxesSubplot at 0xa4fb7bec>
```
23.6 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.

**Note:** The speed up for large data sets only applies to pandas 0.14.0 and later.

In [174]: price = pd.Series(np.random.randn(150).cumsum(),
   ....:                   index=pd.date_range('2000-1-1', periods=150, freq='B'))
   ....:
In [175]: ma = pd.rolling_mean(price, 20)
In [176]: mstd = pd.rolling_std(price, 20)
In [177]: plt.figure()
Out[177]: <matplotlib.figure.Figure at 0xa4cc4eac>

In [178]: plt.plot(price.index, price, 'k')
Out[178]: [<matplotlib.lines.Line2D at 0xa490d18c>]

In [179]: plt.plot(ma.index, ma, 'b')
Out[179]: [<matplotlib.lines.Line2D at 0xa4ac502c>]

In [180]: plt.fill_between(mstd.index, ma-2*mstd, ma+2*mstd, color='b', alpha=0.2)
Out[180]: <matplotlib.collections.PolyCollection at 0xa4ac580c>

23.7 Trellis plotting interface

**Warning:** 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. The docs below include some example on how to convert your existing code to seaborn.

**Note:** The tips data set can be downloaded here. Once you download it execute
We import the rplot API:

```python
In [181]: import pandas.tools.rplot as rplot
```

### 23.7.1 Examples

RPlot was an API for producing Trellis plots. These plots allow you to arrange data in a rectangular grid by values of certain attributes. In the example below, data from the tips data set is arranged by the attributes ‘sex’ and ‘smoker’. Since both of those attributes can take on one of two values, the resulting grid has two columns and two rows. A histogram is displayed for each cell of the grid.

```python
In [182]: plt.figure()
Out[182]: <matplotlib.figure.Figure at 0xa474f76c>

In [183]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [184]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [185]: plot.add(rplot.GeomHistogram())

In [186]: plot.render(plt.gcf())
Out[186]: <matplotlib.figure.Figure at 0xa474f76c>
```
A similar plot can be made with `seaborn` using the `FacetGrid` object, resulting in the following image:

```python
import seaborn as sns
g = sns.FacetGrid(tips_data, row="sex", col="smoker")
g.map(plt.hist, "total_bill")
```
Example below is the same as previous except the plot is set to kernel density estimation. A seaborn example is included beneath.

```python
In [187]: plt.figure()
Out[187]: <matplotlib.figure.Figure at 0xa475b6cc>

In [188]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [189]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [190]: plot.add(rplot.GeomDensity())

In [191]: plot.render(plt.gcf())
Out[191]: <matplotlib.figure.Figure at 0xa475b6cc>
```
g = sns.FacetGrid(tips_data, row="sex", col="smoker")
g.map(sns.kdeplot, "total_bill")
The plot below shows that it is possible to have two or more plots for the same data displayed on the same Trellis grid cell.

```python
In [192]: plt.figure()
Out[192]: <matplotlib.figure.Figure at 0xa45f6cac>

In [193]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [194]: plot.add(rplot.TrellisGrid(["sex", 'smoker']))

In [195]: plot.add(rplot.GeomScatter())

In [196]: plot.add(rplot.GeomPolyFit(degree=2))

In [197]: plot.render(plt.gcf())
Out[197]: <matplotlib.figure.Figure at 0xa45f6cac>
```
A seaborn equivalent for a simple scatter plot:

```python
g = sns.FacetGrid(tips_data, row="sex", col="smoker")
g.map(plt.scatter, "total_bill", "tip")
```
and with a regression line, using the dedicated seaborn regplot function:

g = sns.FacetGrid(tips_data, row="sex", col="smoker", margin_titles=True)
g.map(sns.regplot, "total_bill", "tip", order=2)
Below is a similar plot but with 2D kernel density estimation plot superimposed, followed by a seaborn equivalent:

```python
In [198]: plt.figure()
Out[198]: <matplotlib.figure.Figure at 0xa4273b2c>

In [199]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [200]: plot.add(rplot.TrellisGrid(["sex", "smoker"]))

In [201]: plot.add(rplot.GeomScatter())

In [202]: plot.add(rplot.GeomDensity2D())

In [203]: plot.render(plt.gcf())
Out[203]: <matplotlib.figure.Figure at 0xa4273b2c>
```
```python
g = sns.FacetGrid(tips_data, row="sex", col="smoker")
g.map(plt.scatter, "total_bill", "tip")
g.map(sns.kdeplot, "total_bill", "tip")
```
It is possible to only use one attribute for grouping data. The example above only uses ‘sex’ attribute. If the second grouping attribute is not specified, the plots will be arranged in a column.

In [204]: plt.figure()
Out[204]: <matplotlib.figure.Figure at 0xa420f20c>

In [205]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [206]: plot.add(rplot.TrellisGrid(['sex', '.']))

In [207]: plot.add(rplot.GeomHistogram())

In [208]: plot.render(plt.gcf())
Out[208]: <matplotlib.figure.Figure at 0xa420f20c>
If the first grouping attribute is not specified the plots will be arranged in a row.

In [209]: plt.figure()
Out[209]: <matplotlib.figure.Figure at 0xa3e6e8ec>

In [210]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [211]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [212]: plot.add(rplot.GeomHistogram())

In [213]: plot.render(plt.gcf())
Out[213]: <matplotlib.figure.Figure at 0xa3e6e8ec>
In seaborn, this can also be done by only specifying one of the row and col arguments.

In the example below the colour and shape of the scatter plot graphical objects is mapped to ‘day’ and ‘size’ attributes respectively. You use scale objects to specify these mappings. The list of scale classes is given below with initialization arguments for quick reference.

```python
In [214]: plt.figure()
Out[214]: <matplotlib.figure.Figure at 0xa3de1f6c>

In [215]: plot = rplot.RPlot(tips_data, x='tip', y='total_bill')

In [216]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [217]: plot.add(rplot.GeomPoint(size=80.0, colour=rplot.ScaleRandomColour('day'), shape=rplot.ScaleShape('size'), alpha=1.0))

In [218]: plot.render(plt.gcf())
Out[218]: <matplotlib.figure.Figure at 0xa3de1f6c>
```
This can also be done in `seaborn`, at least for 3 variables:

```python
g = sns.FacetGrid(tips_data, row="sex", col="smoker", hue="day")
g.map(plt.scatter, "tip", "total_bill")
g.add_legend()
```
The pandas I/O API is 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_msgpack` (experimental)
- `read_html`
- `read_gbq` (experimental)
- `read_stata`
- `read_clipboard`
- `read_pickle`

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_msgpack` (experimental)
- `to_html`
- `to_gbq` (experimental)
- `to_stata`
- `to_clipboard`
- `to_pickle`

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.

They can take a number of arguments:

- filepath_or_buffer: Either a string path to a file, URL (including http, ftp, and S3 locations), or any object with a read method (such as an open file or StringIO).
- sep or delimiter: A delimiter / separator to split fields on. With sep=None, read_csv will try to infer the delimiter automatically in some cases by “sniffing”. The separator may be specified as a regular expression; for instance you may use ‘\s*’ to indicate a pipe plus arbitrary whitespace.
- delim_whitespace: Parse whitespace-delimited (spaces or tabs) file (much faster than using a regular expression)
- compression: decompress ‘gzip’ and ‘bz2’ formats on the fly. Set to ‘infer’ (the default) to guess a format based on the file extension.
- dialect: string or csv.Dialect instance to expose more ways to specify the file format
- dtype: A data type name or a dict of column name to data type. If not specified, data types will be inferred. (Unsupported with engine='python')
- header: row number(s) to use as the column names, and the start of the data. Defaults 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 are skipped). Note that this parameter ignores commented lines and empty lines if skip_blank_lines=True (the default), so header=0 denotes the first line of data rather than the first line of the file.
- skip_blank_lines: whether to skip over blank lines rather than interpreting them as NaN values
- skiprows: A collection of numbers for rows in the file to skip. Can also be an integer to skip the first n rows
- index_col: column number, column name, or list of column numbers/names, to use as the index (row labels) of the resulting DataFrame. By default, it will number the rows without using any column, unless there is one more data column than there are headers, in which case the first column is taken as the index.
- names: List of column names to use as column names. To replace header existing in file, explicitly pass header=0.
- na_values: optional list of strings to recognize as NaN (missing values), either in addition to or in lieu of the default set.
- true_values: list of strings to recognize as True
- false_values: list of strings to recognize as False
- keep_default_na: whether to include the default set of missing values in addition to the ones specified in na_values
- parse_dates: if True then index will be parsed as dates (False by default). You can specify more complicated options to parse a subset of columns or a combination of columns into a single date column (list of ints or names, list of lists, or dict) [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column [[1, 3]] -> combine columns 1 and 3 and parse as a single date column {'foo': [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’
• keep_date_col: if True, then date component columns passed into parse_dates will be retained in the output (False by default).

• date_parser: function to use to parse strings into datetime objects. If parse_dates is True, it defaults to the very robust dateutil.parser. Specifying this implicitly sets parse_dates as True. You can also use functions from community supported date converters from date_converters.py

• dayfirst: if True then uses the DD/MM international/European date format (This is False by default)

• thousands: specifies the thousands separator. If not None, this character will be stripped from numeric dtypes. However, if it is the first character in a field, that column will be imported as a string. In the PythonParser, if not None, then parser will try to look for it in the output and parse relevant data to numeric dtypes. Because it has to essentially scan through the data again, this causes a significant performance hit so only use if necessary.

• lineterminator: string (length 1), default None, Character to break file into lines. Only valid with C parser

• quotechar: string. The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

• quoting: int, Controls whether quotes should be recognized. Values are taken from csv.QUOTE_* values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.

• skipinitialspace: boolean, default False, Skip spaces after delimiter

• escapechar: string, to specify how to escape quoted data

• comment: 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, fully commented lines are ignored by the parameter header but not by skiprows. For example, if comment='#', parsing '#emptyn1,2,3na,b,c' with header=0 will result in '1,2,3' being treated as the header.

• nrows: Number of rows to read out of the file. Useful to only read a small portion of a large file

• iterator: If True, return a TextFileReader to enable reading a file into memory piece by piece

• chunksize: An number of rows to be used to “chunk” a file into pieces. Will cause an TextFileReader object to be returned. More on this below in the section on iterating and chunking

• skip_footer: number of lines to skip at bottom of file (default 0) (Unsupported with engine='c')

• converters: a dictionary of functions for converting values in certain columns, where keys are either integers or column labels

• encoding: a string representing the encoding to use for decoding unicode data, e.g. ‘utf-8’ or ‘latin-1’. Full list of Python standard encodings

• verbose: show number of NA values inserted in non-numeric columns

• squeeze: if True then output with only one column is turned into Series

• error_bad_lines: if False then any lines causing an error will be skipped bad lines

• usecols: a subset of columns to return, results in much faster parsing time and lower memory usage.

• mangle_dupe_cols: boolean, default True, then duplicate columns will be specified as ‘X.0’...’X.N’, rather than ‘X’...’X’

• tupleize_cols: boolean, default False, if False, convert a list of tuples to a multi-index of columns, otherwise, leave the column index as a list of tuples

• 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.
Consider a typical CSV file containing, in this case, some time series data:

```python
In [1]: print(open('foo.csv').read())
date,A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

The default for `read_csv` is to create a DataFrame with simple numbered rows:

```python
In [2]: pd.read_csv('foo.csv')
Out[2]:
   date    A    B    C
0 20090101  a  1  2
1 20090102  b  3  4
2 20090103  c  4  5
```

In the case of indexed data, you can pass the column number or column name you wish to use as the index:

```python
In [3]: pd.read_csv('foo.csv', index_col=0)
Out[3]:
       A    B    C
date
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

```python
In [4]: pd.read_csv('foo.csv', index_col='date')
Out[4]:
       A    B    C
date
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

You can also use a list of columns to create a hierarchical index:

```python
In [5]: pd.read_csv('foo.csv', index_col=[0, 'A'])
Out[5]:
       B    C
date
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

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:

```python
In [6]: 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 [7]: dia = csv.excel()

In [8]: dia.quoting = csv.QUOTE_NONE

In [9]: pd.read_csv(StringIO(data), dialect=dia)
Out[9]:
label1   label2   label3
index1      a   c   e
index2       b   d   f

All of the dialect options can be specified separately by keyword arguments:

In [10]: data = 'a,b,c~1,2,3~4,5,6'

In [11]: pd.read_csv(StringIO(data), lineterminator='~')
Out[11]:
a   b   c
0   1   2
1   4   5

Another common dialect option is skipinitialspace, to skip any whitespace after a delimiter:

In [12]: data = 'a, b, c

1, 2, 3

4, 5, 6'

In [13]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [14]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[14]:
a   b   c
0   1   2
1   4   5

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.

24.1.1 Specifying column data types

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

In [15]: data = 'a,b,c

1,2,3

4,5,6

7,8,9'

In [16]: print(data)
a, b, c
1, 2, 3
4, 5, 6
7, 8, 9

In [17]: df = pd.read_csv(StringIO(data), dtype=object)

In [18]: df
Out[18]:
a   b   c
0   1   2
1   4   5

24.1. CSV & Text files 719
2 7 8 9

In [19]: df['a'][0]
Out[19]: '1'

In [20]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})

In [21]: df.dtypes
Out[21]:
    a  int64
    b  object
    c  float64
   dtype: object

Note: The dtype option is currently only supported by the C engine. Specifying dtype with engine other than 'c' raises a ValueError.

24.1.2 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 [22]: data = 'a,b,c
1,2,3
4,5,6
7,8,9'

In [23]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [24]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[24]:
    foo  bar  baz
   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 [25]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [26]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[26]:
    foo  bar  baz
   0  1  2  3
   1  4  5  6
   2  7  8  9

In [27]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[27]:
    foo  bar  baz
   0  a  b  c
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 [28]: data = 'skip this skip it
a,b,c
1,2,3
4,5,6
7,8,9'

In [29]: pd.read_csv(StringIO(data), header=1)
Out[29]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

### 24.1.3 Filtering columns (`usecols`)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names or position numbers:

```python
In [30]: data = 'a,b,c,d
1,2,3,foo
4,5,6,bar
7,8,9,baz'

In [31]: pd.read_csv(StringIO(data))
Out[31]:
   a  b  c  d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [32]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[32]:
   b  d
0  2  foo
1  5  bar
2  8  baz

In [33]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[33]:
   a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz
```

### 24.1.4 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. Both of these are API changes introduced in version 0.15.

```python
In [34]: data = '
a,b,c

# commented line
1,2,3

4,5,6'

In [35]: print(data)
a,b,c
1,2,3
```

---

**24.1. CSV & Text files**

---

721
4, 5, 6

# commented line
In [36]: pd.read_csv(StringIO(data), comment='#')
Out[36]:
     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 [37]: data = 'a,b,c

1,2,3

4,5,6'

In [38]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[38]:
     a  b  c
0  NaN NaN NaN
1    1  2  3
2  NaN NaN NaN
3  NaN NaN NaN
4    4  5  6
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):

```python
In [39]: data = '#comment
   ...: a,b,c
   ...: A,B,C
   ...: 1,2,3

In [40]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[40]:
   A  B  C
0  1  2  3

In [41]: data = 'A,B,C
   ...: #comment
   ...: a,b,c
   ...: 1,2,3

In [42]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[42]:
   a  b  c
0  1  2  3
If both header and skiprows are specified, header will be relative to the end of skiprows. For example:

```python
In [43]: data = '# empty
   ...: # second empty line
   ...: # third empty
   ...: X,Y,Z
   ...: 1,2,3
   ...: A,B,C
   ...: 1,2.,4.
   ...: 5.,NaN,10.0

In [44]: 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 [45]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[45]:
   A  B  C
0  1  2  4
1  5  NaN 10

24.1.5 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:

```python
In [46]: data = b'word,length
   ...: Träumen,7
   ...: Grüße,5'.decode('utf8').encode('latin-1')

In [47]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [48]: df
Out[48]:
       word   length
0   Träumen     7
1    Grüße      5

In [49]: df['word'][1]
```
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

24.1.6 Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the DataFrame’s row names:

In [50]: data = 'a,b,c
4,apple,bat,5.7
8,orange,cow,10'

In [51]: pd.read_csv(StringIO(data))
Out[51]:
   a   b   c
4 apple bat 5.7
8 orange cow 10.0

In [52]: data = 'index,a,b,c
4,apple,bat,5.7
8,orange,cow,10'

In [53]: pd.read_csv(StringIO(data), index_col=0)
Out[53]:
   a   b   c
index
4 apple bat 5.7
8 orange cow 10.0

Ordinarily, you can achieve this behavior using the index_col option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass index_col=False:

In [54]: data = 'a,b,c
4,apple,bat,
8,orange,cow,'

In [55]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [56]: pd.read_csv(StringIO(data))
Out[56]:
   a   b   c
4 apple bat NaN
8 orange cow NaN

In [57]: pd.read_csv(StringIO(data), index_col=False)
Out[57]:
   a   b   c
   0  4 apple bat
   1  8 orange cow

24.1.7 Specifying Date Columns

To better facilitate working with datetime data, read_csv() and read_table() uses the keyword arguments parse_dates and date_parser to allow users to specify a variety of columns and date/time formats to turn the input text data into datetime objects.
The simplest case is to just pass in `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [58]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)
```

```
In [59]: df
Out[59]:
    A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

# These are python datetime objects
```
In [60]: df.index
Out[60]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[ns]', name='date', freq=None, tz=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:

```
In [61]: print(open('tmp.csv').read())
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900
```

```
In [62]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])
```

```
In [63]: df
Out[63]:
   1_2  1_3  0   4
 0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD  0.81
 1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD  0.01
 2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
 3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
 4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
 5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [64]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
...:       keep_date_col=True)
```

```
In [65]: df
Out[65]:
   1_2  1_3  0   1   2  
 0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 19990127 19:00:00
 1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 19990127 20:00:00
 2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD 19990127 21:00:00
 3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD 19990127 21:00:00
 4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD 19990127 22:00:00
 5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD 19990127 23:00:00
```
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 [66]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [67]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [68]: df
```

```
Out[68]:
```

```
nominal  actual  0  4
0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

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 [69]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [70]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec, index_col=0)  # index is the nominal column

In [71]: df
```

```
Out[71]:
```

```
nominal  actual  0  4
0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

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 than as an index on the resulting frame.
### 24.1.8 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 [72]: val = '0.3066101993807095471566981359501369297504425048828125'

In [73]: data = 'a,b,c
1,2,{0}'.format(val)

In [74]: abs(pd.read_csv(StringIO(data), engine='c', float_precision=None)['c'][0] - float(val))
Out[74]: 0.0

In [75]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='high')['c'][0] - float(val))
Out[75]: 5.5511151231257827e-17

In [76]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='round_trip')['c'][0] - float(val))
Out[76]: 0.0
```

### 24.1.9 Date Parsing Functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```python
In [77]: import pandas.io.date_converters as conv

In [78]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
                    date_parser=conv.parse_date_time)

In [79]: df
Out[79]:
   nominal  actual   0   4
0  1999-01-27 19:00:00 1999-01-27 18:56:00 KORD  0.81
1  1999-01-27 20:00:00 1999-01-27 19:56:00 KORD  0.01
2  1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
3  1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
4  1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
5  1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

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` contains 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.

### 24.1.10 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.

```python
# Try to infer the format for the index column
In [80]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True, ...:
                           infer_datetime_format=True)
   ....:

In [81]: df
Out[81]:
   A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

### 24.1.11 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 [82]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
```
2/6/2000, 10, b
3/6/2000, 15, c

In [83]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[83]:
   date  value  cat
0  2000-01-06   5   a
1  2000-02-06  10   b
2  2000-03-06  15   c

In [84]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[84]:
   date  value  cat
0  2000-06-01   5   a
1  2000-06-02  10   b
2  2000-06-03  15   c

### 24.1.12 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

In [85]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [86]: df = pd.read_csv('tmp.csv', sep='|')

In [87]: df
Out[87]:
   ID      level  category
0 Patient1  123,000       x
1 Patient2   23,000       y
2 Patient3 1,234,018      z

In [88]: df.level.dtype
Out[88]: dtype('O')

The `thousands` keyword allows integers to be parsed correctly

In [89]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [90]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [91]: df
Out[91]:
   ID  level  category
0 Patient1 123000       x
1 Patient2  23000       y
2 Patient3 1234018 z

In [92]: df.level.dtype
Out[92]: dtype('int64')

24.1.13 NA Values

To control which values are parsed as missing values (which are signified by NaN), specify a list of strings in na_values. If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0,5] are recognized as NaN).

To completely override the default values that are recognized as missing, specify keep_default_na=False. The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A','N/A', 'NA', '#NA', 'NULL', 'NaN', '-NaN', 'nan', '-nan'].

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

24.1.14 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.

24.1.15 Comments

Sometimes comments or meta data may be included in a file:

In [93]: 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 parse includes the comments in the output:

In [94]: df = pd.read_csv('tmp.csv')

In [95]: df
Out[95]:
   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:

```python
In [96]: df = pd.read_csv('tmp.csv', comment='#')
```

```python
In [97]: df
Out[97]:
       ID  level category
0  Patient1  123000       x
1  Patient2  23000        y
2  Patient3  1234018      z
```

### 24.1.16 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a Series:

```python
In [98]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018
In [99]: output = pd.read_csv('tmp.csv', squeeze=True)
In [100]: output
Out[100]:
Patient1  123000
Patient2  23000
Patient3  1234018
Name: level, dtype: int64
In [101]: type(output)
Out[101]: pandas.core.series.Series
```

### 24.1.17 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:

```python
In [102]: data= 'a,b,c
1,Yes,2
3,No,4'
In [103]: print(data)
a,b,c
1,Yes,2
3,No,4
In [104]: pd.read_csv(StringIO(data))
Out[104]:
a  b  c
0 1   Yes 2
1 3  No  4
In [105]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[105]:
a  b  c
0 1   True 2
1 3  False 4
```
24.1.18 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:

```python
In [27]: data = 'a,b,c

1,2,3

4,5,6,7

8,9,10'
```

```python
In [28]: pd.read_csv(StringIO(data))
```

```
CParseError Traceback (most recent call last)
CParseError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4
```

You can elect to skip bad lines:

```python
In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
```

```
Skipping line 3: expected 3 fields, saw 4
```

```
an b c
0 1 2 3
1 8 9 10
```

24.1.19 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 [106]: data = 'a,b

"hello, "Bob"", nice to see you",5'

In [107]: print(data)
an, b
"hello, "Bob"", nice to see you",5
```

```python
In [108]: pd.read_csv(StringIO(data), escapechar='\')
```

```
an b
0 hello, "Bob", nice to see you 5
```

24.1.20 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:

- `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.

Consider a typical fixed-width data file:

```python
In [109]: 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 [110]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [111]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [112]: df
Out[112]:
      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
```

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:

```python
#Widths are a list of integers
In [113]: widths = [6, 14, 13, 10]

In [114]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [115]: df
Out[115]:
      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.

New in version 0.13.0.

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).

```python
In [116]: df = pd.read_fwf('bar.csv', header=None, index_col=0)
```

```python
In [117]: df
Out[117]:
      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
```

### 24.1.21 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

24.1. CSV & Text files 733
In [118]: `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 [119]: pd.read_csv('foo.csv')
Out[119]:
   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 [120]: df = pd.read_csv('foo.csv', parse_dates=True)

In [121]: df.index
Out[121]: DatetimeIndex([‘2009-01-01’, ‘2009-01-02’, ‘2009-01-03’], dtype=’datetime64[ns]’, freq=None, tz=None)

24.1.22 Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

In [122]: `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 [123]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])

In [124]: df
Out[124]:
  zit  xit
year indiv
1977 A   1.20 0.60
   B   1.50 0.50
   C   1.70 0.80
1978 A   0.20 0.06
   B   0.70 0.20

734 Chapter 24. IO Tools (Text, CSV, HDF5, ...
24.1.23 Reading columns with a MultiIndex

By specifying list of row locations for the header argument, you can read in a MultiIndex for the columns. Specifying non-consecutive rows will skip the intervening rows. In order to have the pre-0.13 behavior of tupleizing columns, specify tupleize_cols=True.

In [126]: from pandas.util.testing import makeCustomDataframe as mkdf

In [127]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)

In [128]: df.to_csv('mi.csv')

In [129]: print(open('mi.csv').read())

C0, C_10_g0, C_10_g1, C_10_g2
C1, C_11_g0, C_11_g1, C_11_g2
C2, C_12_g0, C_12_g1, C_12_g2
C3, C_13_g0, C_13_g1, C_13_g2
R0, R1, ,
R_10_g0, R_11_g0, R0C0, R0C1, R0C2
R_10_g1, R_11_g1, R1C0, R1C1, R1C2
R_10_g2, R_11_g2, R2C0, R2C1, R2C2
R_10_g3, R_11_g3, R3C0, R3C1, R3C2
R_10_g4, R_11_g4, R4C0, R4C1, R4C2

In [130]: pd.read_csv('mi.csv', header=[0,1,2,3], index_col=[0,1])

Out[130]:
C0 C_10_g0 C_10_g1 C_10_g2
C1 C_11_g0 C_11_g1 C_11_g2
C2 C_12_g0 C_12_g1 C_12_g2
C3 C_13_g0 C_13_g1 C_13_g2
R0 R1
R_10_g0 R_11_g0 R0C0 R0C1 R0C2
R_10_g1 R_11_g1 R1C0 R1C1 R1C2
R_10_g2 R_11_g2 R2C0 R2C1 R2C2
R_10_g3 R_11_g3 R3C0 R3C1 R3C2

24.1. CSV & Text files
Starting in 0.13.0, read_csv will be able to interpret a more common format of multi-columns indices.

```
In [131]: 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 [132]: pd.read_csv('mi2.csv',header=[0,1],index_col=0)
Out[132]:
   a  b  c
one q  r  s  t  u  v
   1  2  3  4  5  6
two 7  8  9 10 11 12
```

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.

24.1.24 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 [133]: print(open('tmp2.sv').read())
:0:1:2:3
0: 0.469112 299907: -0.28263344329: -1.50905850317: -1.13563237102
1: 1.21211202502: -1.73214649053: 0.119208711297: -1.04423596628
2: -0.861848963348: -2.10456921889: -0.494929274069: 0.07180380704
3: 0.72155162244: -0.70677113363: -1.03957498511: 0.271859885543
4: -0.424972329789: 0.567020349794: 0.276232019278: -1.08740069129
5: -0.673689708088: 0.113648409689: -1.47842655244: 0.524987667115
6: 0.404705214806: 0.57704598592: -1.71500201611: -1.03926848351
7: -0.370646858236: -1.5789225064: -1.34431181273: 0.844885141425
8: 1.07576978372: -0.10904997528: 1.64356307036: -1.469387959543
9: 0.357020564133: -0.67460010373: -1.77690371697: -0.96891381244
```

```
In [134]: pd.read_csv('tmp2.sv', sep=None, engine='python')
Out[134]:
     Unnamed  0      1      2      3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -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 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647  0.577046 -1.776904 -0.968914
8  1.075769 -1.157892 -1.344312  0.844885
9  0.357021 -0.674600 -1.776904  0.968914
```
24.1.25 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 [135]: print(open('tmp.sv').read())
|0|1|2|3
0|0.469112299907|-0.28286344329|-1.50905850317|-1.13563237102
1|1.21211202502|-1.73214649053|0.119208711297|-1.04423596628
2|-0.861848963348|-2.10456921889|-0.494929274069|1.07180380704
3|0.721555162244|-0.70677113363|-1.03957498511|0.271859885543
4|-0.424972329789|0.567020349794|0.276232019278|-1.08740069129
5|-0.673689708088|0.113648409689|-1.47842655244|0.524987676115
6|0.40470521868|0.57704598592|-1.71500201611|-1.03926848351
7|-0.370646858236|-1.15789225064|-1.34431181273|0.844885141425
8|1.07576978372|-0.10904997528|1.64356307036|-1.46938795954
9|0.357020564133|-0.67460010373|-1.77690371697|-0.968913812447
```

```
In [136]: table = pd.read_table('tmp.sv', sep='|')
```

```
In [137]: table
Out[137]:
   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
4          4 -0.424972  0.567020   0.276233  -1.087401
5          5  0.673690  0.113648  -1.478427  0.524988
6          6  0.404705  0.577046  -1.715002  -1.039268
7          7 -0.370647 -1.157892  -1.344312  0.844885
8          8  1.075770 -0.109050  1.643563  -1.469388
9          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 [138]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)
```

```
In [139]: reader
Out[139]: <pandas.io.parsers.TextFileReader at 0xa561bcec>
```

```
In [140]: for chunk in reader:

   .....:
       print(chunk)
       .....:
   Unnamed: 0   0   1   2   3
0          0  0.469112 -0.282863 -1.509059 -1.135632
1          1  1.212112 -0.173215  0.119209 -1.044236
2          2 -0.861849 -2.104569  -0.494929  1.071804
3          3  0.721555 -0.706771  -1.039575  0.271860
   Unnamed: 0   0   1   2   3
0          4 -0.424972  0.567020   0.276233  -1.087401
1          5  0.673690  0.113648  -1.478427  0.524988
2          6  0.404705  0.577046  -1.715002  -1.039268
3          7 -0.370647 -1.157892  -1.344312  0.844885
   Unnamed: 0   0   1   2   3
0          8  1.075770 -0.109050  1.643563  -1.469388
1          9  0.357021 -0.674600  -1.776904  -0.968914
```

24.1. CSV & Text files
Specifying `iterator=True` will also return the `TextFileReader` object:

```python
In [141]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)
```

```python
In [142]: reader.get_chunk(5)
Out[142]:
     Unnamed  0  1  2  3
0   0.469112  0 -0.282863 -1.509059 -1.135632
1   1.212112 -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
```

### 24.1.26 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)
- `skip_footer`
- `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'`.

### 24.1.27 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)
- `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

24.1.28 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>split</td>
<td>dict like {index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</td>
</tr>
<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>

- date_format: string, type of date conversion, ‘epoch’ for timestamp, ‘iso’ for ISO8601.
- double_precision: The number of decimal places to use when encoding floating point values, default 10.
- force_ascii: force encoded string to be ASCII, default True.
- date_unit: The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’ or ‘ns’ for seconds, milliseconds, microseconds and nanoseconds respectively. Default ‘ms’.
- default_handler: The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.

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 [143]: dfj = DataFrame(randn(5, 2), columns=list('AB'))

In [144]: json = dfj.to_json()

In [145]: json
Out[145]: '{"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}}'

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 [146]: dfjo = DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
   ....:     columns=list('ABC'), index=list('xyz'))

In [147]: dfjo
Out[147]:
A  B  C
x 1 4 7
y 2 5 8
z 3 6 9

In [148]: sjo = Series(dict(x=15, y=16, z=17), name='D')

In [149]: sjo
Out[149]:
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 [150]: dfjo.to_json(orient="columns")
Out[150]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'
```

Index oriented (the default for Series) similar to column oriented but the index labels are now primary:

```
In [151]: dfjo.to_json(orient="index")
```

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 [153]: 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 [155]: dfjo.to_json(orient="values")
Out[155]: '[[1,4,7],[2,5,8],[3,6,9]]'
```

Split oriented serializes to a JSON object containing separate entries for values, index and columns. Name is also included for Series:

```
In [156]: dfjo.to_json(orient="split")
Out[156]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,9]]}'
```

Note: Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the split option as it uses ordered containers.

Date Handling

Writing in ISO date format

```
In [158]: dfd = DataFrame(randn(5, 2), columns=list('AB'))
In [159]: dfd['date'] = Timestamp('20130101')
In [160]: dfd = dfd.sort_index(1, ascending=False)
In [161]: json = dfd.to_json(date_format='iso')
In [162]: json
```

Writing in ISO date format, with microseconds

```
In [163]: json = dfd.to_json(date_format='iso', microseconds=True)
In [164]: json
```

24.2. JSON
In [163]: json = dfd.to_json(date_format='iso', date_unit='us')

In [164]: json
Out[164]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00.000000Z","2":"2013-01-01T00:00.000000Z","3":"2013-01-01T00:00.000000Z","4":"2013-01-01T00:00.000000Z"}}'

Epoch timestamps, in seconds

In [165]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [166]: json
Out[166]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":1356998400},"B":...}

Writing to a file, with a date index and a date column

In [167]: dfj2 = dfj.copy()

In [168]: dfj2['date'] = Timestamp('20130101')

In [169]: dfj2['ints'] = list(range(5))

In [170]: dfj2['bools'] = True

In [171]: dfj2.index = date_range('20130101', periods=5)

In [172]: dfj2.to_json('test.json')

In [173]: open('test.json').read()
Out[173]: '{"A":{"0":1.2945235903,"1":1357084800000,"2":1357171200000,"3":1357257600000,"4":1357344000000},"B":...}

Fallback Behavior

If the JSON serializer cannot handle the container contents directly it will fallback in the following manner:

- if a `toDict` method is defined by the unrecognised object then that will be called and its returned `dict` will be JSON serialized.
- if a `default_handler` has been passed to `to_json` that will be called to convert the object.
- otherwise an attempt is made to convert the object to a `dict` by parsing its contents. However if the object is complex this will often fail with an `OverflowError`.

Your best bet when encountering `OverflowError` during serialization is to specify a `default_handler`. For example `timedelta` can cause problems:

In [141]: from datetime import timedelta

In [142]: dftd = DataFrame([timedelta(23), timedelta(seconds=5), 42])

In [143]: dftd.to_json()

---------------------------------------------------------------------------
OverflowError Traceback (most recent call last)
OverflowError: Maximum recursion level reached

which can be dealt with by specifying a simple `default_handler`:

In [144]: dftd.to_json(default_handler=str)
Out[144]: '{"0":("0":1987200000,"1":5000,"2":42)}'
In [175]: def my_handler(obj):
       .....:     return obj.total_seconds()
       .....:

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td>dict like {index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</td>
</tr>
<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.

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parseable.

If a non-default `orient` was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see `Orient Options` for an overview.
Data Conversion

The default of convert_axes=True, dtype=True, and convert_dates=True will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to dtype. convert_axes should only be set to False if you need to preserve string-like numbers (e.g. ‘1’, ‘2’) in an axes.

Note: Large integer values may be converted to dates if convert_dates=True and the data and / or column labels appear ‘date-like’. The exact threshold depends on the date_unit specified.

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 [176]: pd.read_json(json)
Out[176]:
    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:

```
In [177]: pd.read_json('test.json')
Out[177]:
    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):

```
In [178]: pd.read_json('test.json', dtype=object).dtypes
Out[178]:
    A   object
    B   object
  bools   object
    date   object
     ints   object
         dtype: object
```

Specify dtypes for conversion:

```
In [179]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[179]:
    A      float32
    B      float64
  bools   int8
    date  datetime64[ns]
```
Preserve string indices:

```python
In [180]: si = DataFrame(np.zeros((4, 4)),
......:     columns=list(range(4)),
......:     index=[str(i) for i in range(4)])
......:
In [181]: si
Out[181]:
     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 [182]: si.index
Out[182]: Index([u'0', u'1', u'2', u'3'], dtype='object')
In [183]: si.columns
Out[183]: Int64Index([0, 1, 2, 3], dtype='int64')
In [184]: json = si.to_json()
In [185]: sij = pd.read_json(json, convert_axes=False)
In [186]: sij
Out[186]:
     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 [187]: sij.index
Out[187]: Index([u'0', u'1', u'2', u'3'], dtype='object')
In [188]: sij.columns
Out[188]: Index([u'0', u'1', u'2', u'3'], dtype='object')

Dates written in nanoseconds need to be read back in nanoseconds:

```python
In [189]: json = dfj2.to_json(date_unit='ns')
# Try to parse timestamps as milliseconds -> Won't Work
In [190]: dfju = pd.read_json(json, date_unit='ms')
In [191]: dfju
Out[191]:
   A          B  bools  date  ints
0  1.356998e+18 -1.294524  True 1356998400000000000  0
1  1.357085e+18  0.276662  True 1356998400000000000  1
2  1.357171e+18 -0.013960 -0.472035 True 1356998400000000000  2
3  1.357258e+18 -0.006154 -0.923061 True 1356998400000000000  3
4  1.357344e+18  0.895717  0.805244 True 1356998400000000000  4

# Let pandas detect the correct precision
```
In [192]: dfju = pd.read_json(json)

In [193]: dfju
Out[193]:
   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 [194]: dfju = pd.read_json(json, date_unit='ns')

In [195]: dfju
Out[195]:
   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

**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 [196]: randfloats = np.random.uniform(-100, 1000, 10000)

In [197]: randfloats.shape = (1000, 10)

In [198]: dffloats = DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [199]: jsonfloats = dffloats.to_json()

In [200]: timeit read_json(jsonfloats)
10 loops, best of 3: 23.4 ms per loop

In [201]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 6.39 ms per loop

The speedup is less noticeable for smaller datasets:

In [202]: jsonfloats = dffloats.head(100).to_json()

In [203]: timeit read_json(jsonfloats)
100 loops, best of 3: 6 ms per loop

In [204]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 3.38 ms per loop
**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

New in version 0.13.0.

pandas provides a utility function to take a dict or list of dicts and `normalize` this semi-structured data into a flat table.

```python
In [205]: from pandas.io.json import json_normalize

In [206]: data = [{
   ...: 'state': 'Florida',
   ...: 'shortname': 'FL',
   ...: 'info': {
   ...:     'governor': 'Rick Scott'
   ...:   },
   ...:   'counties': [{
   ...:     'name': 'Dade', 'population': 12345},
   ...:     { 'name': 'Broward', 'population': 40000},
   ...:     { 'name': 'Palm Beach', 'population': 60000}]
   ...: },
   ...: {
   ...: 'state': 'Ohio',
   ...: 'shortname': 'OH',
   ...: 'info': {
   ...:     'governor': 'John Kasich'
   ...:   },
   ...:   'counties': [{
   ...:     'name': 'Summit', 'population': 1234},
   ...:     { 'name': 'Cuyahoga', 'population': 1337}]
   ...: }]

In [207]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor']])
```

```
   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
```

### 24.3 HTML

#### 24.3.1 Reading HTML Content

**Warning:** We highly encourage you to read the HTML parsing gotchas regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

New in version 0.12.0.
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 [208]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'

In [209]: dfs = read_html(url)

In [210]: dfs
```

```
Out[210]:
[          Bank Name          City ST CERT \
0       Edgebrook Bank     Chicago IL 57772
1   Doral BankEn Espanol       San Juan PR 32102
2  Capitol City Bank & Trust Company Atlanta GA 33938
3    Highland Community Bank     Chicago IL 20290
4  First National Bank of Crestview     Crestview FL 17557
5    Northern Star Bank           Mankato MN 34983
6  Frontier Bank, FSB D/B/A El Paseo Bank     Palm Desert CA 34738
   ... ... ... ... ... ... ...
532          Hamilton Bank, NAEn Espanol   Miami FL 24382
533    Sinclair National Bank          Gravette AR 34248
534            Superior Bank, FSB Hinsdale IL 32646
535    Malta National Bank             Malta OH 6629
536  First Alliance Bank & Trust Co.     Manchester NH 34264
537  National State Bank of Metropolis     Metropolis IL 3815
538  Bank of Honolulu                 Honolulu HI 21029

          Acquiring Institution       Closing Date   \
0  Republic Bank of Chicago           May 8, 2015
1  Banco Popular de Puerto Rico       February 27, 2015
2  First-Citizens Bank & Trust Company    February 13, 2015
3      United Fidelity Bank, fsb       January 23, 2015
4        First NBC Bank              January 16, 2015
5        BankVista                December 19, 2014
6  Bank of Southern California, N.A.   November 7, 2014
...  ...  ...  ...  ...  ...
532     Israel Discount Bank of New York     January 11, 2002
533  Delta Trust & Bank                September 7, 2001
534  Superior Federal, FSB             July 27, 2001
535  North Valley Bank                May 3, 2001
536  Southern New Hampshire Bank & Trust  February 2, 2001
537  Banterra Bank of Marion          December 14, 2000
538    Bank of the Orient             October 13, 2000

          Updated Date Loss Share Type Agreement Terminated Termination Date \
0      May 19, 2015      NaN      NaN      NaN      NaN
1        May 13, 2015      NaN      NaN      NaN      NaN
2    April 21, 2015      NaN      NaN      NaN      NaN
3    April 21, 2015      NaN      NaN      NaN      NaN
4    April 21, 2015      NaN      NaN      NaN      NaN
5    March 26, 2015      NaN      NaN      NaN      NaN
6    March 26, 2015      NaN      NaN      NaN      NaN
...  ...  ...  ...  ...  ...
532     June 5, 2012      NaN      NaN      NaN      NaN
```
533  February 10, 2004  none  NaN  NaN
534  August 19, 2014  none  NaN  NaN
535  November 18, 2002  none  NaN  NaN
536  February 18, 2003  none  NaN  NaN
537  March 17, 2005  none  NaN  NaN
538  March 17, 2005  none  NaN  NaN

[539 rows x 10 columns]

**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 [211]: with open(file_path, 'r') as f:
   :     : dfs = read_html(f.read())
   :
In [212]: dfs
```

```
[ Bank Name City ST CERT \
0 Banks of Wisconsin d/b/a Bank of Kenosha Kenosha WI 35386
1 Central Arizona Bank Scottsdale AZ 34527
2 Sunrise Bank Valdosta GA 58185
3 Pisgah Community Bank Asheville NC 58701
4 Douglas County Bank Douglasville GA 21649
5 Parkway Bank Lenoir NC 57158
6 Chipola Community Bank Marianna FL 58034
.. ... ... .. ...
499 Hamilton Bank, NAEn Espanol Miami FL 24382
500 Sinclair National Bank Gravette AR 34248
501 Superior Bank, FSB Hinsdale IL 32646
502 Malta National Bank Malta OH 6629
503 First Alliance Bank & Trust Co. Manchester NH 34264
504 National State Bank of Metropolis Metropolis IL 3815
505 Bank of Honolulu Honolulu HI 21029

   | 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 19, 2013 | May 16, 2013 |
.. | ... | ... | ...
500 | Delta Trust & Bank      | September 7, 2001 | February 10, 2004 |
502 | North Valley Bank       | May 3, 2001 | November 18, 2002 |
503 | Southern New Hampshire Bank & Trust | February 2, 2001 | February 18, 2003 |
504 | Banterra Bank of Marion | December 14, 2000 | March 17, 2005 |
505 | Bank of the Orient      | October 13, 2000 | March 17, 2005 |

[506 rows x 7 columns]

You can even pass in an instance of `StringIO` if you so desire

24.3. HTML 749
In [213]: with open(file_path, 'r') as f:
.....:
sio = StringIO(f.read())
.....:

In [214]: dfs = read_html(sio)

In [215]: dfs
Out[215]:

<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>...</td>
<td>...</td>
<td>...</td>
<td>...</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>
</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>
<tr>
<td>First Federal Bank of Florida</td>
<td>April 19, 2013</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Israel Discount Bank of New York</td>
<td>January 11, 2002</td>
<td>June 5, 2012</td>
</tr>
<tr>
<td>Delta Trust &amp; Bank</td>
<td>September 7, 2001</td>
<td>February 10, 2004</td>
</tr>
<tr>
<td>North Valley Bank</td>
<td>May 3, 2001</td>
<td>November 18, 2002</td>
</tr>
<tr>
<td>Southern New Hampshire Bank &amp; Trust</td>
<td>February 2, 2001</td>
<td>February 18, 2003</td>
</tr>
<tr>
<td>Banterra Bank of Marion</td>
<td>December 14, 2000</td>
<td>March 17, 2005</td>
</tr>
<tr>
<td>Bank of the Orient</td>
<td>October 13, 2000</td>
<td>March 17, 2005</td>
</tr>
</tbody>
</table>

[506 rows x 7 columns]

Note: The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn’t run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text

match = 'Metcalf Bank'
df_list = read_html(url, match=match)

Specify a header row (by default <th> elements are used to form the column index); if specified, the header row is taken from the data minus the parsed header elements (<th> elements).
dfs = read_html(url, header=0)

Specify an index column
dfs = read_html(url, index_col=0)

Specify a number of rows to skip
dfs = read_html(url, skiprows=0)

Specify a number of rows to skip using a list (xrange (Python 2 only) works as well)
dfs = read_html(url, skiprows=range(2))

Don’t infer numeric and date types
dfs = read_html(url, infer_types=False)

Specify an HTML attribute
dfs1 = read_html(url, attrs={'id': 'table'})
dfs2 = read_html(url, attrs={'class': 'sortable'})
print (np.array_equal(dfs1[0], dfs2[0]))  # Should be True

Use some combination of the above
dfs = read_html(url, match='Metcalf Bank', index_col=0)

Read in pandas to_html output (with some loss of floating point precision)
df = DataFrame(randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = 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 = read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])

or

dfs = 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 = 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 [216]: df = DataFrame(randn(2, 2))

In [217]: df
Out[217]:
     0     1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [218]: print(df.to_html())  # raw html
    <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>

HTML:
The columns argument will limit the columns shown

In [219]: 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>

HTML:
float_format takes a Python callable to control the precision of floating point values.
In [220]: `print(df.to_html(float_format='{{0:.10f}}'.format))`

```html
<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>-0.1847438576</td>  
<td>0.4969711327</td>  
</tr>
<tr>  
<td>1</td>  
<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

In [221]: `print(df.to_html(bold_rows=False))`

```html
<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.

In [222]: `print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class']))`

```html
<table border="1" class="awesome_table_class even_more_awesome_class">
<thead>
<tr style="text-align: right;">  
<th></th>  
<th>0</th>  
</tr>
</thead>
```

24.3. **HTML**
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`.

```
In [223]: df = DataFrame({'a': list('&<>'), 'b': randn(3)})
```

Escaped:
```
In [224]: 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:
```
In [225]: 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>&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>
```
24.4 Excel files

The `read_excel()` method can read Excel 2003 (.xls) and Excel 2007 (.xlsx) files using the `xlrd` Python module and use the same parsing code as the above to convert tabular data into a DataFrame. See the *cookbook* for some advanced strategies.

24.4.1 Reading Excel Files

New in version 0.16.

`read_excel` can read more than one sheet, by setting `sheetname` to either a list of sheet names, a list of sheet positions, or `None` to read all sheets.

New in version 0.13.

Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

New in version 0.12.

`ExcelFile` has been moved to the top level namespace.

There are two approaches to reading an excel file. The `read_excel` function and the `ExcelFile` class. `read_excel` is for reading one file with file-specific arguments (ie. identical data formats across sheets). `ExcelFile` is for reading one file with sheet-specific arguments (ie. various data formats across sheets). Choosing the approach is largely a question of code readability and execution speed.

Equivalent class and function approaches to read a single sheet:

```python
# using the ExcelFile class
xls = pd.ExcelFile('path_to_file.xls')
data = xls.parse('Sheet1', index_col=None, na_values=['NA'])
```
# using the read_excel function
```python
data = read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Equivalent class and function approaches to read multiple sheets:
```python
data = {}
# For when Sheet1's format differs from Sheet2
xls = pd.ExcelFile('path_to_file.xls')
data['Sheet1'] = xls.parse('Sheet1', index_col=None, na_values=['NA'])
data['Sheet2'] = xls.parse('Sheet2', index_col=1)

# For when Sheet1’s format is identical to Sheet2
data = read_excel('path_to_file.xls', ['Sheet1','Sheet2'], index_col=None, na_values=['NA'])
```

## Specifying Sheets

**Note:** The second argument is `sheetname`, not to be confused with `ExcelFile.sheet_names`

**Note:** An ExcelFile’s attribute `sheet_names` provides access to a list of sheets.

- The arguments `sheetname` allows specifying the sheet or sheets to read.
- The default value for `sheetname` 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', sheetname=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', sheetname=['Sheet1',3])
```
**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 `parse_cols` keyword to allow you to specify a subset of columns to parse.

If `parse_cols` is an integer, then it is assumed to indicate the last column to be parsed.

`read_excel('path_to_file.xls', 'Sheet1', parse_cols=2)`

If `parse_cols` is a list of integers, then it is assumed to be the file column indices to be parsed.

`read_excel('path_to_file.xls', 'Sheet1', parse_cols=[0, 2, 3])`

**Cell Converters**

It is possible to transform the contents of Excel cells via the `converters` option. For instance, to convert a column to boolean:

`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})
```

**24.4.2 Writing Excel Files**

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:

`df.to_excel('path_to_file.xls', 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. One difference from 0.12.0 is that the `index_label` will be placed in the second row instead of the first. You can get the previous behaviour by setting the `merge_cells` option in `to_excel()` to `False`:

`df.to_excel('path_to_file.xls', 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.
24.4.3 Excel writer engines

New in version 0.13.

df.read_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='openpyxl')

df.to_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='openpyxl')

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 1.6.1 up to but not including 2.0.0, and experimental support for OpenPyxl 2.0.0 and later.
- `xlsxwriter`
- `xlwt`

# 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')

24.5 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:

clipdf = pd.read_clipboard()

In [226]: clipdf
Out[226]:
   A  B  C
0  x  1  4  p
1  y  2  5  q
2  z  3  6  r
The `to_clipboard` method can be used to write the contents of a DataFrame to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a DataFrame into clipboard and reading it back.

```python
In [227]: df = pd.DataFrame(randn(5,3))

In [228]: df
Out[228]:
     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 [229]: df.to_clipboard()

In [230]: pd.read_clipboard()
Out[230]:
     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 [231]: df
Out[231]:
     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 [232]: df.to_pickle('foo.pkl')
```

The `read_pickle` function in the `pandas` namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```python
In [233]: read_pickle('foo.pkl')
Out[233]:
     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

**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)

**Warning:** Several internal refactorings, 0.13 ([Series Refactoring](#)), and 0.15 ([Index Refactoring](#)), preserve compatibility with pickles created prior to these versions. However, these must be read with `pd.read_pickle`, rather than the default python `pickle.load`. See [this question](#) for a detailed explanation.

**Note:** These methods were previously `pd.save` and `pd.load`, prior to 0.12.0, and are now deprecated.

### 24.7 msgpack (experimental)

New in version 0.13.0.

Starting in 0.13.0, pandas is supporting 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.

```
In [234]: df = DataFrame(np.random.rand(5,2),columns=list('AB'))
In [235]: df.to_msgpack('foo.msg')
In [236]: pd.read_msgpack('foo.msg')
Out[236]:
   A     B
0  0.154336  0.710999
1  0.398096  0.765220
2  0.586749  0.293052
3  0.290293  0.710783
4  0.988593  0.062106
```

```
In [237]: s = Series(np.random.rand(5),index=date_range('20130101',periods=5))
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [238]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1,2,3]), s)
In [239]: pd.read_msgpack('foo.msg')
Out[239]:
   A     B
0  0.154336  0.710999
1  0.398096  0.765220
2  0.586749  0.293052
3  0.290293  0.710783
4  0.988593  0.062106
     u'foo', array([1, 2, 3]), 2013-01-01 0.690810
     2013-01-02 0.235907
     2013-01-03 0.712756
```
You can pass `iterator=True` to iterate over the unpacked results

```
In [240]: for o in pd.read_msgpack('foo.msg', iterator=True):
    ......:    print o
    ......:
        A    B
    0  0.154336  0.710999
    1  0.398096  0.765220
    2  0.586749  0.293052
    3  0.290293  0.710783
    4  0.988593  0.062106
    foo

[1 2 3]
2013-01-01  0.690810
2013-01-02  0.235907
2013-01-03  0.712756
2013-01-04  0.119599
2013-01-05  0.023493
Freq: D, dtype: float64
```

You can pass `append=True` to the writer to append to an existing pack

```
In [241]: df.to_msgpack('foo.msg', append=True)

In [242]: pd.read_msgpack('foo.msg')
Out[242]:
    A    B
 0  0.154336  0.710999
 1  0.398096  0.765220
 2  0.586749  0.293052
 3  0.290293  0.710783
 4  0.988593  0.062106
foo
2013-01-01  0.690810
2013-01-02  0.235907
2013-01-03  0.712756
2013-01-04  0.119599
2013-01-05  0.023493
Freq: D, dtype: float64
```

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.

```
In [243]: pd.to_msgpack('foo2.msg', { 'dict' : [ { 'df' : df }, { 'string' : 'foo' }, { 'scalar' : 1. }, { 's' : s } ] })
In [244]: pd.read_msgpack('foo2.msg')
Out[244]:

{u'dict': ({u'df': A    B
        0  0.154336  0.710999
        1  0.398096  0.765220
        2  0.586749  0.293052
        3  0.290293  0.710783
        4  0.988593  0.062106
foo
2013-01-01  0.690810
2013-01-02  0.235907
2013-01-03  0.712756
2013-01-04  0.119599
2013-01-05  0.023493
Freq: D, dtype: float64, A    B
        0  0.154336  0.710999
        1  0.398096  0.765220
        2  0.586749  0.293052
        3  0.290293  0.710783
        4  0.988593  0.062106
}}
```
3  0.290293  0.710783
4  0.988593  0.062106,
{u'string': u'foo'},
{u'scalar': 1.0},
{u's': 2013-01-01 0.690810
2013-01-02 0.235907
2013-01-03 0.712756
2013-01-04 0.119599
2013-01-05 0.023493
Freq: D, dtype: float64})
```

### 24.7.1 Read/Write API

Msgpacks can also be read from and written to strings.

In [245]: df.to_msgpack()
Out[245]: `blocks` `items` `name` `dtype` `compress` `data` `A` `B` `klass` `Index` `typ` `index` `compress` `shape` `values` `P` `O`

Furthermore you can concatenate the strings to produce a list of the original objects.

In [246]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[246]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.154336</td>
<td>0.710999</td>
</tr>
<tr>
<td>1</td>
<td>0.398096</td>
<td>0.765220</td>
</tr>
<tr>
<td>2</td>
<td>0.586749</td>
<td>0.293052</td>
</tr>
<tr>
<td>3</td>
<td>0.290293</td>
<td>0.710783</td>
</tr>
<tr>
<td>4</td>
<td>0.988593</td>
<td>0.062106</td>
</tr>
</tbody>
</table>
2013-01-01 0.690810
2013-01-02 0.235907
2013-01-03 0.712756
2013-01-04 0.119599
2013-01-05 0.023493
Freq: D, dtype: float64)

### 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:** As of version 0.15.0, pandas requires PyTables >= 3.0.0. Stores written with prior versions of pandas / PyTables >= 2.3 are fully compatible (this was the previous minimum PyTables required version).

**Warning:** There is a PyTables indexing bug 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.

In [247]: store = HDFStore('store.h5')

In [248]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
Empty
Objects can be written to the file just like adding key-value pairs to a dict:

In [249]: np.random.seed(1234)

In [250]: index = date_range('1/1/2000', periods=8)

In [251]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [252]: df = DataFrame(randn(8, 3), index=index,
                columns=['A', 'B', 'C'])

In [253]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
               major_axis=date_range('1/1/2000', periods=5),
               minor_axis=['A', 'B', 'C', 'D'])

# store.put('s', s) is an equivalent method
In [254]: store['s'] = s

In [255]: store['df'] = df

In [256]: store['wp'] = wp

# the type of stored data
In [257]: store.root.wp._v_attrs.pandas_type
Out[257]: 'wide'

In [258]: store
Out[258]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame (shape->[8,3])
/s series (shape->[5])
/wp wide (shape->[2,5,4])

In a current or later Python session, you can retrieve stored objects:

# store.get('df') is an equivalent method
In [259]: store['df']
Out[259]:
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
In [260]: store.df
Out[260]:
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
Deletion of the object specified by the key

```python
# store.remove('wp') is an equivalent method
In [261]: del store['wp']
```

```python
In [262]: store
Out[262]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame (shape->[8,3])
/s series (shape->[5])
```

Closing a Store, Context Manager

```python
In [263]: store.close()

In [264]: store
Out[264]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
File is CLOSED
```

```python
In [265]: store.is_open
Out[265]: False
```

```python
# Working with, and automatically closing the store with the context
# manager
In [266]: with 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. (new in 0.11.0)

```python
In [267]: df_tl = DataFrame(dict(A=list(range(5)), B=list(range(5))))

In [268]: df_tl.to_hdf('store_tl.h5','table',append=True)

In [269]: read_hdf('store_tl.h5', 'table', where = ['index>2'])
Out[269]:
   A  B
3  3  3
4  4  4
```

## 24.8.2 Fixed Format

Note: This was prior to 0.13.0 the Storer 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 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
dataframe = randn(10,2).to_hdf('test_fixed.h5','df')
pd.read_hdf('test_fixed.h5','df',where='index>5')
```

```python
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.

New in version 0.13.

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 [270]: store = HDFStore('store.h5')
In [271]: df2 = df[4:]
In [272]: store.append('df', df1)
In [273]: store.append('df', df2)
```

```python
In [274]: store
Out[274]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
`/df` frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
```

```python
In [275]: store.select('df')
Out[275]:
   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
```
# the type of stored data
In [277]: store.root.df._v_attrs.pandas_type
Out[277]: '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 [278]: store.put('foo/bar/bah', df)

In [279]: store.append('food/orange', df)

In [280]: store.append('food/apple', df)

In [281]: store
Out[281]:<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])
/foo/apple frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/orange frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])

# a list of keys are returned
In [282]: store.keys()
Out[282]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [283]: store.remove('food')

In [284]: store
Out[284]:<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])

## 24.8.5 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 appends will truncate strings at this length.

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 [285]: df_mixed = DataFrame({ 'A' : randn(8),
.....: 'B' : randn(8),
}
......:    'C' : np.array(randn(8), dtype='float32'),
......:    'string' : 'string',
......:    'int' : 1,
......:    'bool' : True,
......:    'datetime64' : Timestamp('20010102'))},
......:    index=list(range(8)))

In [286]: df_mixed.ix[3:5, ['A', 'B', 'string', 'datetime64']] = np.nan

In [287]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})

In [288]: df_mixed1 = store.select('df_mixed')

In [289]: df_mixed1
Out[289]:
A   B     C  bool  datetime64  int  string
0  0.704721 -1.152659 -0.430096  True  2001-01-02  1 string
1 -0.785435  0.631979  0.767369  True  2001-01-02  1 string
2  0.462060  0.039513  0.984920  True  2001-01-02  1 string
3  NaN     NaN  0.270836  True   NaT   1  NaN
4    NaN     NaN  1.391986  True   NaT   1  NaN
5    NaN     NaN  0.079842  True   NaT   1  NaN
6  2.007843  0.152631 -0.399965  True  2001-01-02  1 string
7  0.226963  0.164530 -1.027851  True  2001-01-02  1 string

In [290]: df_mixed1.get_dtype_counts()
Out[290]:
bool  1
datetime64[ns]  1
float32  2
float64  2
int64  1
object  1
dtype: int64

# we have provided a minimum string column size
In [291]: store.root.df_mixed.table
Out[291]:
/df_mixed/table (Table(8,)) ''
description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
    "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
    "values_block_2": Int64Col(shape=(1,), dflt=0, pos=3),
    "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
    "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
    "values_block_5": StringCol(itemsize=50, shape=(1,), dflt='', pos=6)}
byteorder := 'little'
chunkshape := (689,)
autoindex := True
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

24.8.6 Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.
Index:

```python
In [292]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
                           ['one', 'two', 'three'],
                           [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
                           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
                       [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
                           names=['foo', 'bar'])
```

```python
In [293]: df_mi = DataFrame(np.random.randn(10, 3), index=index,
                           columns=['A', 'B', 'C'])
```

```python
In [294]: df_mi
Out[294]:
   A       B       C
foo bar
foo 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
```

```python
In [295]: store.append('df_mi',df_mi)
```

```python
In [296]: store.select('df_mi')
Out[296]:
   A       B       C
foo bar
foo 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 [297]: store.select('df_mi', 'foo=bar')
```

```
Out[297]:
   A       B       C
foo bar
foo 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
```

### 24.8.7 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

```python
string = "HolyMoly"
store.select('df', 'index == string')
```

instead of this

```python
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

```python
store.select('df', 'index == %r' % string)
```

which will quote string.

Here are some examples:

In [298]: dfq = DataFrame(randn(10,4),columns=list('ABCD'),index=date_range('20130101',periods=10))

In [299]: store.append('dfq',dfq,format='table',data_columns=True)

Use boolean expressions, with in-line function evaluation.

In [300]: store.select('dfq',"index>Timestamp('20130104') & columns=['A', 'B']")

Out[300]:
```
   A     B
2013-01-05  1.210384  0.797435
2013-01-06  0.850346  1.176812
2013-01-07  0.984188  0.121728
2013-01-08  0.796595  0.474021
2013-01-09  0.804834  2.123620
2013-01-10  0.334198  0.536784
```

Use and inline column reference

In [301]: store.select('dfq',where="A>0 or C>0")

Out[301]:
```
     A     B     C     D
2013-01-01  0.436258 -1.703013  0.393711  0.479324
2013-01-02  0.299016  0.694103  0.678630  0.239556
2013-01-03  0.151227  0.816127  1.893534  0.639633
2013-01-04 -0.962029 -2.085266  1.930247 -1.735349
2013-01-05  1.210384  0.797435 -0.379811  0.702562
2013-01-07  0.984188 -0.121728  2.365769  0.496143
2013-01-08  0.796595  0.474021 -0.056696  1.357797
2013-01-10  0.334198  0.536784 -0.743830 -0.320204
```

Works with a Panel as well.

In [302]: store.append('wp',wp)

In [303]: store
Out[303]:
```<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```
In [304]: store.select('wp', "major_axis>Timestamp('20000102') & minor_axis=['A', 'B']")
Out[304]:
<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 [305]: store.select('df', "columns=['A', 'B']")
Out[305]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</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>

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 [306]: wp.to_frame()
Out[306]:

<table>
<thead>
<tr>
<th></th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>major</td>
<td>minor</td>
<td></td>
</tr>
<tr>
<td>2000-01-01</td>
<td>A</td>
<td>1.058969</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.397840</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.337438</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>1.047579</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>A</td>
<td>1.045938</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.863717</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.122092</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-04</td>
<td>B</td>
<td>0.036142</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-2.074978</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.247792</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>A</td>
<td>-0.897157</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.136795</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.018289</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.755414</td>
</tr>
</tbody>
</table>

[20 rows x 2 columns]

# limiting the search
In [307]: store.select('wp','major_axis>20000102 & minor_axis=['A','B'])",
       ....:
       start=0, stop=10)
.....:
Out[307]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 1 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to B

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.

Using timedelta64[ns]
New in version 0.13.
Beginning in 0.13.0, 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:

Warning: This requires numpy >= 1.7

In [308]: from datetime import timedelta

In [309]: dftd = DataFrame(dict(A = Timestamp('20130101'), B = [ Timestamp('20130101') + timedelta(days=i,seconds=10) for i in range(10) ]))

In [310]: dftd['C'] = dftd['A']-dftd['B']

In [311]: dftd
Out[311]:
   A             B                C
0 2013-01-01 2013-01-01 00:00:10 -1 days +23:59:50
1 2013-01-01 2013-01-02 00:00:10 -2 days +23:59:50
2 2013-01-01 2013-01-03 00:00:10 -3 days +23:59:50
3 2013-01-01 2013-01-04 00:00:10 -4 days +23:59:50
4 2013-01-01 2013-01-05 00:00:10 -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10 -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10 -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10 -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10 -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50

In [312]: store.append('dftd',dftd,data_columns=True)

In [313]: store.select('dftd','C<'-3.5D'')
Out[313]:
   A             B                C
4 2013-01-01 2013-01-05 00:00:10 -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10 -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10 -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10 -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10 -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50
24.8.8 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 (starting 0.10.1) 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 [314]: i = store.root.df.table.cols.index

In [315]: i.optlevel, i.kind
Out[315]: (6, 'medium')

# change an index by passing new parameters
In [316]: store.create_table_index('df', optlevel=9, kind='full')

In [317]: i = store.root.df.table.cols.index

In [318]: i.optlevel, i.kind
Out[318]: (9, 'full')
```

See here for how to create a completely-sorted-index (CSI) on an existing store.

24.8.9 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`.

```python
In [319]: df_dc = df.copy()

In [320]: df_dc['string'] = 'foo'

In [321]: df_dc.ix[4:6,'string'] = np.nan

In [322]: df_dc.ix[7:9,'string'] = 'bar'

In [323]: df_dc['string2'] = 'cool'

In [324]: df_dc.ix[1:3,['B','C']] = 1.0

In [325]: df_dc
Out[325]:
   A  B    C      string  string2
0 2000-01-01  0.887163  0.859588 -0.636524    foo    cool
1 2000-01-02  0.015696  1.000000  1.000000    foo    cool
2 2000-01-03  0.991946  1.000000  1.000000    foo    cool
3 2000-01-04 -0.334077  0.002118  0.405453    foo    cool
4 2000-01-05  0.289092  1.321158 -1.546906   NaN    cool
5 2000-01-06 -0.202646 -0.655969  0.193421   NaN    cool
6 2000-01-07  0.553439  1.318152 -0.469305    foo    cool
7 2000-01-08  0.675554 -1.817027 -0.183109    bar    cool
```
# on-disk operations
In [326]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])

In [327]: store.select('df_dc', [ Term('B>0') ])
Out[327]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>string</th>
<th>string2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.887163</td>
<td>0.859588</td>
<td>-0.636524</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.015696</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.991946</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.334077</td>
<td>0.002118</td>
<td>0.405453</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.289092</td>
<td>1.321158</td>
<td>-1.546906</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.553439</td>
<td>1.318152</td>
<td>-0.469305</td>
<td>foo</td>
</tr>
</tbody>
</table>

# getting creative
In [328]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
Out[328]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>string</th>
<th>string2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-02</td>
<td>0.015696</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.991946</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.334077</td>
<td>0.002118</td>
<td>0.405453</td>
<td>foo</td>
</tr>
</tbody>
</table>

# this is in-memory version of this type of selection
In [329]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[329]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>string</th>
<th>string2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-02</td>
<td>0.015696</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.991946</td>
<td>1.000000</td>
<td>1.000000</td>
<td>foo</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.334077</td>
<td>0.002118</td>
<td>0.405453</td>
<td>foo</td>
</tr>
</tbody>
</table>

# we have automagically created this index and the B/C/string/string2 columns are stored separately as "PyTables" columns
In [330]: store.root.df_dc.table
Out[330]:

/df_dc/table (Table(8,)) ''

description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
    "B": Float64Col(shape=(), dflt=0.0, pos=2),
    "C": Float64Col(shape=(), dflt=0.0, pos=3),
    "string": StringCol(itemsize=3, shape=(), dflt='', pos=4),
    "string2": StringCol(itemsize=4, shape=(), dflt='', pos=5)}
byteorder := 'little'
chunkshape := (1680,)
autoindex := True
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False}

There is some performance degradation by making lots of columns into data columns, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!)

774 Chapter 24. IO Tools (Text, CSV, HDF5, ...)
24.8.10 Iterator

Starting in 0.11.0, 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 [331]: 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
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
    A   B    C
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109

Note: New in version 0.12.0.

You can also use the iterator with read_hdf which will open, then automatically close the store when finished iterating.

for df in read_hdf('store.h5','df', chunksize=3):
    print(df)

Note, that the chunksize keyword applies to the source rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

In [332]: dfeq = DataFrame({'number': np.arange(1,11)})

In [333]: dfeq
Out[333]:
   number
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10

In [334]: store.append('dfeq', dfeq, data_columns=['number'])

In [335]: def chunks(l, n):
    .....:         return [l[i:i+n] for i in range(0, len(l), n)]
    .....:

In [336]: evens = [2,4,6,8,10]

In [337]: coordinates = store.select_as_coordinates('dfeq','number=evens')
In [338]: for c in chunks(coordinates, 2):
.....:    print store.select('dfeq', where=c)
.....:
       number
1  2
3  4
5  6
7  8
9 10

24.8.11 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 [339]: store.select_column('df_dc', 'index')
Out[339]:
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
dtype: datetime64[ns]

In [340]: store.select_column('df_dc', 'string')
Out[340]:
0  foo
1  foo
2  foo
3  foo
4  NaN
5  NaN
6  foo
7  bar
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 [341]: df_coord = DataFrame(np.random.randn(1000,2),index=date_range('20000101',periods=1000))

In [342]: store.append('df_coord',df_coord)

In [343]: c = store.select_as_coordinates('df_coord','index>20020101')

In [344]: c.summary()
Out[344]: u'Int64Index: 268 entries, 732 to 999'
In [345]: store.select('df_coord', where=c)
Out[345]:
         0       1
2002-01-02 -0.667994 -0.368175
2002-01-03  0.020119  -0.823208
2002-01-04 -0.165481   0.720866
2002-01-05  1.295919  -0.527767
2002-01-06 -0.463393  -0.150792
2002-01-07 -1.139341  -0.954387
2002-01-08  0.051837  -0.147048
...       ...       ...
2002-09-20  0.058626  -0.489107
2002-09-21 -0.356873  -0.437071
2002-09-22 -0.243534  -0.093778
2002-09-23 -0.615983   0.414649
2002-09-24  0.202096  -0.297561
2002-09-25  0.681661   0.538311
2002-09-26 -0.614051   0.769058
[268 rows x 2 columns]

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this mask would be a resulting index from an indexing operation. This example selects the months of a datetimeindex which are 5.

In [346]: df_mask = DataFrame(np.random.randn(1000,2),index=date_range('20000101',periods=1000))
In [347]: store.append('df_mask',df_mask)
In [348]: c = store.select_column('df_mask','index')
In [349]: where = c[_datetime(c).month==5].index
In [350]: store.select('df_mask', where=where)
Out[350]:
         0       1
2000-05-01 -0.098554 -0.280782
2000-05-02  0.739851  1.627182
2000-05-03  0.301323 -0.145601
2000-05-04  0.227530  1.048856
2000-05-05  1.773939  1.116887
2000-05-06  1.081251  1.509416
2000-05-07 -0.498694  -0.913155
...       ...       ...
2002-05-25 -0.497252  0.348099
2002-05-26 -1.287350  -1.488122
2002-05-27 -0.726220   0.507747
2002-05-28  0.189871   0.980528
2002-05-29  0.555156   0.369371
2002-05-30 -0.637441  -3.434819
2002-05-31 -0.070283  -0.278044
[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.
24.8.12 Multiple Table Queries

New in 0.10.1 are the methods `append_to_multiple` and `select_as_multiple`, that 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.

```python
In [352]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
                  columns=['A', 'B', 'C', 'D', 'E', 'F'])

In [353]: df_mt['foo'] = 'bar'

In [354]: df_mt.ix[1, ('A', 'B')] = np.nan

# you can also create the tables individually
In [355]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None },
                           df_mt, selector='df1_mt')

In [356]: store.select('df1_mt')
```

# individual tables were created

```python
In [357]: store.select('df1_mt')
```
Out[357]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.816310</td>
<td>1.282296</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.684353</td>
<td>-1.755306</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.315814</td>
<td>1.455079</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.027564</td>
<td>0.046757</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.416244</td>
<td>-0.821168</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.665090</td>
<td>1.084344</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.607460</td>
<td>0.790907</td>
</tr>
</tbody>
</table>

In [358]: store.select('df2_mt')

Out[358]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-1.521825</td>
<td>-0.428670</td>
<td>-1.550209</td>
<td>0.826839</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>1.236974</td>
<td>-1.328279</td>
<td>0.662291</td>
<td>1.894976</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.746478</td>
<td>0.851039</td>
<td>1.415686</td>
<td>-0.929096</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.452287</td>
<td>1.575492</td>
<td>-0.197377</td>
<td>-0.219901</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>1.190342</td>
<td>2.115021</td>
<td>0.148762</td>
<td>1.073931</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.709897</td>
<td>-2.022441</td>
<td>0.714697</td>
<td>0.318215</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.852225</td>
<td>0.096696</td>
<td>-0.379903</td>
<td>0.929313</td>
<td>bar</td>
</tr>
</tbody>
</table>

# as a multiple
In [359]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'], selector='df1_mt')

Out[359]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-07</td>
<td>0.665090</td>
<td>1.084344</td>
<td>-0.709897</td>
<td>-2.022441</td>
<td>0.714697</td>
<td>0.318215</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.607460</td>
<td>0.790907</td>
<td>0.852225</td>
<td>0.096696</td>
<td>-0.379903</td>
<td>0.929313</td>
<td>bar</td>
</tr>
</tbody>
</table>

24.8.13 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 [360]: store.remove('wp', 'major_axis>20000102' )
Out[360]: 12

In [361]: store.select('wp')
Out[361]:
<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
```

Please note that HDF5 DOES NOT RECLAIM SPACE in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again WILL TEND TO INCREASE THE FILE SIZE. To clean the file, use ptrepack (see below).

### 24.8.14 Compression

PyTables allows the stored data to be compressed. This applies to all kinds of stores, not just tables.

- Pass `complevel=int` for a compression level (1-9, with 0 being no compression, and the default)
- Pass `complib=lib` where `lib` is any of zlib, bzip2, lzo, blosc for whichever compression library you prefer.

HDFStore will use the file based compression scheme if no overriding `complib` or `complevel` options are provided. blosc offers very fast compression, and is my most used. Note that lzo and bzip2 may not be installed (by Python) by default.

Compression for all objects within the file

- `store_compressed = HDFStore('store_compressed.h5', complevel=9, complib='blosc')`

Or on-the-fly compression (this only applies to tables). You can turn off file compression for a specific table by passing `complevel=0`

- `store.append('df', df, complib='zlib', complevel=5)`

ptrepack

PyTables offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied PyTables utility ptrepack. In addition, ptrepack can change compression levels after the fact.

- `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.

### 24.8.15 Notes & Caveats

- Once a table is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
• If a row has np.nan for EVERY COLUMN (having a nan in a string, or a NaT in a datetime-like column counts as having a value), then those rows WILL BE DROPPED IMPLICITLY. This limitation may be addressed in the future.

• 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 issue (:2397) for more information.

• If you use locks to manage write access between multiple processes, you may want to use fsync() before releasing write locks. For convenience you can use store.flush(fsync=True) to do this for you.

• PyTables only supports fixed-width string columns in tables. The sizes of a string based indexing column (e.g. columns or minor_axis) are determined as the maximum size of the elements in that axis or by passing the parameter

• 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. Generally identifiers that have spaces, start with numbers, or _ or have – embedded are not considered natural. These types of identifiers cannot be used in a where clause and are generally a bad idea.

### 24.8.16 DataTypes

HDFStore will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

- **floating**: float64, float32, float16 (using np.nan to represent invalid values)
- **integer**: int64, int32, int8, uint64, uint32, uint8
- **bool**
- **datetime64[ns]** (using NaT to represent invalid values)
- **object**: strings (using np.nan to represent invalid values)

Currently, unicode and datetime columns (represented with a dtype of object), WILL FAIL. In addition, even though a column may look like a datetime64[ns], if it contains np.nan, this WILL FAIL. You can try to convert datetimelike columns to proper datetime64[ns] columns, that possibly contain NaT to represent invalid values. (Some of these issues have been addressed and these conversion may not be necessary in future versions of pandas)

```python
In [362]: import datetime

In [363]: df = DataFrame(dict(datelike=Series([datetime.datetime(2001, 1, 1),
                                           ....:
                                           datetime.datetime(2001, 1, 2), np.nan])))

In [364]: df
Out[364]:
        datelike
0  2001-01-01
1  2001-01-02
2    NaT
```
In [365]: df.dtypes
Out[365]:
   datelike    datetime64[ns]
dtype: object

# to convert
In [366]: df['datelike'] = Series(df['datelike'].values, dtype='M8[ns]')

In [367]: df
Out[367]:
   datelike
0  2001-01-01
1  2001-01-02
2    NaT

In [368]: df.dtypes
Out[368]:
   datelike    datetime64[ns]
dtype: object

24.8.17 Categorical Data

New in version 0.15.2.

Writing data to a HDFStore that contains a category dtype was implemented in 0.15.2. Queries work the same as if it was an object array. However, the category dtyped data is stored in a more efficient manner.

In [369]: dfcat = DataFrame({'A': Series(list('aabbcdba')).astype('category'),
                      'B': np.random.randn(8)})

In [370]: dfcat
Out[370]:
   A    B
0  a  0.811031
1  a -0.356817
2  b  1.047085
3  b  0.664705
4  c -0.086919
5  d  0.416905
6  b -0.764381
7  a -0.287229

In [371]: dfcat.dtypes
Out[371]:
   A  category
   B  float64
dtype: object

In [372]: cstore = pd.HDFStore('cats.h5', mode='w')

In [373]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [374]: result = cstore.select('dfcat', where="A in ['b','c']")

In [375]: result
Out[375]:
   A    B
0  b -0.764381
1  c -0.086919
2  b -0.764381
3  c -0.086919
A B
2 b 1.047085
3 b 0.664705
4 c -0.086919
6 b -0.764381

```python
In [376]: result.dtypes
Out[376]:
A category
B float64
dtype: object
```

**Warning:** The format of the Categorical is readable by prior versions of pandas (< 0.15.2), but will retrieve the data as an integer based column (e.g. the codes). However, the categories can be retrieved but require the user to select them manually using the explicit meta path.

The data is stored like so:

```python
In [377]: cstore
Out[377]:
<class 'pandas.io.pytables.HDFStore'>
File path: cats.h5
/dfcat frame_table (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A])
/dfcat/meta/A/meta series_table (typ->appendable,nrows->4,ncols->1,indexers->[index],dc->[values])
```

```python
# to get the categories
In [378]: cstore.select('dfcat/meta/A/meta')
Out[378]:
0 a
1 b
2 c
3 d
dtype: object
```

### 24.8.18 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.

Starting in 0.11.0, passing a min_itemsize dict will cause all passed columns to be created as data_columns automatically.

**Note:** If you are not passing any data_columns, then the min_itemsize will be the maximum of the length of any string passed

```python
In [379]: dfs = DataFrame(dict(A = 'foo', B = 'bar'),index=list(range(5)))
```
In [380]: dfs
Out[380]:
   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 [381]: store.append('dfs', dfs, min_itemsize = 30)

In [382]: store.get_storer('dfs').table
Out[382]:
/dfs/table (Table(5,)) ''
  description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": StringCol(itemsize=30, shape=(2,), dflt='', 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 [383]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })

In [384]: store.get_storer('dfs2').table
Out[384]:
/dfs2/table (Table(5,)) ''
  description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": StringCol(itemsize=3, shape=(1,), dflt='', pos=1),
    "A": StringCol(itemsize=30, shape=(), dflt='', pos=2)
  }
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False
  }

nan_rep

String columns will serialize a np.nan (a missing value) with the nan_rep string representation. This defaults to the string value nan. You could inadvertently turn an actual nan value into a missing value.

In [385]: dfss = DataFrame(dict(A = ['foo','bar','nan']))

In [386]: dfss
Out[386]:
   A
0  foo
1  bar
2  nan

In [387]: store.append('dfss', dfss)

In [388]: store.select('dfss')
# here you need to specify a different nan rep
In [389]: store.append('dfss2', dfss, nan_rep='_nan_')

In [390]: store.select('dfss2')
Out[390]:
A
0 foo
1 bar
2 nan

## 24.8.19 External Compatibility

HDFStore writes table format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, HDFStore can read native PyTables format tables.

It is possible to write an HDFStore object that can easily be imported into R using the rhdf5 library (Package website). Create a table format store like this:

In [391]: np.random.seed(1)

In [392]: 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 [393]: df_for_r.head()
Out[393]:
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 [394]: store_export = HDFStore('export.h5')

In [395]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [396]: store_export
Out[396]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
/df_for_r frame_table (typ->appendable,nrows->100,ncols->3,indexers->[index])

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.
```

24.8. HDF5 (PyTables)
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:

> data = loadhdf5data("transfer.hdf5")
> head(data)

    first  second  class
1 0.41702204 0.3266449   0
2 0.72032449 0.5270581   0
3 0.00011437 0.8859421   1
4 0.30233257 0.3572698   1
5 0.14675589 0.9085352   1
6 0.09233859 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.

24.8.20 Backwards Compatibility

0.10.1 of HDFStore can read tables created in a prior version of pandas, however query terms using the prior (undocumented) methodology are unsupported. HDFStore will issue a warning if you try to use a legacy-format file. You must read in the entire file and write it out using the new format, using the method copy to take advantage of the updates. The group attribute pandas_version contains the version information. copy takes a number of options, please see the docstring.

# a legacy store
In [397]: legacy_store = HDFStore(legacy_file_path, 'r')

In [398]: legacy_store
Out[398]:
<class 'pandas.io.pytables.HDFStore'>
# copy (and return the new handle)

In [399]: new_store = legacy_store.copy('store_new.h5')

In [400]: new_store
Out[400]:
<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/df1Mixed frame_table [0.10.0] (typ->appendable,nrows->30,ncols->11,indexers->[index])
/df1Mixed wide (shape->[3,30,4])
/pl Mixed wide_table [0.10.0] (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,minor_axis])
/pl Mixed wide_table [0.10.0] (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis])

In [401]: new_store.close()

## 24.8.21 Performance

- tables format come with a writing performance penalty as compared to fixed stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.

- You can pass chunksize=<int> to append, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.

- You can pass expectedrows=<int> to the first append, to set the TOTAL number of 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.

## 24.8.22 Experimental

HDFStore supports Panel4D storage.

In [402]: p4d = Panel4D({ 'l1' : wp })

In [403]: p4d
Out[403]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 1 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: l1 to l1
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
In [404]: store.append('p4d', p4d)

In [405]: store
Out[405]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df   frame_table  {typ->appendable,nrows->8,ncols->3,indexers->{index}}
/df1_mt frame_table  {typ->appendable,nrows->7,ncols->2,indexers->{index},dc->[A,B]}
/df2_mt frame_table  {typ->appendable,nrows->7,ncols->5,indexers->{index}}
/df_coord frame_table  {typ->appendable,nrows->1000,ncols->2,indexers->{index}}
/df_dc frame_table  {typ->appendable,nrows->8,ncols->5,indexers->{index},dc->[B,C,string,string2]}
/df_mask frame_table  {typ->appendable,nrows->1000,ncols->2,indexers->{index}}
/df_mi frame_table  {typ->appendable_multi,nrows->10,ncols->5,indexers->{index},dc->[A,B]}  
/df_mixed frame_table  {typ->appendable,nrows->8,ncols->7,indexers->{index}}
/dfeq frame_table  {typ->appendable,nrows->10,ncols->1,indexers->{index},dc->[number]}  
/dfq frame_table  {typ->appendable,nrows->10,ncols->4,indexers->{index},dc->[A,B,C,D]}
/dfs frame_table  {typ->appendable,nrows->5,ncols->1,indexers->{index}}
/dfs2 frame_table  {typ->appendable,nrows->5,ncols->2,indexers->{index}}
/dfs2 frame_table  {typ->appendable,nrows->5,ncols->2,indexers->{index}}
/dfss frame_table  {typ->appendable,nrows->3,ncols->1,indexers->{index}}
/dfss2 frame_table  {typ->appendable,nrows->3,ncols->1,indexers->{index}}
/dftd frame_table  {typ->appendable,nrows->10,ncols->3,indexers->{index},dc->[A,B,C]}
/foo/bar/bah frame  {shape->[8,3]}
/p4d wide_table  {typ->appendable,nrows->40,ncols->1,indexers->{items,major_axis,minor_axis}}
/wp wide_table  {typ->appendable,nrows->8,ncols->2,indexers->{major_axis,major_axis,minor_axis}}

These, by default, index the three axes items, major_axis, minor_axis. On an AppendableTable it is possible to setup with the first append a different indexing scheme, depending on how you want to store your data. Pass the axes keyword with a list of dimensions (currently must by exactly 1 less than the total dimensions of the object). This cannot be changed after table creation.

In [406]: store.append('p4d2', p4d, axes=['labels', 'major_axis', 'minor_axis'])

In [407]: store
Out[407]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df   frame_table  {typ->appendable,nrows->8,ncols->3,indexers->{index}}
/df1_mt frame_table  {typ->appendable,nrows->7,ncols->2,indexers->{index},dc->[A,B]}
/df2_mt frame_table  {typ->appendable,nrows->7,ncols->5,indexers->{index}}
/df_coord frame_table  {typ->appendable,nrows->1000,ncols->2,indexers->{index}}
/df_dc frame_table  {typ->appendable,nrows->8,ncols->5,indexers->{index},dc->[B,C,string,string2]}
/df_mask frame_table  {typ->appendable,nrows->1000,ncols->2,indexers->{index}}
/df_mi frame_table  {typ->appendable_multi,nrows->10,ncols->5,indexers->{index},dc->[A,B]}  
/df_mixed frame_table  {typ->appendable,nrows->8,ncols->7,indexers->{index}}
/dfeq frame_table  {typ->appendable,nrows->10,ncols->1,indexers->{index},dc->[number]}  
/dfq frame_table  {typ->appendable,nrows->10,ncols->4,indexers->{index},dc->[A,B,C,D]}
/dfs frame_table  {typ->appendable,nrows->5,ncols->1,indexers->{index}}
/dfs2 frame_table  {typ->appendable,nrows->5,ncols->2,indexers->{index}}
/dfs2 frame_table  {typ->appendable,nrows->5,ncols->2,indexers->{index}}
/dfss frame_table  {typ->appendable,nrows->3,ncols->1,indexers->{index}}
/dfss2 frame_table  {typ->appendable,nrows->3,ncols->1,indexers->{index}}
/dftd frame_table  {typ->appendable,nrows->10,ncols->3,indexers->{index},dc->[A,B,C]}
/foo/bar/bah frame  {shape->[8,3]}
/p4d wide_table  {typ->appendable,nrows->40,ncols->1,indexers->{items,major_axis,minor_axis}}
/wp wide_table  {typ->appendable,nrows->8,ncols->2,indexers->{major_axis,major_axis,minor_axis}}

In [408]: store.select('p4d2', [ Term('labels=1'), Term('items=Item1'), Term('minor_axis=A_big_strings')])
Out[408]:
24.9 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.

New in version 0.14.0.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some cookbook examples for some advanced strategies.

The key functions are:

- `read_sql_table(table_name, con[, schema, ...])` Read SQL database table into a DataFrame.
- `read_sql_query(sql, con[, index_col, ...])` Read SQL query into a DataFrame.
- `read_sql(sql, con[, index_col, ...])` Read SQL query or database table into a DataFrame.
- `DataFrame.to_sql(name, con[, flavor, ...])` Write records stored in a DataFrame to a SQL database.

24.9.1 pandas.read_sql_table

pandas.read_sql_table(table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None)

Read SQL database table into a DataFrame.

Given a table name and an SQLAlchemy engine, returns a DataFrame. This function does not support DBAPI connections.

Parameters

- **table_name**: string
  Name of SQL table in database

- **con**: SQLAlchemy engine
  Sqlite DBAPI connection mode not supported

- **schema**: string, default None
  Name of SQL schema in database to query (if database flavor supports this). If None, use default schema (default).

- **index_col**: string, optional
  Column to set as index

- **coerce_float**: boolean, default True
  Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.
parse_dates : list or dict

- 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
- List of column names to select from sql 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_query Read SQL query into a DataFrame.

Notes

- Any datetime values with time zone information will be converted to UTC

24.9.2 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 to be executed

con : SQLAlchemy engine 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, optional
- Column name to use as index for the returned DataFrame object.

coerce_float : boolean, default True
- Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

params : list, tuple or dict, optional
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

- 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

### 24.9.3 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

SQL query to be executed or database table name.

con : SQLAlchemy engine 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, optional

column name to use as index for the returned DataFrame object.

coerce_float : boolean, default True

Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

params : list, tuple or dict, optional
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
- 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
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.

### 24.9.4 pandas.DataFrame.to_sql

DataFrame.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'
  The flavor of SQL to use. Ignored when using SQLAlchemy engine. 'mysql' is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.
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

In [409]: from sqlalchemy import create_engine

# Create your connection.
In [410]: engine = create_engine('sqlite:///memory:')</n

24.9.5 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 [411]: 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 [412]: data.to_sql('data_chunked', engine, chunksize=1000)
```

### 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 [413]: from sqlalchemy.types import String
In [414]: 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.

### 24.9.6 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 [415]: pd.read_sql_table('data', engine)
Out[415]:
    index  id  Date  Col_1  Col_2  Col_3
0       0   26 2010-10-18     X   27.50   True
1       1   42 2010-10-19     Y  -12.50  False
2       2   63 2010-10-20     Z    5.73   True
```

You can also specify the name of the column as the DataFrame index, and specify a subset of columns to be read.

```
In [416]: pd.read_sql_table('data', engine, index_col='id')
Out[416]:
    index  Date  Col_1  Col_2  Col_3
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 [417]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[417]:
       Col_1  Col_2
0        X  27.50
```
And you can explicitly force columns to be parsed as dates:

```
In [418]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[418]:
   index  id    Date  Col_1  Col_2  Col_3
0      0   26  2010-10-18    X   27.50   True
1      1   42  2010-10-19    Y  -12.50  False
2      2   63  2010-10-20    Z    5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to `pandas.to_datetime()`:

```
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()`

### 24.9.7 Schema support

**New in version 0.15.0.**

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')
```

### 24.9.8 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 [419]: pd.read_sql_query('SELECT * FROM data', engine)
Out[419]:
   index  id    Date  Col_1  Col_2  Col_3
0      0   26  2010-10-18    X   27.50   True
1      1   42  2010-10-19    Y  -12.50  False
2      2   63  2010-10-20    Z    5.73   True
```

Of course, you can specify a more “complex” query.

```
In [420]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engine)
Out[420]:
   id  Col_1  Col_2
0  42    Y  -12.5
```

The `read_sql_query()` function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [421]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))
In [422]: df.to_sql('data_chunks', engine, index=False)
```

24.9. SQL Queries 795
In [423]: 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.

    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)])

24.9.9 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.

    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

## 24.9.10 Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```python
import sqlite3
con = sqlite3.connect(':memory:)
```

And then issue the following queries:

```python
data.to_sql('data', cnx)
pd.read_sql_query("SELECT * FROM data", con)
```

## 24.10 Google BigQuery (Experimental)

New in version 0.13.0.

The pandas.io.gbq module provides a wrapper for Google’s BigQuery analytics web service to simplify retrieving results from BigQuery tables using SQL-like queries. Result sets are parsed into a pandas DataFrame with a shape and data types derived from the source table. Additionally, DataFrames can be appended to existing BigQuery tables if the destination table is the same shape as the DataFrame.

For specifics on the service itself, see here

As an example, suppose you want to load all data from an existing BigQuery table : `test_dataset.test_table` into a DataFrame using the `read_gbq()` function.

```python
# Insert your BigQuery Project ID Here
# Can be found in the Google web console
projectid = "xxxxxxxx"

data_frame = pd.read_gbq('SELECT * FROM test_dataset.test_table', project_id = projectid)
```

You will then be authenticated to the specified BigQuery account via Google’s Oauth2 mechanism. In general, this is as simple as following the prompts in a browser window which will be opened for you. Should the browser not be available, or fail to launch, a code will be provided to complete the process manually. Additional information on the authentication mechanism can be found here

You can define which column from BigQuery to use as an index in the destination DataFrame as well as a preferred column order as follows:

```python
data_frame = pd.read_gbq('SELECT * FROM test_dataset.test_table',
    index_col='index_column_name',
    col_order=['col1', 'col2', 'col3'], project_id = projectid)
```

Finally, you can append data to a BigQuery table from a pandas DataFrame using the `to_gbq()` function. This function uses the Google streaming API which requires that your destination table exists in BigQuery. Given the BigQuery table already exists, your DataFrame should match the destination table in column order, structure, and data types. DataFrame indexes are not supported. By default, rows are streamed to BigQuery in chunks of 10,000 rows, but you can pass other chuck values via the `chunksize` argument. You can also see the progress of your post via the `verbose` flag which defaults to True. The http response code of Google BigQuery can be successful (200) even if the append failed. For this reason, if there is a failure to append to the table, the complete error response
from BigQuery is returned which can be quite long given it provides a status for each row. You may want to start with smaller chunks to test that the size and types of your dataframe match your destination table to make debugging simpler.

df = pandas.DataFrame({'string_col_name' : ['hello'],
                      'integer_col_name' : [1],
                      'boolean_col_name' : [True]})
df.to_gbq('my_dataset.my_table', project_id = projectid)

The BigQuery SQL query language has some oddities, see here

While BigQuery uses SQL-like syntax, it has some important differences from traditional databases both in functionality, API limitations (size and quantity of queries or uploads), and how Google charges for use of the service. You should refer to Google documentation often as the service seems to be changing and evolving. BigQuery is best for analyzing large sets of data quickly, but it is not a direct replacement for a transactional database.

You can access the management console to determine project id’s by:
<https://code.google.com/apis/console/b/0/?noredirect>

As of 0.15.2, the gbq module has a function `generate_bq_schema` which will produce the dictionary representation of the schema.

df = pandas.DataFrame({'A': [1.0]})
gbq.generate_bq_schema(df, default_type='STRING')

---

Warning: To use this module, you will need a valid BigQuery account. See <https://cloud.google.com/products/big-query> for details on the service.

24.11 Stata Format

New in version 0.12.0.

24.11.1 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).

In [424]: df = DataFrame(randn(10, 2), columns=list('AB'))

In [425]: 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`.

### 24.11.2 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 [426]: pd.read_stata('stata.dta')
Out[426]:
   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
```

New in version 0.16.0.

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 [427]: reader = pd.read_stata('stata.dta', chunksize=3)

In [428]: 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 [429]: reader = pd.read_stata('stata.dta', iterator=True)

In [430]: chunk1 = reader.read(5)

In [431]: chunk2 = reader.read(5)
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using

### 24.11. Stata Format
StataMissingValue objects, and columns containing missing values will have object data type.

`read_stata()` and `StataReader` supports .dta formats 104, 105, 108, 113-115 (Stata 10-12) and 117 (Stata 13+).

**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.

### 24.11.3 Categorical Data

New in version 0.15.2.

Categorical data can be exported to Stata data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. Stata does not have an explicit equivalent to a Categorical and information about whether the variable is ordered is lost when exporting.

**Warning:** Stata only supports string value labels, and so `str` is called on the categories when exporting data. Exporting Categorical variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from Stata data files as Categorical variables using the keyword argument `convert_categoricals` (True by default). The keyword argument `order_categoricals` (True by default) determines whether imported Categorical variables are ordered.

**Note:** When importing categorical data, the values of the variables in the Stata data file are not preserved since Categorical variables always use integer data types between -1 and n-1 where n is the number of categories. If the original values in the Stata data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original Stata data values and the category codes of imported Categorical variables: missing values are assigned code -1, and the smallest original value is assigned 0, the second smallest is assigned 1 and so on until the largest original value is assigned the code n-1.

**Note:** Stata supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a Categorical with string categories for the values that are labeled and numeric categories for values with no label.

### 24.12 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.

#### 24.12.1 netCDF

xray 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.
24.13 Performance Considerations

This is an informal comparison of various IO methods, using pandas 0.13.1.

In [3]: df = DataFrame(randn(1000000,2),columns=list('AB'))
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A 1000000 non-null values
B 1000000 non-null values
dtypes: float64(2)

Writing

In [14]: %timeit test_sql_write(df)
1 loops, best of 3: 6.24 s per loop

In [15]: %timeit test_hdf_fixed_write(df)
1 loops, best of 3: 237 ms per loop

In [26]: %timeit test_hdf_fixed_write_compress(df)
1 loops, best of 3: 245 ms per loop

In [16]: %timeit test_hdf_table_write(df)
1 loops, best of 3: 901 ms per loop

In [27]: %timeit test_hdf_table_write_compress(df)
1 loops, best of 3: 952 ms per loop

In [17]: %timeit test_csv_write(df)
1 loops, best of 3: 3.44 s per loop

Reading

In [18]: %timeit test_sql_read()
1 loops, best of 3: 766 ms per loop

In [19]: %timeit test_hdf_fixed_read()
10 loops, best of 3: 19.1 ms per loop

In [28]: %timeit test_hdf_fixed_read_compress()
10 loops, best of 3: 36.3 ms per loop

In [20]: %timeit test_hdf_table_read()
10 loops, best of 3: 39 ms per loop

In [29]: %timeit test_hdf_table_read_compress()
10 loops, best of 3: 60.6 ms per loop

In [22]: %timeit test_csv_read()
1 loops, best of 3: 620 ms per loop

Space on disk (in bytes)

25843712 Apr 8 14:11 test.sql
24007368 Apr 8 14:11 test_fixed.hdf
15580682 Apr 8 14:11 test_fixed_compress.hdf
24458444 Apr 8 14:11 test_table.hdf
And here’s the code

```python
import sqlite3
import os
from pandas.io import sql

df = DataFrame(randn(1000000,2),columns=list('AB'))

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)
```
Warning: In pandas 0.17.0, the sub-package `pandas.io.data` will be removed in favor of a separately installable `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 the same as in pandas v0.16.1. (GH8961)
You should replace the imports of the following:

```python
from pandas.io import data, wb
```
With:

```python
from pandas_datareader import data, wb
```

Functions from `pandas.io.data` and `pandas.io.ga` extract data from various Internet sources into a DataFrame. Currently the following sources are supported:

- **Yahoo! Finance**
- **Google Finance**
- **St.Louis FED (FRED)**
- **Kenneth French’s data library**
- **World Bank**
- **Google Analytics**

It should be noted, that various sources support different kinds of data, so not all sources implement the same methods and the data elements returned might also differ.

### 25.1 Yahoo! Finance

```python
In [1]: import pandas.io.data as web
In [2]: import datetime
In [3]: start = datetime.datetime(2010, 1, 1)
In [4]: end = datetime.datetime(2013, 1, 27)
In [5]: f = web.DataReader("F", 'yahoo', start, end)
In [6]: f.ix['2010-01-04']
```
```
Out[6]:
Open 10.170000
```
25.2 Yahoo! Finance Options

*Experimental*

The `Options` class allows the download of options data from Yahoo! Finance.

The `get_all_data` method downloads and caches option data for all expiry months and provides a formatted DataFrame with a hierarchical index, so it is easy to get to the specific option you want.

In [7]: from pandas.io.data import Options

In [8]: aapl = Options('aapl', 'yahoo')

In [9]: data = aapl.get_all_data()

In [10]: data.iloc[0:5, 0:5]

Out[10]:

<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
<th>Bid</th>
<th>Ask</th>
<th>Chg</th>
<th>PctChg</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.29</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00034290</td>
<td>97.55</td>
<td>92.10</td>
<td>94.30</td>
<td>0.00</td>
<td>0.00%</td>
</tr>
<tr>
<td>35.71</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00035710</td>
<td>88.30</td>
<td>90.80</td>
<td>91.95</td>
<td>0.00</td>
<td>0.00%</td>
</tr>
<tr>
<td>37.14</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00037140</td>
<td>90.70</td>
<td>88.50</td>
<td>90.55</td>
<td>0.60</td>
<td>0.67%</td>
</tr>
</tbody>
</table>

# Show the $100 strike puts at all expiry dates:
In [11]: data.loc[(100, slice(None), 'put'),:].iloc[0:5, 0:5]

Out[11]:

<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
<th>Bid</th>
<th>Ask</th>
<th>Chg</th>
<th>PctChg</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2015-06-19</td>
<td>put</td>
<td>AAPL150619P000100000</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00%</td>
</tr>
<tr>
<td>100</td>
<td>2015-06-26</td>
<td>put</td>
<td>AAPL150626P000100000</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>-0.02</td>
<td>-50.00%</td>
</tr>
<tr>
<td>100</td>
<td>2015-07-17</td>
<td>put</td>
<td>AAPL150717P000100000</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>-0.01</td>
<td>-25.00%</td>
</tr>
<tr>
<td>100</td>
<td>2015-08-21</td>
<td>put</td>
<td>AAPL150821P000100000</td>
<td>0.22</td>
<td>0.22</td>
<td>0.24</td>
<td>0.03</td>
<td>15.79%</td>
</tr>
<tr>
<td>100</td>
<td>2015-10-16</td>
<td>put</td>
<td>AAPL151016P000100000</td>
<td>0.60</td>
<td>0.60</td>
<td>0.62</td>
<td>0.07</td>
<td>13.21%</td>
</tr>
</tbody>
</table>

# Show the volume traded of $100 strike puts at all expiry dates:
In [12]: data.loc[(100, slice(None), 'put'),['Vol']].head()

Out[12]:

<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Vol</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2015-06-19</td>
<td>put</td>
<td>AAPL150619P000100000</td>
<td>263</td>
</tr>
<tr>
<td>100</td>
<td>2015-06-26</td>
<td>put</td>
<td>AAPL150626P000100000</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>2015-07-17</td>
<td>put</td>
<td>AAPL150717P000100000</td>
<td>52</td>
</tr>
<tr>
<td>100</td>
<td>2015-08-21</td>
<td>put</td>
<td>AAPL150821P000100000</td>
<td>23</td>
</tr>
<tr>
<td>100</td>
<td>2015-10-16</td>
<td>put</td>
<td>AAPL151016P000100000</td>
<td>126</td>
</tr>
</tbody>
</table>

If you don’t want to download all the data, more specific requests can be made.
In [13]: import datetime
In [14]: expiry = datetime.date(2016, 1, 1)
In [15]: data = aapl.get_call_data(expiry=expiry)
In [16]: data.iloc[0:5:, 0:5]
Out[16]:
<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
<th>Bid</th>
<th>Ask</th>
<th>Chg</th>
<th>PctChg</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.29</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00034290</td>
<td>97.55</td>
<td>92.10</td>
<td>94.30</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>35.71</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00035710</td>
<td>88.30</td>
<td>90.80</td>
<td>91.95</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>37.14</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00037140</td>
<td>90.70</td>
<td>88.50</td>
<td>90.55</td>
<td>0.6</td>
<td>0.67%</td>
</tr>
<tr>
<td>38.57</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00038570</td>
<td>86.45</td>
<td>87.40</td>
<td>89.95</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>40.00</td>
<td>2016-01-15</td>
<td>call</td>
<td>AAPL160115C00040000</td>
<td>89.40</td>
<td>86.00</td>
<td>87.75</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Note that if you call `get_all_data` first, this second call will happen much faster, as the data is cached.

If a given expiry date is not available, data for the next available expiry will be returned (January 15, 2015 in the above example).

Available expiry dates can be accessed from the `expiry_dates` property.

In [17]: aapl.expiry_dates
Out[17]:
[datetime.date(2015, 6, 19),
 datetime.date(2015, 6, 26),
 datetime.date(2015, 7, 2),
 datetime.date(2015, 7, 10),
 datetime.date(2015, 7, 17),
 datetime.date(2015, 7, 24),
 datetime.date(2015, 7, 31),
 datetime.date(2015, 8, 21),
 datetime.date(2015, 10, 16),
 datetime.date(2015, 12, 18),
 datetime.date(2016, 1, 15),
 datetime.date(2016, 6, 17),
 datetime.date(2017, 1, 20)]

In [18]: data = aapl.get_call_data(expiry=aapl.expiry_dates[0])
In [19]: data.iloc[0:5:, 0:5]
Out[19]:
<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
<th>Bid</th>
<th>Ask</th>
<th>Chg</th>
<th>PctChg</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00065000</td>
<td>64.25</td>
<td>61.95</td>
<td>62.35</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>70</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00070000</td>
<td>60.72</td>
<td>56.85</td>
<td>57.35</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>75</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00075000</td>
<td>57.21</td>
<td>51.85</td>
<td>52.45</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>80</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00080000</td>
<td>47.50</td>
<td>46.80</td>
<td>47.40</td>
<td>-1.36</td>
<td>-2.78%</td>
</tr>
<tr>
<td>85</td>
<td>2015-06-19</td>
<td>call</td>
<td>AAPL150619C00085000</td>
<td>42.50</td>
<td>41.85</td>
<td>42.40</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

A list-like object containing dates can also be passed to the `expiry` parameter, returning options data for all expiry dates in the list.

In [20]: data = aapl.get_near_stock_price(expiry=aapl.expiry_dates[0:3])
In [21]: data.iloc[0:5:, 0:5]
Out[21]:
<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
<th>Bid</th>
<th>Ask</th>
<th>Chg</th>
<th>PctChg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The month and year parameters can be used to get all options data for a given month.

### 25.3 Google Finance

In [22]: import pandas.io.data as web
In [23]: import datetime
In [24]: start = datetime.datetime(2010, 1, 1)
In [25]: end = datetime.datetime(2013, 1, 27)
In [26]: f = web.DataReader("F", 'google', start, end)
In [27]: f.ix['2010-01-04']
Out[27]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>10.17</td>
</tr>
<tr>
<td>High</td>
<td>10.28</td>
</tr>
<tr>
<td>Low</td>
<td>10.05</td>
</tr>
<tr>
<td>Close</td>
<td>10.28</td>
</tr>
<tr>
<td>Volume</td>
<td>60855796.00</td>
</tr>
<tr>
<td>Name:</td>
<td>2010-01-04 00:00:00, dtype: float64</td>
</tr>
</tbody>
</table>

### 25.4 FRED

In [28]: import pandas.io.data as web
In [29]: import datetime
In [30]: start = datetime.datetime(2010, 1, 1)
In [31]: end = datetime.datetime(2013, 1, 27)
In [32]: gdp=web.DataReader("GDP", "fred", start, end)
In [33]: gdp.ix['2013-01-01']
Out[33]:
| GDP   | 16502.4 |
| Name: | 2013-01-01 00:00:00, dtype: float64 |

# Multiple series:
In [34]: inflation = web.DataReader(["CPIAUCSL", "CPILFESL"], "fred", start, end)
In [35]: inflation.head()
Out[35]:
<table>
<thead>
<tr>
<th>DATE</th>
<th>CPIAUCSL</th>
<th>CPILFESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-01-01</td>
<td>217.488</td>
<td>220.633</td>
</tr>
</tbody>
</table>
2010-02-01 217.281 220.731
2010-03-01 217.353 220.783
2010-04-01 217.403 220.822
2010-05-01 217.290 220.962

25.5 Fama/French

Dataset names are listed at Fama/French Data Library.

In [36]: import pandas.io.data as web

In [37]: ip = web.DataReader("5_Industry_Portfolios", "famafrench")

In [38]: ip[4].ix[192607]
Out[38]:
1 Cnsmr 5.43
2 Manuf 2.73
3 HiTec 1.83
4 Hlth 1.77
5 Other 2.16
Name: 192607, dtype: float64

25.6 World Bank

Pandas users can easily access thousands of panel data series from the World Bank’s World Development Indicators by using the wb I/O functions.

25.6.1 Indicators

Either from exploring the World Bank site, or using the search function included, every world bank indicator is accessible.

For example, if you wanted to compare the Gross Domestic Products per capita in constant dollars in North America, you would use the search function:

In [1]: from pandas.io import wb

In [2]: wb.search('gdp.*capita.*const').iloc[:,:2]
Out[2]:
   id        name
3242 GDPPCKD GDP per Capita, constant US$, millions
5143 NY.GDP.PCAP.KD GDP per capita (constant 2005 US$)
5145 NY.GDP.PCAP.KN GDP per capita (constant LCU)
5147 NY.GDP.PCAP.PP.KD GDP per capita, PPP (constant 2005 internation...

Then you would use the download function to acquire the data from the World Bank’s servers:

In [3]: dat = wb.download(indicator='NY.GDP.PCAP.KD', country=['US', 'CA', 'MX'], start=2005, end=2008)

In [4]: print(dat)

NY.GDP.PCAP.KD
country  year  value
Canada 2008 36005.5004978584
The resulting dataset is a properly formatted DataFrame with a hierarchical index, so it is easy to apply .groupby transformations to it:

```python
In [6]: dat['NY.GDP.PCAP.KD'].groupby(level=0).mean()
```

```
Out[6]:

<table>
<thead>
<tr>
<th>country</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>35765.569188</td>
</tr>
<tr>
<td>Mexico</td>
<td>7965.245332</td>
</tr>
<tr>
<td>United States</td>
<td>43112.417952</td>
</tr>
</tbody>
</table>

dtype: float64
```

Now imagine you want to compare GDP to the share of people with cellphone contracts around the world.

```python
In [7]: wb.search('cell.*%').iloc[:,:2]
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3990</td>
<td>IT.CEL.SETS.FE.ZS Mobile cellular telephone users, female (% of ...</td>
</tr>
<tr>
<td>3991</td>
<td>IT.CEL.SETS.MA.ZS Mobile cellular telephone users, male (% of po...</td>
</tr>
<tr>
<td>4027</td>
<td>IT.MOB.COV.ZS Population coverage of mobile cellular teleph...</td>
</tr>
</tbody>
</table>
```

Notice that this second search was much faster than the first one because pandas now has a cached list of available data series.

```python
In [13]: ind = ['NY.GDP.PCAP.KD', 'IT.MOB.COV.ZS']
In [14]: dat = wb.download(indicator=ind, country='all', start=2011, end=2011).dropna()
In [15]: dat.columns = ['gdp', 'cellphone']
In [16]: print(dat.tail())
```

```
gdp         cellphone
country year
Swaziland 2011 2413.952853 94.9
Tunisia    2011 3687.340170 100.0
Uganda     2011 405.332501  100.0
Zambia     2011  767.911290  62.0
Zimbabwe   2011 419.236086  72.4
```

Finally, we use the statsmodels package to assess the relationship between our two variables using ordinary least squares regression. Unsurprisingly, populations in rich countries tend to use cellphones at a higher rate:

```python
In [17]: import numpy as np
In [18]: import statsmodels.formula.api as smf
In [19]: mod = smf.ols("cellphone ~ np.log(gdp)", dat).fit()
In [20]: print(mod.summary())
```

```
OLS Regression Results
------------------------------------------------------------------------------
Dep. Variable:           cellphone     R-squared:                       0.297
Model:                  OLS              Adj. R-squared:                  0.274
Method:                 Least Squares   F-statistic:                    13.08
```

808 Chapter 25. Remote Data Access
Date: Thu, 25 Jul 2013    Prob (F-statistic): 0.00105
Time: 15:24:42    Log-Likelihood: -139.16
No. Observations: 33    AIC: 282.3
Df Residuals: 31    BIC: 285.3
Df Model: 1

===============================================================================
coef    std err    t    P>|t|    [95.0% Conf. Int.]
===============================================================================
Intercept 16.5110 19.071 0.866 0.393 -22.384 55.406
np.log(gdp) 9.9333 2.747 3.616 0.001 4.331 15.535
===============================================================================
Omnibus: 36.054    Durbin-Watson: 2.071
Prob(Omnibus): 0.000    Jarque-Bera (JB): 119.133
Skew: -2.314    Prob(JB): 1.35e-26
Kurtosis: 11.077    Cond. No. 45.8

25.6.2 Country Codes

New in version 0.15.1.

The `country` argument accepts a string or list of mixed two or three character ISO country codes, as well as dynamic World Bank exceptions to the ISO standards.

For a list of the the hard-coded country codes (used solely for error handling logic) see `pandas.io.wb.country_codes`.

25.6.3 Problematic Country Codes & Indicators

**Note:** The World Bank’s country list and indicators are dynamic. As of 0.15.1, `wb.download()` is more flexible. To achieve this, the warning and exception logic changed.

The world bank converts some country codes in their response, which makes error checking by pandas difficult. Retired indicators still persist in the search.

Given the new flexibility of 0.15.1, improved error handling by the user may be necessary for fringe cases.

To help identify issues:

There are at least 4 kinds of country codes:
1. Standard (2/3 digit ISO) - returns data, will warn and error properly.
2. Non-standard (WB Exceptions) - returns data, but will falsely warn.
3. Blank - silently missing from the response.
4. Bad - causes the entire response from WB to fail, always exception inducing.

There are at least 3 kinds of indicators:
1. Current - Returns data.
2. Retired - Appears in search results, yet won’t return data.
3. Bad - Will not return data.

Use the `errors` argument to control warnings and exceptions. Setting errors to ignore or warn, won’t stop failed responses. (ie, 100% bad indicators, or a single “bad” (#4 above) country code).
25.7 Google Analytics

The `ga` module provides a wrapper for Google Analytics API to simplify retrieving traffic data. Result sets are parsed into a pandas DataFrame with a shape and data types derived from the source table.

25.7.1 Configuring Access to Google Analytics

The first thing you need to do is to setup accesses to Google Analytics API. Follow the steps below:

1. **In the Google Developers Console**
   - (a) enable the Analytics API
   - (b) create a new project
   - (c) create a new Client ID for an “Installed Application” (in the “APIs & auth / Credentials section” of the newly created project)
   - (d) download it (JSON file)

2. **On your machine**
   - (a) rename it to `client_secrets.json`
   - (b) move it to the `pandas/io` module directory

The first time you use the `read_ga()` function, a browser window will open to ask you to authentify to the Google API. Do proceed.

25.7.2 Using the Google Analytics API

The following will fetch users and pageviews (metrics) data per day of the week, for the first semester of 2014, from a particular property.

```python
import pandas.io.ga as ga

account_id = "2360420",
profile_id = "19462946",
property_id = "UA-2360420-5",
metrics = ['users', 'pageviews'],
dimensions = ['dayOfWeek'],
start_date = "2014-01-01",
end_date = "2014-08-01",
index_col = 0,
filters = "pagePath=~aboutus;ga:country==France",
}
```

The only mandatory arguments are `metrics`, `dimensions` and `start_date`. We strongly recommend that you always specify the `account_id`, `profile_id` and `property_id` to avoid accessing the wrong data bucket in Google Analytics.

The `index_col` argument indicates which dimension(s) has to be taken as index.

The `filters` argument indicates the filtering to apply to the query. In the above example, the page URL has to contain `aboutus` AND the visitors country has to be France.
Detailed information in the following:

- pandas & google analytics, by yhat
- Google Analytics integration in pandas, by Chang She
- Google Analytics Dimensions and Metrics Reference
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 = DataFrame({
'a': randn(1000),
 'b': randn(1000),
'N': 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  181  1.212112 -0.497767  x
5  458 -0.173215  0.837519  x
6  159  0.119209  1.103245  x
..   ...    ...  ...  ..
993 910  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):

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:

In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
611748 function calls (609735 primitive calls) in 0.561 seconds

Ordered by: internal time
List reduced from 102 to 4 due to restriction <4>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1000    0.306    0.000    0.492    0.000 <ipython-input-4-91e33489f136>:1(integrate_f)
552423   0.179    0.000   0.179    0.000 <ipython-input-3-bc41a25943f6>:1(f)
1000    0.007    0.000    0.007    0.000 {range}
3000    0.007    0.000    0.047    0.000 series.py:519(__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:

In [6]: %load_ext cythonmagic

Now, let’s simply copy our functions over to cython as is (the suffix is here to distinguish between function versions):

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)
   ...: 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
   ...

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)
   58325 function calls (56312 primitive calls) in 0.078 seconds
   Ordered by: internal time
   List reduced from 100 to 4 due to restriction <4>
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    3000  0.008    0.000  0.052    0.000 series.py:519(__getitem__)
    3000  0.007    0.000  0.017    0.000 internals.py:3481(get_values)
    3000  0.007    0.000  0.039    0.000 index.py:1582(get_value)
    6000  0.007    0.000  0.026    0.000 {pandas.lib.values_from_object}

26.1.4 Using ndarray

It’s calling series... a lot! It’s creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in python, so maybe we could 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):

26.1. Cython (Writing C extensions for pandas)
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:** In 0.13.0 since Series has been factored to no longer sub-class ndarray but instead subclass NDFrame, 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.

Prior to 0.13.0

```
apply_integrate_f(df['a'], df['b'], df['N'])
```

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:

```
In [11]: %prun -l 4 apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
190 function calls in 0.003 seconds
```

Ordered by: internal time
List reduced from 49 to 4 due to restriction <4>

```
n calls  tottime  percall  cumtime  percall  filename:lineno(function)
1     0.002  0.002   0.002   0.002  _cython_magic_073e22cb442403aaff864a05d833c10b.apply_f
3     0.000  0.000   0.000   0.000  internals.py:2872(iget)
6     0.000  0.000   0.000   0.000  generic.py:2152(isetattr__)
3     0.000  0.000   0.000   0.000  generic.py:1079(_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:

```
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 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(n):
    ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
    ....:     return res
```

```
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.

We simply take the plain python code from above and annotate with the @jit decorator.

```
import numba

@numba.jit
def f_plain(x):
```
```python
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
```

```python
@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
```

```python
def compute_numba(df):
    result = apply_integrate_f_numba(df['a'].values, df['b'].values, df['N'].values)
    return Series(result, index=df.index, name='result')
```

Similar to above, we directly pass `numpy` arrays directly to the numba function. Further we are wrapping the results to provide a nice interface by passing/returning `pandas` objects.

In [4]: %%timeit compute_numba(df)
1000 loops, best of 3: 798 us per loop

Read more in the numba docs.

### 26.3 Expression Evaluation via `eval()` (Experimental)

New in version 0.13.

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., \( df + 2 \times \pi / s ** 4 \% 42 - \text{the\_golden\_ratio} \)
- Comparison operations, including chained comparisons, e.g., \( 2 < df < df2 \)
- Boolean operations, e.g., \( df < df2 \) and \( df3 < df4 \) or \( \text{not df\_bool} \)
- 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., \( \text{pd.eval('df')} \) (this is not very useful)

This Python syntax is not allowed:

- Expressions
  - Function calls
  - 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:

In [13]: import pandas as pd
In [14]: from pandas import DataFrame, Series
In [15]: from numpy.random import randn
In [16]: import numpy as np
In [17]: nrows, ncols = 20000, 100
In [18]: df1, df2, df3, df4 = [DataFrame(randn(nrows, ncols)) for _ in range(4)]

Now let’s compare adding them together using plain ol’ Python versus eval():
In [19]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 24.4 ms per loop

In [20]: %timeit pd.eval('df1 + df2 + df3 + df4')
100 loops, best of 3: 13.8 ms per loop

Now let’s do the same thing but with comparisons:

In [21]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
10 loops, best of 3: 71.4 ms per loop

In [22]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
10 loops, best of 3: 24.5 ms per loop

eval() also works with unaligned pandas objects:

In [23]: s = Series(randn(50))

In [24]: %timeit df1 + df2 + df3 + df4 + s
10 loops, best of 3: 80.5 ms per loop

In [25]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
10 loops, best of 3: 57 ms per loop

Note: Operations such as

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 (Experimental)

New in version 0.13.

In addition to the top level pandas.eval() function you can also evaluate an expression in the “context” of a DataFrame.

In [26]: df = DataFrame(randn(5, 2), columns=['a', 'b'])

In [27]: df.eval('a + b')
Out[27]:
   0     -0.246747
   1      0.867786
   2     -1.626063
   3     -1.134978
   4     -1.027798
dtype: float64

Any expression that is a valid pandas.eval() expression is also a valid DataFrame.eval() expression, with the added benefit that you don’t have to prefix the name of the DataFrame to the column(s) you’re interested in evaluating.
In addition, you can perform assignment of columns within an expression. This allows for formulaic evaluation. Only a single assignment is permitted. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

```
In [28]: df = DataFrame(dict(a=range(5), b=range(5, 10)))
In [29]: df.eval('c = a + b')
In [30]: df.eval('d = a + b + c')
In [31]: df.eval('a = 1')
In [32]: df
Out[32]:
    a  b  c  d
0   1  5  5  10
1   1  6  7  14
2   1  7  9  18
3   1  8 11  22
4   1  9 13  26
```

The equivalent in standard Python would be

```
In [33]: df = DataFrame(dict(a=range(5), b=range(5, 10)))
In [34]: df['c'] = df.a + df.b
In [35]: df['d'] = df.a + df.b + df.c
In [36]: df['a'] = 1
In [37]: df
Out[37]:
    a  b  c  d
0   1  5  5  10
1   1  6  7  14
2   1  7  9  18
3   1  8 11  22
4   1  9 13  26
```

### 26.3.4 Local Variables

In pandas version 0.14 the local variable API has changed. In pandas 0.13.x, you could refer to local variables the same way you would in standard Python. For example,

```python
df = DataFrame(randn(5, 2), columns=['a', 'b'])
newcol = randn(len(df))
df.eval('b + newcol')
```

```
UndefinedVariableError: name 'newcol' is not defined
```

As you can see from the exception generated, this syntax is no longer allowed. 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,

```
In [38]: df = DataFrame(randn(5, 2), columns=list('ab'))
In [39]: newcol = randn(len(df))
```
In [40]: df.eval('b + @newcol')
Out[40]:
0  -0.173926
1   2.493083
2  -0.881831
3  -0.691045
4   1.334703
dtype: float64

In [41]: df.query('b < @newcol')
Out[41]:
   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 [42]: a = randn()

In [43]: df.query('@a < a')
Out[43]:
   a   b
0  0.863987 -0.115998

In [44]: df.loc[a < df.a]  # same as the previous expression
Out[44]:
   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 [45]: a, b = 1, 2

In [46]: pd.eval('@a + b')
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 [47]: pd.eval('a + b')
Out[47]: 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 [48]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'
In [49]: x = pd.eval(expr, parser='python')
In [50]: expr_no_parens = 'df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0'
In [51]: y = pd.eval(expr_no_parens, parser='pandas')
In [52]: np.all(x == y)
Out[52]: True

The same expression can be “anded” together with the word `and` as well:
In [53]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'
In [54]: x = pd.eval(expr, parser='python')
In [55]: expr_with_ands = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'
In [56]: y = pd.eval(expr_with_ands, parser='pandas')
In [57]: np.all(x == y)
Out[57]: True

The `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

In [58]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 24.3 ms per loop
In [59]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
10 loops, best of 3: 26.1 ms per loop

### 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
In [60]: df = DataFrame({'strings': np.repeat(list('cba'), 3),
    ....:                     'nums': np.repeat(range(3), 3)})
    ....:
In [61]: df
Out[61]:
   nums  strings
0    0      c
1    0      c
2    0      c
3    1      b
4    1      b
5    1      b
6    2      a
7    2      a
8    2      a

In [62]: df.query('strings == "a" and nums == 1')
Out[62]:
Empty DataFrame
Columns: [nums, strings]
Index: []

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.
We have implemented “sparse” versions of Series, DataFrame, and Panel. These are not sparse in the typical “mostly 0”. You can view these objects as being “compressed” where any data matching a specific value (NaN/missing by default, 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:

```
In [1]: ts = 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
```

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:

```
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
   6   0.000000
   7   0.000000
   8 -0.861849
   9 -2.104569
dtype: float64
```
The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA DataFrame:

```
In [6]: df = DataFrame(randn(10000, 4))
In [7]: df.ix[: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 NaN  NaN  NaN  NaN
9999  0.280249 -1.648493  1.490865 -0.890819
```

```
[10000 rows x 4 columns]
```

```
In [10]: sdf.density
Out[10]: 0.0001
```

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
1 -0.282863
2  NaN
3  NaN
4  NaN
5  NaN
6  NaN
7  NaN
8 -0.861849
9 -2.104569
dtype: float64
```

828 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:

```python
In [12]: arr = np.random.randn(10)
In [14]: sparr = SparseArray(arr)
In [15]: sparr
Out[15]:
[-1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.606027190513, 1.33421134013]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9])
```

Like the indexed objects (SparseSeries, SparseDataFrame, SparsePanel), a SparseArray can be converted back to a regular ndarray by calling to_dense:

```python
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

SparseList is a list-like data structure for managing a dynamic collection of SparseArrays. To create one, simply call the SparseList constructor with a fill_value (defaulting to NaN):

```python
In [17]: spl = SparseList()
In [18]: spl
Out[18]: <pandas.sparse.list.SparseList object at 0x9c67e74c>
```

The two important methods are append and to_array. append can accept scalar values or any 1-dimensional sequence:

```python
In [19]: from numpy import nan
In [20]: spl.append(np.array([1., nan, nan, 2., 3.]))
In [21]: spl.append(5)
In [22]: spl.append(sparr)
In [23]: spl
Out[23]:
<pandas.sparse.list.SparseList object at 0x9c67e74c>
[1.0, nan, nan, 2.0, 3.0]
Fill: nan
IntIndex
Indices: array([0, 3, 4])
[5.0]
```
As you can see, all of the contents are stored internally as a list of memory-efficient SparseArray objects. Once you’ve accumulated all of the data, you can call to_array to get a single SparseArray with all the data:

```
In [24]: spl.to_array()
Out[24]:
[1.0, nan, nan, 2.0, 3.0, 5.0, -1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.606027190513, 1.33421134013]
```

### 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 Interaction with scipy.sparse

Experimental api to transform between sparse pandas and scipy.sparse structures.

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.

```
In [25]: from numpy import nan

In [26]: s = Series([3.0, nan, 1.0, 3.0, nan, nan])

In [27]: 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 [28]: s
Out[28]:
A     B     C     D
1 2 a     0     3
     1       NaN
1 0 b     0     1
     1       3
```
# SparseSeries

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.

```python
In [31]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
                               column_levels=['C', 'D'],
                               sort_labels=True)
```

In [32]: A

```
<3x4 sparse matrix of type '<type 'numpy.float64'>'
with 3 stored elements in COOrdinate format>
```

```
In [33]: A.todense()
Out[33]:
matrix([[0., 0., 1., 3.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

```
In [34]: rows
Out[34]: [(1L, 1L), (1L, 2L), (2L, 1L)]
```

```
In [35]: columns
Out[35]: [('a', 0L), ('a', 1L), ('b', 0L), ('b', 1L)]
```

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

```python
In [36]: A, rows, columns = ss.to_coo(row_levels=['A', 'B', 'C'],
                               column_levels=['D'],
                               sort_labels=False)
```

In [37]: A

```
<3x2 sparse matrix of type '<type 'numpy.float64'>'
with 3 stored elements in COOrdinate format>
```
In [38]: A.todense()
Out[38]:
matrix([[ 3.,  0.],
        [ 1.,  3.],
        [ 0.,  0.]]

In [39]: rows
Out[39]: [(1L, 2L, 'a'), (1L, 1L, 'b'), (2L, 1L, 'b')]

In [40]: columns
Out[40]: [0, 1]

A convenience method `SparseSeries.from_coo()` is implemented for creating a SparseSeries from a scipy.sparse.coo_matrix.

In [41]: from scipy import sparse

In [42]: A = sparse.coo_matrix(((3.0, 1.0, 2.0), ([1, 0, 0], [0, 2, 3])),
                           shape=(3, 4))

In [43]: A
Out[43]:
<3x4 sparse matrix of type '<type 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

In [44]: A.todense()
Out[44]:
matrix([[ 0., 0., 1., 2.],
        [ 3., 0., 0., 0.],
        [ 0., 0., 0., 0.]]

The default behaviour (with dense_index=False) simply returns a SparseSeries containing only the non-null entries.

In [45]: ss = SparseSeries.from_coo(A)

In [46]: ss
Out[46]:
0  2  1
  3  2
1  0  3
dtype: float64
BlockIndex
Block locations: array([0])
Block lengths: array([3])

Specifying dense_index=True will result in an index that is the Cartesian product of the row and columns coordinates of the matrix. Note that this will consume a significant amount of memory (relative to dense_index=False) if the sparse matrix is large (and sparse) enough.

In [47]: ss_dense = SparseSeries.from_coo(A, dense_index=True)

In [48]: ss_dense
Out[48]:
0  0  NaN
  1  NaN
  2  1
  3  2
```
1  0  3
1  NaN
2  NaN
3  NaN
2  0  NaN
1  NaN
2  NaN
3  NaN

dtype: float64
BlockIndex
Block locations: array([2])
Block lengths: array([3])
```
28.1 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 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():

```
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
```
28.1.1 Bitwise boolean

Bitwise boolean operators like `==` and `!=` will 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.

28.1.2 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.

28.2 NaN, Integer NA values and NA type promotions

28.2.1 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
- 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 `isnull` and `notnull` 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.

28.2.2 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 [5]: s = Series([1, 2, 3, 4, 5], index=list('abcde'))

In [6]: s
Out[6]:
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.

### 28.2.3 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, in practice I have found very few cases where this is an issue in practice. Some explanation for the motivation here in the next section.

### 28.2.4 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></td>
<td>int8, int16, int32, int64</td>
</tr>
<tr>
<td>numpy.integer</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.

28.3 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 .ix. The following code will generate exceptions:

```python
s = Series(range(5))
s[-1]
df = DataFrame(np.random.randn(5, 4))
df
df.ix[-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).

28.4 Label-based slicing conventions

28.4.1 Non-monotonic indexes require exact matches

28.4.2 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 [11]: s = Series(randn(6), index=list('abcdef'))
In [12]: s
Out[12]:
   a   -0.345411
   b    1.721799
   c    0.171342
   d    1.222367
   e    1.228721
   f    0.549175
   dtype: float64
```

Suppose we wished to slice from c to e, using integers this would be

```python
In [13]: s[2:5]
Out[13]:
   c    0.171342
   d    1.222367
```
e  1.228721
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:

s.ix['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 design to make label-based slicing include both endpoints:

In [14]: s.ix['c':'e']
Out[14]:
c   0.171342
d  1.222367
e  1.228721
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.

## 28.5 Miscellaneous indexing gotchas

### 28.5.1 Reindex versus ix gotchas

Many users will find themselves using the ix indexing capabilities as a concise means of selecting data from a pandas object:

In [15]: df = DataFrame(randn(6, 4), columns=['one', 'two', 'three', 'four'],
   index=list('abcdef'))

In [16]: df
Out[16]:
   one   two   three    four
a -1.982099 -0.366112 -0.228622 -1.663680
b  0.527377 -1.428764 -0.177802  0.382121
c -0.049456  0.556557  0.993878 -0.433240
d -0.077343  1.052958  1.528472  0.644673
e -1.261108  1.265039  0.424791  0.385124
f -1.176251 -0.074802 -0.384239  1.075475

In [17]: df.ix[['b', 'c', 'e']]
Out[17]:
   one   two   three    four
b  0.527377 -1.428764 -0.177802  0.382121
c -0.049456  0.556557  0.993878 -0.433240
e -1.261108  1.265039  0.424791  0.385124

This is, of course, completely equivalent in this case to using the reindex method:

In [18]: df.reindex(['b', 'c', 'e'])
Out[18]:
   one   two   three    four
b  0.527377 -1.428764 -0.177802  0.382121
c -0.049456  0.556557  0.993878 -0.433240
e -1.261108  1.265039  0.424791  0.385124
Some might conclude that `ix` and `reindex` are 100% equivalent based on this. This is indeed true except in the case of integer indexing. For example, the above operation could alternately have been expressed as:

```
In [19]: df.ix[[1, 2, 4]]
Out[19]:
    one   two  three   four
b  0.527377 -1.428764 -0.177802  0.382121
c -0.049456  0.556557  0.993878 -0.433240
e -1.261108  1.265039  0.424791  0.385124
```

If you pass `[1, 2, 4]` to `reindex` you will get another thing entirely:

```
In [20]: df.reindex([1, 2, 4])
Out[20]:
     one   two  three   four
   1   NaN   NaN   NaN   NaN
   2   NaN   NaN   NaN   NaN
   4   NaN   NaN   NaN   NaN
```

So it’s important to remember that `reindex` is strict label indexing only. This can lead to some potentially surprising results in pathological cases where an index contains, say, both integers and strings:

```
In [21]: s = Series([1, 2, 3], index=['a', 0, 1])
In [22]: s
Out[22]:
     a   1
     0   2
     1   3
dtype: int64
In [23]: s.ix[[0, 1]]
Out[23]:
     0   2
     1   3
dtype: int64
In [24]: s.reindex([0, 1])
Out[24]:
     0   2
     1   3
dtype: int64
```

Because the index in this case does not contain solely integers, `ix` falls back on integer indexing. By contrast, `reindex` only looks for the values passed in the index, thus finding the integers 0 and 1. While it would be possible to insert some logic to check whether a passed sequence is all contained in the index, that logic would exact a very high cost in large data sets.

### 28.5.2 Reindex potentially changes underlying Series dtype

The use of `reindex_like` can potentially change the dtype of a `Series`.

```
In [25]: series = Series([1, 2, 3])
In [26]: x = Series([True])
In [27]: x.dtype
dtype('bool')
```
In [28]: x = Series([True]).reindex_like(series)

In [29]: x.dtype
Out[29]: dtype('O')

This is because reindex_like 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.

### 28.6 Timestamp limitations

#### 28.6.1 Minimum and maximum timestamps

Since pandas represents timestamps in nanosecond resolution, the timespan that can be represented using a 64-bit integer is limited to approximately 584 years:

In [30]: begin = Timestamp.min

In [31]: begin
Out[31]: Timestamp('1677-09-22 00:12:43.145225')

In [32]: end = Timestamp.max

In [33]: end
Out[33]: Timestamp('2262-04-11 23:47:16.854775807')

See [here](#) for ways to represent data outside these bound.

### 28.7 Parsing Dates from Text Files

When parsing multiple text file columns into a single date column, the new date column is prepended to the data and then index_col specification is indexed off of the new set of columns rather than the original ones:

In [34]: print(open('tmp.csv').read())
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

In [35]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [36]: df = read_csv('tmp.csv', header=None,
       ....:   parse_dates=date_spec,
       ....:   keep_date_col=True,
       ....:   index_col=0)

# index_col=0 refers to the combined column "nominal" and not the original
# first column of 'KORD' strings
In [37]: df
28.8 Differences with NumPy

For Series and DataFrame objects, \( \text{var} \) normalizes by \( N-1 \) to produce unbiased estimates of the sample variance, while NumPy's \( \text{var} \) normalizes by \( N \), which measures the variance of the sample. Note that \( \text{cov} \) normalizes by \( N-1 \) in both pandas and NumPy.

28.9 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.

28.10 HTML Table Parsing

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function \( \text{read_html} \).

**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.

**Issues with using Anaconda**

- Anaconda ships with lxml version 3.2.0; the following workaround for Anaconda was successfully used to deal with the versioning issues surrounding lxml and BeautifulSoup4.

**Note:** Unless you have both:

- A strong restriction on the upper bound of the runtime of some code that incorporates read_html()
- Complete knowledge that the HTML you will be parsing will be 100% valid at all times

then you should install html5lib and things will work swimmingly without you having to muck around with conda. If you want the best of both worlds then install both html5lib and lxml. If you do install lxml then you need to perform the following commands to ensure that lxml will work correctly:

```bash
# remove the included version
conda remove lxml

# install the latest version of lxml
pip install 'git+git://github.com/lxml/lxml.git'

# install the latest version of beautifulsoup4
pip install 'bzr+lp:beautifulsoup4'
```

Note that you need bzf and git installed to perform the last two operations.

**28.11 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 [38]: x = np.array(list(range(10)), '>i4')  # big endian

In [39]: newx = x.byteswap().newbyteorder()  # force native byteorder

In [40]: s = Series(newx)

See the NumPy documentation on byte order for more details.
CHAPTER TWENTYNINE

RPY2 / R INTERFACE

Warning: In v0.16.0, the pandas.rpy interface has been deprecated and will be removed in a future version. Similar functionality can be accessed through the rpy2 project. See the updating section for a guide to port your code from the pandas.rpy to rpy2 functions.

29.1 Updating your code to use rpy2 functions

In v0.16.0, the pandas.rpy module has been deprecated and users are pointed to the similar functionality in rpy2 itself (rpy2 >= 2.4).

Instead of importing import pandas.rpy.common as com, the following imports should be done to activate the pandas conversion support in rpy2:

```python
from rpy2.robjects import pandas2ri
pandas2ri.activate()
```

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(). So these functions can be used to replace the existing functions in pandas:

- `com.convert_to_r_dataframe(df)` should be replaced with `pandas2ri.py2ri(df)`
- `com.convert_robj(rdf)` should be replaced with `pandas2ri.ri2py(rdf)`

Note: these functions are for the latest version (rpy2 2.5.x) and were called pandas2ri.pandas2ri() and pandas2ri.ri2pandas() previously.

Some of the other functionality in pandas.rpy can be replaced easily as well. For example to load R data as done with the load_data function, the current method:

```python
df_iris = com.load_data('iris')
```

can be replaced with:

```python
from rpy2.robjects import r
r.data('iris')
df_iris = pandas2ri.ri2py(r[name])
```

The convert_to_r_matrix function can be replaced by the normal pandas2ri.py2ri to convert dataframes, with a subsequent call to R as.matrix function.
Warning: Not all conversion functions in rpy2 are working exactly the same as the current methods in pandas. If you experience problems or limitations in comparison to the ones in pandas, please report this at the issue tracker.

See also the documentation of the rpy2 project.

29.2 R interface with rpy2

If your computer has R and rpy2 (> 2.2) installed (which will be left to the reader), you will be able to leverage the below functionality. On Windows, doing this is quite an ordeal at the moment, but users on Unix-like systems should find it quite easy. rpy2 evolves in time, and is currently reaching its release 2.3, while the current interface is designed for the 2.2.x series. We recommend to use 2.2.x over other series unless you are prepared to fix parts of the code, yet the rpy2-2.3.0 introduces improvements such as a better R-Python bridge memory management layer so it might be a good idea to bite the bullet and submit patches for the few minor differences that need to be fixed.

# if installing for the first time
hg clone http://bitbucket.org/lgautier/rpy2

cd rpy2
hg pull
hg update version_2.2.x
sudo python setup.py install

Note: To use R packages with this interface, you will need to install them inside R yourself. At the moment it cannot install them for you.

Once you have done installed R and rpy2, you should be able to import pandas.rpy.common without a hitch.

29.3 Transferring R data sets into Python

The load_data function retrieves an R data set and converts it to the appropriate pandas object (most likely a DataFrame):

In [1]: import pandas.rpy.common as com

In [2]: infert = com.load_data('infert')

In [3]: infert.head()

Out[3]:

<table>
<thead>
<tr>
<th>education</th>
<th>age</th>
<th>parity</th>
<th>induced</th>
<th>case</th>
<th>spontaneous</th>
<th>stratum</th>
<th>pooled.stratum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0-5yrs</td>
<td>26</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2 0-5yrs</td>
<td>42</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3 0-5yrs</td>
<td>39</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4 0-5yrs</td>
<td>34</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5 6-11yrs</td>
<td>35</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>32</td>
</tr>
</tbody>
</table>

29.4 Converting DataFrames into R objects

New in version 0.8.
Starting from pandas 0.8, there is experimental support to convert DataFrames into the equivalent R object (that is, data.frame):

```
In [4]: from pandas import DataFrame

In [5]: df = DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C':[7,8,9]},
                    index=['one', 'two', 'three'])

In [6]: r_dataframe = com.convert_to_r_dataframe(df)

In [7]: print(type(r_dataframe))
<class '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.

You can also use convert_to_r_matrix to obtain a Matrix instance, but bear in mind that it will only work with homogeneously-typed DataFrames (as R matrices bear no information on the data type):

```
In [9]: r_matrix = com.convert_to_r_matrix(df)

In [10]: print(type(r_matrix))
<class 'rpy2.robjects.vectors.Matrix'>

In [11]: print(r_matrix)
   A B C
one 1 4 7
two 2 5 8
three 3 6 9
```

29.5 Calling R functions with pandas objects

29.6 High-level interface to R estimators
CHAPTER
THIRTY

PANDAS ECOSYSTEM

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. It has great support for pandas data objects.

### 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 quandl/Python

Quandl API for Python wraps the Quandl REST API to return Pandas DataFrames with timeseries indexes.
30.4.2 pydatastream

PyDatastream is a Python interface to the Thomson Datworks 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).

30.4.3 pandaSDMX

pandaSDMX is an extensible library to retrieve and acquire statistical data and metadata disseminated in SDMX 2.1. This standard is currently supported by the European statistics office (Eurostat) and the European Central Bank (ECB). Datasets may be returned as pandas Series or multi-indexed DataFrames.

30.4.4 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 xray

xray 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 Blaze

Blaze provides a standard API for doing computations with various in-memory and on-disk backends: NumPy, Pandas, SQLAlchemy, MongoDB, PyTables, PySpark.
CHAPTER
THIRTYONE

COMPARISON WITH R / R LIBRARIES

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

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

This page is also here to offer a bit of a translation guide for users of these R packages.

For transfer of DataFrame objects from pandas to R, one option is to use HDF5 files, see External Compatibility for an example.

31.1 Base R

31.1.1 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

```
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
6  1.075770  1.643563
```
In [3]: df.loc[:, ['a', 'c']]
Out[3]:
    a       c
0 -1.039575 -0.424972
1  0.567020 -1.087401
2 -0.673690 -1.478427
3  0.524988 -0.472035
4 -1.715002 -0.370647
5 -1.157892  0.844885
6  1.075770  1.643563
7 -1.469388 -0.674600
8 -1.776904 -1.294524
9  0.413738 -0.472035

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]:
     a       b       c       d       e       f       g       h       i       j
0  2.565646  1.431256  1.340309  0.875906 -2.211372  0.974466 -2.006747
1 -0.097883  0.695775  0.341734 -1.743161 -0.826591 -0.345352  1.314232
2 -0.082240 -2.182937  0.380396  1.266143  0.299368 -0.863838  0.408204
3 -0.489682  0.369374 -0.034571  0.221471 -0.744471  0.758527  1.729689
4  0.901805  1.171216  0.520260  0.650776 -1.461665 -1.137707 -0.891060
5 -0.260838  0.281957  1.523962 -0.008434  1.952541 -1.056652  0.533946
6  0.576897  1.146000  1.487349  2.015523 -1.833722  1.771740 -0.670027
7  0.065624  0.307665 -1.898358  1.389045 -0.873585 -0.699862  0.812477
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.

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

In [10]: g = df.groupby(['by1','by2'])

In [11]: g[['v1','v2']].mean()
Out[11]:
   by1  by2
v1  
  1   95   5   55
  99   5   55
```

31.1.2 aggregate
For more details and examples see the groupby documentation.

### 31.1.3 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:

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

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

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

The `apply()` method can be used to replicate this:

```python
In [14]: s = pd.Series(np.arange(5), dtype=np.float32)
In [15]: pd.Series(pd.match(s, [2,4], np.nan))
Out[15]:
0   NaN
1   NaN
2     0
3   NaN
4     1
dtype: float64
```

For more details and examples see the reshaping documentation.

### 31.1.4 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`:
baseball <-
  data.frame(team = gl(5, 5,
        labels = paste("Team", LETTERS[1:5]),
        player = sample(letters, 25),
        batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball.example$team, max)

In pandas we may use \texttt{pivot\_table()} method to handle this:

\begin{Verbatim}
In [16]: \texttt{import random}

In [17]: \texttt{import string}

In [18]: 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 [19]: baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)
\end{Verbatim}

\begin{verbatim}
Out[19]:
   team   
team 1 0.394457
team 2 0.395730
team 3 0.343015
team 4 0.388863
team 5 0.377379
Name: batting avg, dtype: float64
\end{verbatim}

For more details and examples see the \textit{reshaping documentation}.

\subsection{31.1.5 subset}

New in version 0.13.

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

\begin{verbatim}
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,]  # note the comma
\end{verbatim}

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

\begin{Verbatim}
In [20]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [21]: df.query('a <= b')
\end{Verbatim}

\begin{verbatim}
Out[21]:
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
\end{verbatim}
In [22]: df[df.a <= df.b]
Out[22]:
    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

In [23]: df.loc[df.a <= df.b]
Out[23]:
    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

For more details and examples see the query documentation.

31.1.6 with

New in version 0.13.

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:

In [24]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [25]: df.eval('a + b')
Out[25]:
   0   1   2   3   4   5   6   7   8   9
0 -0.920205 -0.860236 1.154370 0.188140 -1.163718 0.001397 -0.825694 -1.138198 -1.708034 1.148616
dtype: float64

In [26]: df.a + df.b  # same as the previous expression
Out[26]:
   0   1   2   3   4   5   6   7   8   9
0 -0.920205 -0.860236 1.154370 0.188140 -1.163718 0.001397 -0.825694 -1.138198 -1.708034 1.148616
In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see the eval documentation.

### 31.2 zoo

### 31.3 xts

### 31.4 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>

#### 31.4.1 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 [27]: 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 [28]: grouped = df.groupby([\'month\', \'week\'])
```
In [29]: print grouped['x'].agg([np.mean, np.std])

    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.

31.5 reshape / reshape2

31.5.1 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.

In [30]: a = np.array(list(range(1,24))+[np.NAN]).reshape(2,3,4)

In [31]: pd.DataFrame([tuple(list(x)+[val]) for x, val in np.ndenumerate(a)])

Out[31]:
    0 1 2 3
  0 0 0 0 1
  1 0 0 1 2
  2 0 0 2 3
  3 0 0 3 4
  4 0 1 0 5
  5 0 1 1 6
  6 0 1 2 7
  ... ... ...
  17 1 1 1 18
  18 1 1 2 19
  19 1 1 3 20
  20 1 2 0 21
  21 1 2 1 22
  22 1 2 2 23
  23 1 2 3 NaN

[24 rows x 4 columns]

31.5.2 melt.list

An expression using a list called `a` in R where you want to melt it into a data.frame:
a <- as.list(c(1:4, NA))
data.frame(melt(a))

In Python, this list would be a list of tuples, so DataFrame() method would convert it to a dataframe as required.

In [32]: a = list(enumerate(list(range(1,5))+[np.NAN]))

In [33]: pd.DataFrame(a)
Out[33]:
   0 1
0  0 1
1  1 2
2  2 3
3  3 4
4  4 NaN

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

31.5.3 melt.data.frame

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

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

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

In [34]: cheese = pd.DataFrame({'first' : ['John', 'Mary'],
...:                       'last' : ['Doe', 'Bo'],
...:                       'height' : [5.5, 6.0],
...:                       'weight' : [130, 150]})

In [35]: pd.melt(cheese, id_vars=['first', 'last'])
Out[35]:
   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 [36]: cheese.set_index(['first', 'last']).stack() # alternative way
Out[36]:
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.
31.5.4 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()`:

```python
In [37]: 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 [38]: mdf = pd.melt(df, id_vars=['month', 'week'])
In [39]: pd.pivot_table(mdf, values='value', index=['variable', 'week'],
    ....:                 columns=['month'], aggfunc=np.mean)
Out[39]:
   month 5  6  7
variable week
   x 1 114.001700 132.227290  65.808204
   2 124.669553 147.495706  82.882820
   y 1 225.636630 301.864228  91.706834
   2  57.692665 215.851669 218.004383
   z 1  17.793871  7.124644 17.679823
   2 15.068355 13.873974  9.394966
```

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 [40]: df = pd.DataFrame(
    ....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
    ....:     'Amount': [10, 7, 4, 2, 5, 6, 2]
    ....: )
    ....:
In [41]: df.pivot_table(values='Amount', index=['Animal'], columns=['FeedType'], aggfunc=np.sum)
Out[41]:
    FeedType
Animal    A  B
Animal1  14  5
Animal2  22 21
Animal3  11  6
```

...
In [41]: df.pivot_table(values='Amount', index='Animal', columns='FeedType', aggfunc='sum')
Out[41]:
        FeedType
Animal      A   B
Animal1  10   5
Animal2   2  13
Animal3   6  NaN

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

In [42]: df.groupby(['Animal', 'FeedType'])['Amount'].sum()
Out[42]:
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.

### 31.5.5 factor

New in version 0.15.

pandas has a data type for categorical data.

```python
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 [43]: pd.cut(pd.Series([1,2,3,4,5,6]), 3)
Out[43]:
0   (0.995, 2.667]
1   (0.995, 2.667]
2    (2.667, 4.333]
3    (2.667, 4.333]
4     (4.333, 6]
5      (4.333, 6]
dtype: category
Categories (3, object): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6]]

In [44]: pd.Series([1,2,3,2,2,3]).astype("category")
Out[44]:
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.
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:

```python
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.

```python
In [3]: url = 'https://raw.github.com/pydata/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):

```sql
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:

```python
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
Out[6]:
   total_bill  tip  smoker  time
0   16.99  1.01  No  Dinner
1   10.34  1.66  No  Dinner
2   21.01  3.50  No  Dinner
```
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
dtype: int64

In [10]: tips[is_dinner].head(5)
Out[10]:
   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
```

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;
```

```python
# tips of more than $5.00 at Dinner meals
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
Out[11]:
   total_bill  tip     sex  smoker  day   time  size
23    39.42  7.58      Male       No  Sat  Dinner   4
```
## pandas: powerful Python data analysis toolkit, Release 0.16.2

### Raw Data

<table>
<thead>
<tr>
<th>ID</th>
<th>Total Bill</th>
<th>Tip</th>
<th>Sex</th>
<th>Smoker</th>
<th>Day</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>30.40</td>
<td>5.60</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>47</td>
<td>32.40</td>
<td>6.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>52</td>
<td>34.81</td>
<td>5.20</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>59</td>
<td>48.27</td>
<td>6.73</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>116</td>
<td>29.85</td>
<td>5.14</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>5</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>172</td>
<td>7.25</td>
<td>5.15</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>181</td>
<td>23.33</td>
<td>5.65</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>183</td>
<td>23.17</td>
<td>6.50</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>211</td>
<td>25.89</td>
<td>5.16</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</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>214</td>
<td>28.17</td>
<td>6.50</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>239</td>
<td>29.03</td>
<td>5.92</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
</tbody>
</table>

-- tips by parties of at least 5 diners OR bill total was more than $45

```python
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]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>48.27</td>
<td>6.73</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
</tr>
<tr>
<td>125</td>
<td>29.80</td>
<td>4.20</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
</tr>
<tr>
<td>141</td>
<td>34.30</td>
<td>6.70</td>
<td>Male</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</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>
</tr>
<tr>
<td>143</td>
<td>27.05</td>
<td>5.00</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
</tr>
<tr>
<td>155</td>
<td>29.85</td>
<td>5.14</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</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>
</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>
</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>
</tr>
<tr>
<td>185</td>
<td>20.69</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
<tr>
<td>187</td>
<td>30.46</td>
<td>2.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</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>
</tr>
<tr>
<td>216</td>
<td>28.15</td>
<td>3.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
</tr>
</tbody>
</table>
```

### NULL Checking

NULL checking is done using the `notnull()` and `isnull()` methods.

```python
In [14]: frame
Out[14]:

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>NaN</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>D</td>
<td>I</td>
</tr>
</tbody>
</table>
```

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

```sql
SELECT *
FROM frame
WHERE col2 IS NULL;
```
In [15]: frame[frame['col2'].isnull()]
Out[15]:
   col1  col2
0   B   NaN

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

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

In [16]: frame[frame['col1'].notnull()]
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 performed using the similarly named `groupby()` method. `groupby()` typically refers to a process where we’d like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

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

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

The pandas equivalent would be:

```python
In [17]: tips.groupby('sex').size()
Out[17]:
   sex
  Female  87
         Male  157
dtype: int64
```

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

```python
In [18]: tips.groupby('sex').count()
Out[18]:
         total_bill  tip  smoker  day  time  size
   sex
Female  87     87     87    87    87   87
       Male  157    157    157   157   157
```

Alternatively, we could have applied the `count()` method to an individual column:
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]:
        tip
   size  mean
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
```
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).

```python
In [22]:
   df1 = pd.DataFrame({
                      'key': ['A', 'B', 'C', 'D'],
                      'value': np.random.randn(4)})

In [23]:
   df2 = pd.DataFrame({
                      'key': ['B', 'D', 'D', 'E'],
                      'value': np.random.randn(4)})
```

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

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

### 32.4.1 INNER JOIN

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

```python
In [24]: pd.merge(df1, df2, on='key')
```

```python
Out[24]:
          key  value_x  value_y
0         B    1.075416  -0.227314
1         D    1.065735   2.102726
2         D    1.065735  -0.092796
```

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

```python
In [25]: indexed_df2 = df2.set_index('key')

In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)
```

```python
Out[26]:
          key  value_x  value_y
1         B    1.075416  -0.227314
3         D    1.065735   2.102726
3         D    1.065735  -0.092796
```

### 32.4.2 LEFT OUTER JOIN

```sql
-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

```python
In [27]: pd.merge(df1, df2, left_on='key', right_index=True)  # Using right_index=True
```

```python
Out[27]:
          key  value_x  value_y
0         B    1.075416  -0.227314
1         D    1.065735   2.102726
2         D    1.065735  -0.092796
3         B    1.075416  -0.227314
3         D    1.065735  -0.092796
```
# show all records from df1
In [27]: pd.merge(df1, df2, on='key', how='left')
Out[27]:
   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

32.4.3 RIGHT JOIN

-- show all records from df2
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;

# show all records from df2
In [28]: pd.merge(df1, df2, on='key', how='right')
Out[28]:
   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

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).

-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;

# show all records from both frames
In [29]: pd.merge(df1, df2, on='key', how='outer')
Out[29]:
   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

32.5 UNION

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

32.5. UNION
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
*/

In [32]: pd.concat([df1, df2]).drop_duplicates()
Out[32]:
   city rank
0  Chicago   1
1  San Francisco   2
2  New York City   3
0  Chicago   1
1     Boston   4
2  Los Angeles   5

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

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

32.6 UPDATE

32.7 DELETE
33.1 Input/Output

33.1.1 Pickling

`read_pickle(path)` Load pickled pandas object (or any other pickled object) from the specified file path

**pandas.read_pickle**

`pandas.read_pickle(path)`
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

**Parameters**

`path` : string
File path

**Returns**

`unpickled` : type of object stored in file

33.1.2 Flat File

`read_table(filepath_or_buffer[, sep, ...])` Read general delimited file into DataFrame

`read_csv(filepath_or_buffer[, sep, dialect, ...])` Read CSV (comma-separated) file into DataFrame

`read_fwf(filepath_or_buffer[, colspecs, widths])` Read a table of fixed-width formatted lines into DataFrame
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.read_table

pandas.read_table(filepath_or_buffer, sep='\t', dialect=None, compression='infer', doublequote=True, escapechar=None, quotechar='', quoting=0, skipinitialspace=False, lineterminator=None, header='infer', index_col=None, names=None, prefix=None, skiprows=None, skipfooter=None, skip_footer=0, na_values=None, na_fvalues=None, true_values=None, false_values=None, delimiter=None, converters=None, dtype=None, usecols=None, engine=None, delim_whitespace=False, as_recarray=False, na_filter=True, compact_ints=False, use_unicode=False, low_memory=True, buffer_lines=None, warn_bad_lines=True, error_bad_lines=True, keep_default_na=True, thousands=None, comment=None, decimal='.', parse_dates=False, keep_date_col=False, dayfirst=False, date_parser=None, memory_map=False, float_precision=None, nrows=None, iterator=False, chunksize=None, verbose=False, encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_cols=False, infer_datetime_format=False, skip_blank_lines=True)

Read general delimited file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Parameters filepath_or_buffer : 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.csv

sep : string, default t (tab-stop)

Delimiter to use. Regular expressions are accepted.

engine : {'c', 'python'}

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

lineterminator : string (length 1), default None

Character to break file into lines. Only valid with C parser

quotechar : string (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 None

Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3). Default (None) results in QUOTE_MINIMAL behavior.

skipinitialspace : boolean, default False

Skip spaces after delimiter

escapechar : string (length 1), default None

One-character string used to escape delimiter when quoting is QUOTE_NONE.

dtype : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (Unsupported with engine='python')

compression : {'gzip', 'bz2', 'infer', None}, default 'infer'
For on-the-fly decompression of on-disk data. If ‘infer’, then use gzip or bz2 if filepath_or_buffer is a string ending in `.gz` or `.bz2`, respectively, and no decompression otherwise. Set to None for no decompression.

dialect : string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

header : int, list of ints

Row number(s) to use as the column names, and the start of the data. Defaults 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 are 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.

skiprows : list-like or integer

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file

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)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string, default None

Prefix to add to column numbers when no header, e.g ‘X’ for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

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

parse_dates : boolean, list of ints or names, list of lists, or dict

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. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’ A fast-path exists for iso8601-formatted dates.

keep_date_col : boolean, default False
If True and parse_dates specifies combining multiple columns then keep the original columns.

**date_parser** : function

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

**thousands** : str, default None

Thousands separator

**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.

**decimal** : str, default ‘.’

Character to recognize as decimal point. E.g. use ‘,’ for European data

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**iterator** : boolean, default False

Return TextFileReader object

**chunksize** : int, default None

Return TextFileReader object for iteration

**skipfooter** : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c')

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**delimiter** : string, default None

Alternative argument name for sep. Regular expressions are accepted.

**encoding** : string, default None

Encoding to use for UTF when reading/writing (ex. ‘utf-8’). List of Python standard encodings
squeeze : boolean, default False

If the parsed data only contains one column then return a Series

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

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols : boolean, default True

Duplicate columns will be specified as ‘X.0’...’X.N’, rather than ‘X’...’X’

tupleize_cols : boolean, default False

Leave a list of tuples on columns as is (default is to convert to a Multi Index 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. (Only valid with C parser)

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. (Only valid with C parser).

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

skip_blank_lines : boolean, default True

If True, skip over blank lines rather than interpreting as NaN values

Returns result : DataFrame or TextParser

pandas.read_csv

pandas.read_csv(filepath_or_buffer, sep=' ', dialect=None, compression='infer', doublequote=True, escapechar=None, quotechar='', quoting=0, skipinitialspace=False, lineterminator=None, header='infer', index_col=None, names=None, prefix=None, skiprows=None, skipfooter=None, skip_footer=0, na_values=None, keep_default_na=True, thousands=None, comment=None, decimal='.', parse_dates=False, keep_date_col=False, dayfirst=False, date_parser=None, memory_map=False, float_precision=None, nrows=None, iterator=False, chunksize=None, verbose=False, encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_cols=False, infer_datetime_format=False, skip_blank_lines=True)

Read CSV (comma-separated) file into DataFrame

33.1. Input/Output 879
Also supports optionally iterating or breaking of the file into chunks.

**Parameters**

- **filepath_or_buffer**: 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.csv

- **sep**: string, default ‘,’
  
  Delimiter to use. If sep is None, will try to automatically determine this. Regular expressions are accepted.

- **engine**: {‘c’, ‘python’}
  
  Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

- **lineterminator**: string (length 1), default None
  
  Character to break file into lines. Only valid with C parser

- **quotechar**: string (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 None
  
  Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3). Default (None) results in QUOTE_MINIMAL behavior.

- **skipinitialspace**: boolean, default False
  
  Skip spaces after delimiter

- **escapechar**: string (length 1), default None
  
  One-character string used to escape delimiter when quoting is QUOTE_NONE.

- **dtype**: Type name or dict of column -> type
  
  Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (Unsupported with engine=’python’)

- **compression**: {‘gzip’, ‘bz2’, ‘infer’, None}, default ‘infer’
  
  For on-the-fly decompression of on-disk data. If ‘infer’, then use gzip or bz2 if filepath_or_buffer is a string ending in ‘.gz’ or ‘.bz2’, respectively, and no decompression otherwise. Set to None for no decompression.

- **dialect**: string or csv.Dialect instance, default None
  
  If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

- **header**: int, list of ints
  
  Row number(s) to use as the column names, and the start of the data. Defaults 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 are 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.
skiprows : list-like or integer

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file

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)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string, default None

Prefix to add to column numbers when no header, e.g ‘X’ for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

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

parse_dates : boolean, list of ints or names, list of lists, or dict

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. {‘foo’ : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’ A fast-path exists for iso8601-formatted dates.

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : function

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

thousands : str, default 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.

decimal : str, default ‘.’

Character to recognize as decimal point. E.g. use ‘,’ for European data

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

iterator : boolean, default False

Return TextFileReader object

chunksize : int, default None

Return TextFileReader object for iteration

skipfooter : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c')

converters : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers
or column labels

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

delimiter : string, default None

Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None

Encoding to use for UTF when reading/writing (ex. ‘utf-8’). List of Python standard
encodings

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

na_filter : boolean, default True

Detect missing value markers (empty strings and the value of na_values). In data with-
out any NAs, passing na_filter=False can improve the performance of reading a large
file

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory
usage.

mangle_dupe_cols : boolean, default True

Duplicate columns will be specified as ‘X.0’...'X.N’, rather than ‘X'...'X'
tupleize_cols : boolean, default False
Leave a list of tuples on columns as is (default is to convert to a Multi Index 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. (Only valid with C parser)

**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. (Only valid with C parser).

**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

**skip_blank_lines**: boolean, default True

If True, skip over blank lines rather than interpreting as NaN values

**Returns**
result: DataFrame or TextParser

### 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.

**Parameters**

**filepath_or_buffer**: 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.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 (default=’infer’).

**widths**: list of ints. optional

A list of field widths which can be used instead of ‘colspecs’ if the intervals are contiguous.

**lineterminator**: string (length 1), default None

Character to break file into lines. Only valid with C parser

**quotechar**: string (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 None

Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3). Default (None) results in QUOTE_MINIMAL behavior.
**skipinitialspace**: boolean, default False

Skip spaces after delimiter

**escapechar**: string (length 1), default None

One-character string used to escape delimiter when quoting is QUOTE_NONE.

**dtype**: Type name or dict of column -> type

Data type for data or columns. E.g. `{‘a’: np.float64, ‘b’: np.int32}` (Unsupported with engine=’python’)

**compression**: {'gzip', ‘bz2’, ‘infer’, None}, default ‘infer’

For on-the-fly decompression of on-disk data. If ‘infer’, then use gzip or bz2 if filepath_or_buffer is a string ending in ‘.gz’ or ‘.bz2’, respectively, and no decompression otherwise. Set to None for no decompression.

**dialect**: string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char. See csv.Dialect documentation for more details.

**header**: int, list of ints

Row number(s) to use as the column names, and the start of the data. Defaults 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 are 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.

**skiprows**: list-like or integer

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file

**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)

**names**: array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

**prefix**: string, default None

Prefix to add to column numbers when no header, e.g ‘X’ for X0, X1, ...

**na_values**: list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

**true_values**: list

Values to consider as True

**false_values**: list

Values to consider as False
**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

**parse_dates** : boolean, list of ints or names, list of lists, or dict

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. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo' A fast-path exists for iso8601-formatted dates.

**keep_date_col** : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

**date_parser** : function

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

**thousands** : str, default None

Thousands separator

**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.

**decimal** : str, default ‘.’

Character to recognize as decimal point. E.g. use ‘,’ for European data

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**iterator** : boolean, default False

Return TextFileReader object

**chunksize** : int, default None

Return TextFileReader object for iteration

**skipfooter** : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine=’c’)

**converters** : dict, default None
Dict of functions for converting values in certain columns. Keys can either be integers or column labels

verbose : boolean, default False
Indicate number of NA values placed in non-numeric columns

delimiter : string, default None
Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None
Encoding to use for UTF when reading/writing (ex. ‘utf-8’). List of Python standard encodings

squeeze : boolean, default False
If the parsed data only contains one column then return a Series

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

usecols : array-like
Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols : boolean, default True
Duplicate columns will be specified as ‘X.0’...'X.N’, rather than ‘X'...'X'

tupleize_cols : boolean, default False
Leave a list of tuples on columns as is (default is to convert to a Multi Index 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. (Only valid with C parser)

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. (Only valid with C parser).

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

skip_blank_lines : boolean, default True
If True, skip over blank lines rather than interpreting as NaN values

Returns result : DataFrame or TextParser
Also, ‘delimiter’ is used to specify the filler character of the fields if it is not spaces (e.g., ‘~’).
33.1.3 Clipboard

```python
read_clipboard(**kwargs)  # Read text from clipboard and pass to read_table.
```

**pandas.read_clipboard**

```python
pandas.read_clipboard(**kwargs)
```

Read text from clipboard and pass to `read_table`. See `read_table` for the full argument list.

If unspecified, `sep` defaults to ‘s+’

Returns parsed : DataFrame

33.1.4 Excel

```python
read_excel(io[, sheetname])
```

Read an Excel table into a pandas DataFrame

```python
ExcelFile.parse([sheetname, header, ...])
```

Read an Excel table into DataFrame

**pandas.read_excel**

```python
pandas.read_excel(io, sheetname=0, **kwds)
```

Read an Excel table into a pandas DataFrame

**Parameters**

- `io` : string, file-like object, 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

- `sheetname` : string, int, mixed list of strings/ints, 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.

  str|int -> DataFrame is returned. list|None -> Dict of DataFrames is returned, with keys representing sheets.

  Available Cases

  • Defaults to 0 -> 1st sheet as a DataFrame
  
  • 1 -> 2nd sheet as a DataFrame
  
  • “Sheet1” -> 1st sheet as a DataFrame
  
  • [0,"Sheet5"] -> 1st, 2nd & 5th sheet as a dictionary of DataFrames
  
  • None -> All sheets as a dictionary of DataFrames

- `header` : int, default 0
  
  Row to use for the column labels of the parsed DataFrame

- `skiprows` : list-like
  
  Rows to skip at the beginning (0-indexed)
skip_footer : int, default 0
    Rows at the end to skip (0-indexed)

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.

index_col : int, default None
    Column to use as the row labels of the DataFrame. Pass None if there is no such column

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 column names and column ranges (e.g. “A:E” or “A,C,E:F”)

na_values : list-like, default None
    List of additional strings to recognize as NA/NaN

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

effect: string, default None
    If io is not a buffer or path, this must be set to identify io. Acceptable values are None or xlrd

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

has_index_names : boolean, default False
    True if the cols defined in index_col have an index name and are not in the header. Index name will be placed on a separate line below the header.

Returns parsed : DataFrame or Dict of DataFrames
    DataFrame from the passed in Excel file. See notes in sheetname argument for more information on when a Dict of Dataframes is returned.

pandas.ExcelFile.parse

ExcelFile.parse(sheetname=0, header=0, skiprows=None, skip_footer=0, index_col=None, parse_cols=None, parse_dates=False, date_parser=None, na_values=None, thousands=None, chunksize=None, convert_float=True, has_index_names=False, converters=None, **kwds)
    Read an Excel table into DataFrame
Parameters **sheetname**: 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.

str|int -> DataFrame is returned. list|None -> Dict of DataFrames is returned, with keys representing sheets.

Available Cases

- Defaults to 0 -> 1st sheet as a DataFrame
- 1 -> 2nd sheet as a DataFrame
- “Sheet1” -> 1st sheet as a DataFrame
- [0,1,”Sheet5”] -> 1st, 2nd & 5th sheet as a dictionary of DataFrames
- None -> All sheets as a dictionary of DataFrames

**header**: int, default 0

Row to use for the column labels of the parsed DataFrame

**skiprows**: list-like

Rows to skip at the beginning (0-indexed)

**skip_footer**: int, default 0

Rows at the end to skip (0-indexed)

**converters**: dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**index_col**: int, default None

Column to use as the row labels of the DataFrame. Pass None if there is no such column

**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 int then indicates list of column numbers to be parsed
- If string then indicates comma separated list of column names and column ranges (e.g. “A:E” or “A,C,E:F”)

**parse_dates**: boolean, default False

Parse date Excel values,

**date_parser**: function default None

Date parsing function

**na_values**: list-like, default None

List of additional strings to recognize as NA/NaN

**thousands**: str, default None

Thousands separator
chunksize : int, default None
Size of file chunk to read for lazy evaluation.

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.

has_index_names : boolean, default False
True if the cols defined in index_col have an index name and are not in the header

verbose : boolean, default False
Set to True to print a single statement when reading each excel sheet.

Returns parsed : DataFrame or Dict of DataFrames
DataFrame from the passed in Excel file. See notes in sheetname argument for more information on when a Dict of Dataframes is returned.

33.1.5 JSON

read_json([path_or_buf, orient, typ, dtype, ...])  Convert a JSON string to pandas object

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)
Convert a JSON string to pandas object

Parameters filepath_or_buffer : a valid JSON string or file-like
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
• Series
  – default is ‘index’
  – allowed values are: {‘split’,’records’,’index’}
  – The Series index must be unique for orient ‘index’.
• DataFrame
  – default is ‘columns’
  – allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
  – The DataFrame index must be unique for orient ‘index’ and ‘columns’.
  – The DataFrame columns must be unique for orient ‘index’, ‘columns’, and ‘records’.
• 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

**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.

**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.

**Returns**  **result**: Series or DataFrame

---

**json_normalize**(data[, record_path, meta, ...])  “Normalize” semi-structured JSON data into a flat table

**pandas.io.json.json_normalize**

**pandas.io.json.json_normalize**(data, record_path=None, meta=None, meta_prefix=None, record_prefix=None)

“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)
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

Returns frame : DataFrame

Examples

```python
>>> data = [{
    'state': 'Florida',
    'shortname': 'FL',
    'info': {
        'governor': 'Rick Scott'
    },
    'counties': [{
        'name': 'Dade',
        'population': 12345,
        'info': {
            'governor': 'Rick Scott'
        }
    }, {
        'name': 'Broward',
        'population': 40000,
        'info': {
            'governor': 'Rick Scott'
        }
    }, {
        'name': 'Palm Beach',
        'population': 60000
    }],
    'state': 'Ohio',
    'shortname': 'OH',
    'info': {
        'governor': 'John Kasich'
    },
    'counties': [{
        'name': 'Summit',
        'population': 1234,
        'info': {
            'governor': 'John Kasich'
        }
    }, {
        'name': 'Cuyahoga',
        'population': 1337
    }]
}]
```

```python
>>> from pandas.io.json import json_normalize
>>> 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
```

33.1.6 HTML

**read_html**(io[, match, flavor, header, ...]) Read HTML tables into a list of DataFrame objects.

**pandas.read_html**

```
pandas.read_html  
```

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

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’.

infer_types : None, optional

This has no effect since 0.15.0. It is here for backwards compatibility.

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.

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).

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.

33.1.7 HDFStore: PyTables (HDF5)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_hdf</td>
<td>read from the store, close it if we opened it</td>
</tr>
<tr>
<td>HDFStore.put</td>
<td>Store object in HDFStore</td>
</tr>
<tr>
<td>HDFStore.append</td>
<td>Append to Table in file.</td>
</tr>
<tr>
<td>HDFStore.get</td>
<td>Retrieve pandas object stored in file</td>
</tr>
<tr>
<td>HDFStore.select</td>
<td>Retrieve pandas object stored in file, optionally based on where criteria</td>
</tr>
</tbody>
</table>

**pandas.read_hdf**

pandas.read_hdf(path_or_buf, key, **kwargs)

read from the store, close it if we opened it

Retrieve pandas object stored in file, optionally based on where criteria

**Parameters**

- path_or_buf : path (string), or buffer to read from
- key : group identifier in the store
- 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

pandas.HDFStore.put

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.
encoding : default None, provide an encoding for strings
dropna : boolean, default True, do not write an ALL nan row to
    the store settable by the option ‘io.hdf.dropna_table’

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
    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 True, append the input data to the
    existing
data_columns : list of columns to create as data columns, or True to
    use all columns
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 True, do not write an ALL nan row to
  the store settable by the option `io.hdf.dropna_table`

Notes
—

Does *not* check if data being appended overlaps with existing
data in the table, so be careful

pandas.HDFStore.get

HDFStore.get(key)
  Retrieve pandas object stored in file

Parameters
  key : object

Returns
  obj : type of object stored in file

pandas.HDFStore.select

HDFStore.select(key, where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, auto_close=False, **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 : n rows to include in iteration, return an iterator
    auto_close : boolean, should automatically close the store when
     finished, default is False

Returns
  The selected object

Continued on next page
### 33.1.8 SQL

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.read_sql_table</td>
<td>Read SQL database table into a DataFrame.</td>
</tr>
<tr>
<td>pandas.read_sql_query</td>
<td>Read SQL query into a DataFrame.</td>
</tr>
<tr>
<td>pandas.read_sql</td>
<td>Read SQL query or database table into a DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.read_sql_table**

```
pandas.read_sql_table(table_name, con[, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None])
```

Read SQL database table into a DataFrame.

Given a table name and an SQLAlchemy engine, returns a DataFrame. This function does not support DBAPI connections.

**Parameters**

- `table_name` : string
  Name of SQL table in database
- `con` : SQLAlchemy engine
  Sqlite DBAPI connection mode not supported
- `schema` : string, default None
  Name of SQL schema in database to query (if database flavor supports this). If None, use default schema (default).
- `index_col` : string, optional
  Column to set as index
- `coerce_float` : boolean, default True
  Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.
- `parse_dates` : list or dict
  - 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
  List of column names to select from sql 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_query Read SQL query into a DataFrame.

Notes

Any datetime values with time zone information will be converted to UTC

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 to be executed

    con : SQLAlchemy engine 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, optional
        Column name to use as index for the returned DataFrame object.

    coerce_float : boolean, default True
        Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

    params : list, tuple or dict, optional
        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
        • 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

Notes

Any datetime values with time zone information parsed via the parse_dates parameter will be converted to UTC

pandas.read_sql

pandas.read_sql(sql, con, 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
SQL query to be executed or database table name.

con : SQLAlchemy engine 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, optional
column name to use as index for the returned DataFrame object.

columns : list
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.

### 33.1.9 Google BigQuery

| read_gbq(query[, project_id, index_col, ...]) | Load data from Google BigQuery. |
| to_gbq(dataframe, destination_table[, ...]) | Write a DataFrame to a Google BigQuery table. |

**pandas.io.gbq.read_gbq**

Load data from Google BigQuery.

**THIS IS AN EXPERIMENTAL LIBRARY**

The main method a user calls to execute a Query in Google BigQuery and read results into a pandas DataFrame using the v2 Google API client for Python. Documentation for the API is available at https://developers.google.com/api-client-library/python/. Authentication to the Google BigQuery service is via OAuth 2.0 using the product name 'pandas GBQ'.

**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.

**Returns**

- `df` : DataFrame
  DataFrame representing results of query
pandas.io.gbq.to_gbq

```
pandas.io.gbq.to_gbq(dataframe, destination_table, project_id=None, chunksize=10000, verbose=True, reauth=False)
```

Write a DataFrame to a Google BigQuery table.

**THIS IS AN EXPERIMENTAL LIBRARY**

If the table exists, the dataframe will be written to the table using the defined table schema and column types. For simplicity, this method uses the Google BigQuery streaming API. The to_gbq method chunks data into a default chunk size of 10,000. Failures return the complete error response which can be quite long depending on the size of the insert. There are several important limitations of the Google streaming API which are detailed at: https://developers.google.com/bigquery/streaming-data-into-bigquery.

**Parameters**

- **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.

### 33.1.10 STATA

```
pandas.read_stata(filepath_or_buffer[, ...]) : Read Stata file into DataFrame
```

**pandas.read_stata**

```
pandas.read_stata(filepath_or_buffer, convert_dates=True, convert_categoricals=True, encoding=None, index=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

### 33.1. Input/Output
encoding : string, None or encoding

Encoding used to parse the files. Note that Stata doesn’t support unicode. None defaults to iso-8859-1.

index : identifier of index column

identifier of column that should be used as index of the DataFrame

convert_missing : boolean, defaults to False

Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nans. 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: >> df = pandas.read_stata('filename.dta')

Read a Stata dta file in 10,000 line chunks: >> itr = pandas.read_stata('filename.dta’, chunksize=10000) >> for chunk in itr: >> do_something(chunk)

StataReader.data(**kwargs)  DEPRECATED: Reads observations from Stata file, converting them into a dataframe
StataReader.data_label()  Returns data label of Stata file
StataReader.value_labels()  Returns a dict, associating each variable name a dict, associating
StataReader.variable_labels()  Returns variable labels as a dict, associating each variable name
StataWriter.write_file()  

pandas.io.stata.StataReader.data

StataReader.data(**kwargs)

DEPRECATED: Reads observations from Stata file, converting them into a dataframe

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** : identifier of index column

- identifier of column that should be used as index of the DataFrame

**convert_missing** : boolean, defaults to False

- Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nans. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissing-Value 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

---

**pandas.io.stata.StataReader.data_label**

StataReader.

**data_label**

- Returns data label of Stata file

**pandas.io.stata.StataReader.value_labels**

StataReader.

**value_labels**

- Returns a dict, associating each variable name a dict, associating each value its corresponding label

**pandas.io.stata.StataReader.variable_labels**

StataReader.

**variable_labels**

- Returns variable labels as a dict, associating each variable name with corresponding label

**pandas.io.stata.StataWriter.write_file**

StataWriter.

**write_file**
33.2 General functions

33.2.1 Data manipulations
pandas.melt

\[
\text{pandas.melt(frame[, id_vars, value_vars, var_name, ...])}
\]

“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 \((\text{id_vars})\), while all other columns, considered measured variables \((\text{value_vars})\), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

**Parameters**

- **frame**: DataFrame
  
  Column(s) to use as identifier variables.

- **id_vars**: tuple, list, or ndarray, optional
  
  Column(s) to unpivot. If not specified, uses all columns that are not set as \text{id_vars}.

- **value_vars**: tuple, list, or ndarray, optional
  
  Name to use for the ‘variable’ column. If None it uses \text{frame.columns.name} or ‘variable’.

- **var_name**: scalar
  
  Name to use for the ‘value’ column.

- **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:**

\text{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
```
```
>>> 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:

>>> pd.melt(df, id_vars=['A'], value_vars=['B'],
          var_name='myVarname', value_name='myValname')
   myVarname  myValname
0   a      B     1
1   b      B     3
2   c      B     5

If you have multi-index columns:

>>> df.columns = [list('ABC'), list('DEF')]

>>> df
   A  B  C  D  E  F
0  a  1  2  0  1  2
1  b  3  4  3  4  5
2  c  5  6  5  6

>>> pd.melt(df, col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0   a      B     1
1   b      B     3
2   c      B     5

>>> pd.melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
   (A, D) variable_0 variable_1  value
0   a      B      E     1
1   b      B      E     3
2   c      B      E     5
```

`pandas.pivot`

`pandas.pivot(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

**Notes**

Obviously, all 3 of the input arguments must have the same length

**pandas.pivot_table**

```python
pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True)
```

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

**Parameters**

- **data** : DataFrame
- **values** : column to aggregate, optional
- **index** : 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, default numpy.mean, or list of functions
  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

**Returns** DataFrame

**Examples**

```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
```
```python
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...     columns=['C'], aggfunc=np.sum)

>>> table
         small  large
    foo one  1     4
          two  6    NaN
    bar one  5     4
          two  6     7
```

**pandas.crosstab**

`pandas.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, dropna=True)`

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
- `aggfunc` : function, optional
  If no values array is passed, computes a frequency table
- `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)
- `dropna` : boolean, default True
  Do not include columns whose entries are all NaN

**Returns**

- `crosstab` : DataFrame

**Notes**

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified
Examples

```python
>>> a
array([foo, foo, foo, foo, bar, bar,
    bar, bar, foo, foo, foo], dtype=object)
>>> b
array([one, one, one, two, one, one,
    one, two, two, two, one], dtype=object)
>>> c
array([dull, dull, shiny, dull, dull, shiny,
    shiny, dull, shiny, shiny, shiny], dtype=object)

```crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th></th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>one</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>foo</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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 or sequence of scalars
  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
  The precision at which to store and display the bins labels
- **include_lowest**: bool
  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.
**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
good, medium, bad)
>>> pd.cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, labels=False)
array([1, 1, 1, 1, 1, 1], dtype=int64)
```

**pandas.qcut**

`pandas.qcut(x, q, labels=None, retbins=False, precision=3)`

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 or not. Can be useful if bins is given as a scalar.
- **precision**: int
  The precision at which to store and display the bins labels

**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], [0, 1], (1, 2), (2, 3), (3, 4)]
Categories (4, object): 
([0, 1] < (1, 2) < (2, 3) < (3, 4])
>>> pd.qcut(range(5), 3, labels=['good','medium','bad'])
good, good, medium, bad, bad
Categories (3, object): 
(good < medium < bad)
>>> pd.qcut(range(5), 4, labels=False)
array([0, 0, 1, 2, 3], dtype=int64)
```

pandas.merge

```python
pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True)
```

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 (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)
- **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

**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

**Returns** **merged** : DataFrame

The output type will be the same as ‘left’, if it is a subclass of DataFrame.

**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

>>> merge(A, 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.concat**

**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, 1, ...}, 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
**verify_integrity**: boolean, default False

Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.

**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.

**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 the index values on the other axes are still respected in the join.

**copy**: boolean, default True

If False, do not copy data unnecessarily.

**Returns**

**concatenated**: type of objects

**Notes**

The keys, levels, and names arguments are all optional.

**pandas.get_dummies**

`pandas.get_dummies(data, prefix=None, prefix_sep='__', dummy_na=False, columns=None, sparse=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 returned DataFrame should be sparse or not.

New in version 0.16.1.

Returns dummies : DataFrame

Examples

```python
>>> import pandas as pd
>>> s = pd.Series(list('abca'))

>>> get_dummies(s)
    a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0

>>> s1 = ['a', 'b', np.nan]

>>> get_dummies(s1)
    a  b
0  1  0
1  0  1
2  0  0

>>> get_dummies(s1, dummy_na=True)
    a  b  NaN
0  1  0  0
1  0  1  0
2  0  0  1

>>> df = DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'], 'C': [1, 2, 3]})

>>> 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

See also Series.str.get_dummies.

pandas.factorize

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
order : deprecated

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

33.2.2 Top-level missing data

<table>
<thead>
<tr>
<th>isnull(obj)</th>
<th>Detect missing values (NaN in numeric arrays, None/NaN in object arrays)</th>
</tr>
</thead>
<tbody>
<tr>
<td>notnull(obj)</td>
<td>Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.</td>
</tr>
</tbody>
</table>

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 isnulled : 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.notnull boolean inverse of pandas.isnull

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 isnulled : 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.isnull boolean inverse of pandas.notnull
33.2.3 Top-level dealing with datetimelike
to_datetime(arg[, errors, dayfirst, utc, ...])
Convert argument to datetime.

to_timedelta(arg[, unit, box, coerce])
Convert argument to timedelta

date_range([start, end, periods, freq, tz, ...])
Return a fixed frequency datetime index, with day (calendar) as the default

date_range((start, end, periods, freq, tz, ...))
Return a fixed frequency datetime index, with day (calendar) as the default

bdate_range([start, end, periods, freq, tz, ...])
Return a fixed frequency datetime index, with business day as the default

period_range([start, end, periods, freq, name])
Return a fixed frequency datetime index, with day (calendar) as the default

timedelta_range([start, end, periods, freq, ...])
Return a fixed frequency timedelta index, with day as the default

infer_freq(index[, warn])
Infer the most likely frequency given the input index.

pandas.to_datetime

pandas.to_datetime (arg, errors=’ignore’, dayfirst=False, utc=None, box=True, format=None, exact=True, coerce=False, unit=’ns’, infer_datetime_format=False)
Convert argument to datetime.

Parameters arg : string, datetime, array of strings (with possible NAs)

errors : {‘ignore’, ‘raise’}, default ‘ignore’
Errors are ignored by default (values left untouched)

dayfirst : boolean, default False
If True parses dates with the day first, eg 20/01/2005 Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug).

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.

coerce : force errors to NaT (False by default)
Timestamps outside the interval between Timestamp.min and Timestamp.max (approximately 1677-09-22 to 2262-04-11) will be also forced to NaT.

unit : unit of the arg (D,s,ms,us,ns) denote the unit in epoch
(e.g. a unix timestamp), which is an integer/float number

infer_datetime_format : boolean, default False
If no format is given, try to infer the format based on the first datetime string. Provides a large speed-up in many cases.

Returns ret : datetime if parsing succeeded.

Return type depends on input:
• list-like: DatetimeIndex

33.2. General functions
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).

Examples

Take separate series and convert to datetime

```python
>>> import pandas as pd

>>> i = pd.date_range('20000101', periods=100)

>>> df = pd.DataFrame(dict(year=i.year, month=i.month, day=i.day))

>>> pd.to_datetime(df.year*10000 + df.month*100 + df.day, format='%Y%m%d')
0 2000-01-01
1 2000-01-02
...
98 2000-04-08
99 2000-04-09
Length: 100, dtype: datetime64[ns]
```

Or from strings

```python
>>> df = df.astype(str)

>>> pd.to_datetime(df.day + df.month + df.year, format='%d%m%Y')
0 2000-01-01
1 2000-01-02
...
98 2000-04-08
99 2000-04-09
Length: 100, dtype: datetime64[ns]
```

Date that does not meet timestamp limitations:

```python
>>> pd.to_datetime('13000101', format='%Y%m%d')
datetime.datetime(1300, 1, 1, 0, 0)

>>> pd.to_datetime('13000101', format='%Y%m%d', coerce=True)
NaT
```

### pandas.to_timedelta

`pandas.to_timedelta(arg, unit='ns', box=True, coerce=False)`

Convert argument to timedelta

**Parameters**

- `arg`: string, timedelta, array of strings (with possible NAs)
- `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]

- `coerce`: force errors to NaT (False by default)

**Returns**

- `ret`: timedelta64/arrays of timedelta64 if parsing succeeded
pandas: powerful Python data analysis toolkit, Release 0.16.2

**pandas.date_range**

```
pandas.date_range(start=None, end=None, periods=None, freq='D', tz=None, normalize=False, name=None, closed=None)
```

Return a fixed frequency datetime index, 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 or None, default None
  
  If None, must specify start and end

- **freq** : string or DateOffset, default ‘D’ (calendar daily)
  
  Frequency strings can have multiples, e.g. ‘5H’

- **tz** : string or 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** : str, default None
  
  Name of the resulting index

- **closed** : string or None, default None
  
  Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

**Returns**

```
rng : DatetimeIndex
```

**Notes**

2 of start, end, or periods must be specified

**pandas.bdate_range**

```
pandas.bdate_range(start=None, end=None, periods=None, freq='B', tz=None, normalize=True, name=None, closed=None)
```

Return a fixed frequency datetime index, 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 or None, default None
  
  If None, must specify start and end
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 : str, default None

Name for the resulting index

closed : string or None, default None

Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

Returns rng : DatetimeIndex

Notes

2 of start, end, or periods must be specified

pandas.period_range

pandas.period_range (start=None, end=None, periods=None, freq='D', name=None)

Return a fixed frequency datetime index, with day (calendar) as the default frequency

Parameters start : starting value, period-like, optional

date or timedelta-like, default None

end : ending value, period-like, optional

datetime-like, default None

periods : int, default None

Number of periods in the index

freq : str/DateOffset, default ‘D’

Frequency alias

name : str, default None

Name for the resulting PeriodIndex

Returns prng : PeriodIndex

pandas.timedelta_range

pandas.timedelta_range (start=None, end=None, periods=None, freq='D', name=None, closed=None)

Return a fixed frequency timedelta index, with day as the default frequency

Parameters start : string or timedelta-like, default None

Left bound for generating dates

date or timedelta-like, default None

date or datetime-like, default None

Right bound for generating dates
**periods**: integer or None, default None

If None, must specify start and end

**freq**: string or DateOffset, default ‘D’ (calendar daily)

Frequency strings can have multiples, e.g. ‘5H’

**name**: str, default None

Name of the resulting index

**closed**: string or None, default None

Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None)

**Returns**

**rng**: TimedeltaIndex

**Notes**

2 of start, end, or periods must be specified

---

**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.

---

### 33.2.4 Top-level evaluation

**eval**(expr[, parser, engine, truediv, ...])

Evaluate a Python expression as a string using various backends.

---

**pandas.eval**

**pandas.eval**(expr, parser='pandas', engine='numexpr', truediv=True, local_dict=None, global_dict=None, resolvers=(), level=0, target=None)

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* documentation for more details.

**engine** : string, default ‘numexpr’, {'python’, ‘numexpr’}

The engine used to evaluate the expression. Supported engines are

- ‘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** : a target object for assignment, optional, default is None

essentially this is a passed in resolver

**Returns**  
ndarray, numeric scalar, DataFrame, Series

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.
### 33.2.5 Standard moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rolling_count</strong></td>
<td>Rolling count of number of non-NaN observations inside provided window.</td>
</tr>
<tr>
<td><strong>rolling_sum</strong></td>
<td>Moving sum.</td>
</tr>
<tr>
<td><strong>rolling_mean</strong></td>
<td>Moving mean.</td>
</tr>
<tr>
<td><strong>rolling_median</strong></td>
<td>O(N log(window)) implementation using skip list.</td>
</tr>
<tr>
<td><strong>rolling_var</strong></td>
<td>Numerically stable implementation using Welford’s method.</td>
</tr>
<tr>
<td><strong>rolling_std</strong></td>
<td>Moving standard deviation.</td>
</tr>
<tr>
<td><strong>rolling_min</strong></td>
<td>Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs.</td>
</tr>
<tr>
<td><strong>rolling_max</strong></td>
<td>Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs.</td>
</tr>
<tr>
<td><strong>rolling_corr</strong></td>
<td>Moving sample correlation.</td>
</tr>
<tr>
<td><strong>rolling_corr_pairwise</strong></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><strong>rolling_cov</strong></td>
<td>Unbiased moving covariance.</td>
</tr>
<tr>
<td><strong>rolling_skew</strong></td>
<td>Unbiased moving skewness.</td>
</tr>
<tr>
<td><strong>rolling_kurt</strong></td>
<td>Unbiased moving kurtosis.</td>
</tr>
<tr>
<td><strong>rolling_apply</strong></td>
<td>Generic moving function application.</td>
</tr>
<tr>
<td><strong>rolling_quantile</strong></td>
<td>Moving quantile.</td>
</tr>
<tr>
<td><strong>rolling_window</strong></td>
<td>Applies a moving window of type window_type and size window on the data.</td>
</tr>
</tbody>
</table>

**pandas.rolling_count**

Rolling count of number of non-NaN observations inside provided window.

**Parameters**

- **arg** : DataFrame or numpy ndarray-like
  - **window** : int
    - Size of the moving window. This is the number of observations used for calculating the statistic.
  - **freq** : string or DateOffset object, optional (default None)
    - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
  - **center** : boolean, default False
    - Whether the label should correspond with center of window
  - **how** : string, default 'mean'
    - Method for down- or re-sampling

**Returns**

- **rolling_count** : type of caller

**Notes**

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).
pandas.rolling_sum

Moving sum.

Parameters
- **arg**: Series, DataFrame
  - **window**: int
    Size of the moving window. This is the number of observations used for calculating the statistic.
  - **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)
    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.
  - **how**: string, default ‘None’
    Method for down- or re-sampling

Returns **y**: type of input argument

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).

pandas.rolling_mean

Moving mean.

Parameters
- **arg**: Series, DataFrame
  - **window**: int
    Size of the moving window. This is the number of observations used for calculating the statistic.
  - **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)
    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.

how : string, default ‘None’

Method for down- or re-sampling

Returns y : type of input argument

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).

pandas.rolling_median

pandas.rolling_median(arg, window, min_periods=None, freq=None, center=False, how='median', **kwargs)

O(N log(window)) implementation using skip list

Moving median.

Parameters arg : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

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)

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.

how : string, default ‘median’

Method for down- or re-sampling

Returns y : type of input argument

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).
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.rolling_var

pandas.rolling_var(arg, window, min_periods=None, freq=None, center=False, how=None, **kwargs)

Numerically stable implementation using Welford’s method.

Moving variance.

Parameters arg : Series, DataFrame

window : int
   Size of the moving window. This is the number of observations used for calculating the statistic.

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)
   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.

how : string, default ‘None’
   Method for down- or re-sampling

ddf : int, default 1
   Delta Degrees of Freedom. The divisor used in calculations is N - ddf, where N represents the number of elements.

Returns y : type of input argument

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).

pandas.rolling_std

pandas.rolling_std(arg, window, min_periods=None, freq=None, center=False, how=None, **kwargs)

Moving standard deviation.

Parameters arg : Series, DataFrame

window : int
   Size of the moving window. This is the number of observations used for calculating the statistic.

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)

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.

**how** : string, default ‘None’

Method for down- or re-sampling

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.

**Returns** y : type of input argument

**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`).

---

**pandas.rolling_min**

```python
pandas.rolling_min(arg, window, min_periods=None, freq=None, center=False, how='min', **kwargs)
```

Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs. Moving minimum.

**Parameters**

**arg** : Series, DataFrame

**window** : int

Size of the moving window. This is the number of observations used for calculating the statistic.

**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)

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.

**how** : string, default ‘min’

Method for down- or re-sampling

**Returns** y : type of input argument
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`).

**pandas.rolling_max**

```
pandas.rolling_max(arg, window, min_periods=None, freq=None, center=False, how='max', **kwargs)
```

Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs. Moving maximum.

Parameters

- **arg**: Series, DataFrame
- **window**: int
  - Size of the moving window. This is the number of observations used for calculating the statistic.
- **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)
  - 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.
- **how**: string, default ‘max’
  - Method for down- or re-sampling

Returns

- **y**: type of input argument

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`).

**pandas.rolling_corr**

```
pandas.rolling_corr(arg1, arg2=None, window=None, min_periods=None, freq=None, center=False, pairwise=None, how=None)
```

Moving sample correlation.

Parameters

- **arg1**: Series, DataFrame, or ndarray
- **arg2**: Series, DataFrame, or ndarray, optional
  - if not supplied then will default to `arg1` and produce pairwise output
**window**: int

Size of the moving window. This is the number of observations used for calculating the statistic.

**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)

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.

**how**: string, default ‘None’

Method for down- or re-sampling

**pairwise**: bool, default False

If False then only matching columns between arg1 and arg2 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 Panel in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**Returns y**: type depends on inputs

- DataFrame / DataFrame -> DataFrame (matches on columns) or Panel (pairwise)
- DataFrame / Series -> Computes result for each column
- Series / Series -> Series

**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`).

**pandas.rolling_corr_pairwise**

`pandas.rolling_corr_pairwise(df1, df2=None, window=None, min_periods=None, freq=None, center=False)`

Deprecated. Use `rolling_corr(..., pairwise=True)` instead.

Pairwise moving sample correlation

**Parameters**

- **df1**: DataFrame
- **df2**: DataFrame
- **window**: int
  
  Size of the moving window. This is the number of observations used for calculating the statistic.
- **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)

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.

**how** : string, default ‘None’

Method for down- or re-sampling

**Returns** y : Panel whose items are df1.index values

**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`).

**pandas.rolling_cov**

`pandas.rolling_cov`  

`pandas.rolling_cov(arg1, arg2=None, window=None, min_periods=None, freq=None, center=False, pairwise=None, how=None, ddof=1)`  

Unbiased moving covariance.

**Parameters arg1** : Series, DataFrame, or ndarray

**arg2** : Series, DataFrame, or ndarray, optional

if not supplied then will default to arg1 and produce pairwise output

**window** : int

Size of the moving window. This is the number of observations used for calculating the statistic.

**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)

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.

**how** : string, default ‘None’

Method for down- or re-sampling

**pairwise** : bool, default False
If False then only matching columns between arg1 and arg2 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 Panel 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**

\(y\) : type depends on inputs

- DataFrame / DataFrame -> DataFrame (matches on columns) or Panel (pairwise)
- DataFrame / Series -> Computes result for each column
- Series / Series -> Series

**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`).

**pandas.rolling_skew**

pandas.rolling_skew(arg, window, min_periods=None, freq=None, center=False, how=None, **kwargs)

Unbiased moving skewness.

**Parameters**

- **arg**: Series, DataFrame
  - **window**: int
    - Size of the moving window. This is the number of observations used for calculating the statistic.
  - **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)
    - 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.
  - **how**: string, default ‘None’
    - Method for down- or re-sampling

**Returns**

\(y\) : type of input argument

**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`).

### pandas.rolling_kurt

```python
def pandas.rolling_kurt(arg, window, min_periods=None, freq=None, center=False, how=None, **kwargs):
```

Unbiased moving kurtosis.

**Parameters**

- **arg**: Series, DataFrame
- **window**: int
  - Size of the moving window. This is the number of observations used for calculating the statistic.
- **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)
  - 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.
- **how**: string, default ‘None’
  - Method for down- or re-sampling

**Returns**

- **y**: type of input argument

**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`).

### pandas.rolling_apply

```python
def pandas.rolling_apply(arg, window, func, min_periods=None, freq=None, center=False, args=(), kwargs={}):
```

Generic moving function application.

**Parameters**

- **arg**: Series, DataFrame
- **window**: int
  - Size of the moving window. This is the number of observations used for calculating the statistic.
- **func**: function
  - Must produce a single value from an ndarray input
**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)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**center**: boolean, default False

Whether the label should correspond with center of window

**args**: tuple

Passed on to func

**kwargs**: dict

Passed on to func

**Returns y**: type of input argument

**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`).

**pandas.rolling_quantile**

`pandas.rolling_quantile(arg, window, quantile, min_periods=None, freq=None, center=False)`

Moving quantile.

**Parameters arg**: Series, DataFrame

**window**: int

Size of the moving window. This is the number of observations used for calculating the statistic.

**quantile**: float

`0 <= quantile <= 1`

**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)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**center**: boolean, default False

Whether the label should correspond with center of window

**Returns y**: type of input argument
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`).

**pandas.rolling_window**

Applies a moving window of type `window_type` and size `window` on the data.

**Parameters**

- **arg** : Series, DataFrame
  - `window` : int or ndarray
    Weighting window specification. If the window is an integer, then it is treated as the window length and `win_type` is required
  - `win_type` : str, default None
    Window type (see Notes)
  - `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)
    Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
  - `center` : boolean, default False
    Whether the label should correspond with center of window
  - `mean` : boolean, default True
    If True computes weighted mean, else weighted sum
  - `axis` : {0, 1}, default 0
  - `how` : string, default ‘mean’
    Method for down- or re-sampling

**Returns**

- **y** : type of input argument

**Notes**

The recognized window types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
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`).

### 33.2.6 Standard expanding window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>expanding_count</strong></td>
<td>Expanding count of number of non-NaN observations.</td>
</tr>
<tr>
<td><strong>expanding_sum</strong></td>
<td>Expanding sum.</td>
</tr>
<tr>
<td><strong>expanding_mean</strong></td>
<td>Expanding mean.</td>
</tr>
<tr>
<td><strong>expanding_median</strong></td>
<td>O(N log(window)) implementation using skip list.</td>
</tr>
<tr>
<td><strong>expanding_var</strong></td>
<td>Numerically stable implementation using Welford’s method.</td>
</tr>
<tr>
<td><strong>expanding_std</strong></td>
<td>Expanding standard deviation.</td>
</tr>
<tr>
<td><strong>expanding_min</strong></td>
<td>Moving min of 1d array of <code>dtype=float64</code> along <code>axis=0</code> ignoring <code>NaNs</code>.</td>
</tr>
<tr>
<td><strong>expanding_max</strong></td>
<td>Moving max of 1d array of <code>dtype=float64</code> along <code>axis=0</code> ignoring <code>NaNs</code>.</td>
</tr>
<tr>
<td><strong>expanding_corr</strong></td>
<td>Expanding sample correlation.</td>
</tr>
<tr>
<td><strong>expanding_corr_pairwise</strong></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><strong>expanding_cov</strong></td>
<td>Unbiased expanding covariance.</td>
</tr>
<tr>
<td><strong>expanding_skew</strong></td>
<td>Unbiased expanding skewness.</td>
</tr>
<tr>
<td><strong>expanding_kurt</strong></td>
<td>Unbiased expanding kurtosis.</td>
</tr>
<tr>
<td><strong>expanding_apply</strong></td>
<td>Generic expanding function application.</td>
</tr>
<tr>
<td><strong>expanding_quantile</strong></td>
<td>Expanding quantile.</td>
</tr>
</tbody>
</table>

#### pandas.expanding_count

**pandas.expanding_count** *(arg, freq=None)*  
Expanding count of number of non-NaN observations.

**Parameters**  
arg : DataFrame or numpy ndarray-like

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**Returns**  
expanding_count : type of caller
Notes

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*).

**pandas.expanding_sum**

```python
pandas.expanding_sum(arg, min_periods=1, freq=None, **kwargs)
```

Expanding sum.

- **Parameters**
  - *arg*: Series, DataFrame
  - *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)
    - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

- **Returns**
  - *y*: type of input argument

**pandas.expanding_mean**

```python
pandas.expanding_mean(arg, min_periods=1, freq=None, **kwargs)
```

Expanding mean.

- **Parameters**
  - *arg*: Series, DataFrame
  - *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)
    - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

- **Returns**
  - *y*: type of input argument

**pandas.expanding_median**

```python
pandas.expanding_median(arg, min_periods=1, freq=None, **kwargs)
```

O(N log(window)) implementation using skip list

Expanding median.

- **Parameters**
  - *arg*: Series, DataFrame
  - *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)
    - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**Returns** y : type of input argument

**pandas.expanding_var**

```
pandas.expanding_var(arg, min_periods=1, freq=None, **kwargs)
```

Numerically stable implementation using Welford’s method.

Expanding variance.

**Parameters**

- **arg** : Series, DataFrame
- **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)
  Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
- **ddof** : int, default 1
  Delta Degrees of Freedom. The divisor used in calculations is \( N - \) ddof, where \( N \) represents the number of elements.

**Returns** y : type of input argument

**pandas.expanding_std**

```
pandas.expanding_std(arg, min_periods=1, freq=None, **kwargs)
```

Expanding standard deviation.

**Parameters**

- **arg** : Series, DataFrame
- **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)
  Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
- **ddof** : int, default 1
  Delta Degrees of Freedom. The divisor used in calculations is \( N - \) ddof, where \( N \) represents the number of elements.

**Returns** y : type of input argument

**pandas.expanding_min**

```
pandas.expanding_min(arg, min_periods=1, freq=None, **kwargs)
```

Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs. Expanding minimum.
Parameters arg : Series, DataFrame

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)

Frequency to conform the data to before computing the statistic. Specified as a fre-
quency string or DateOffset object.

Returns y : type of input argument

pandas.expanding_max

pandas.expanding_max(arg, min_periods=1, freq=None, **kwargs)

Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs. Expanding maximum.

Parameters arg : Series, DataFrame

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)

Frequency to conform the data to before computing the statistic. Specified as a fre-
quency string or DateOffset object.

Returns y : type of input argument

pandas.expanding_corr

pandas.expanding_corr(arg1, arg2=None, min_periods=1, freq=None, pairwise=None)

Expanding sample correlation.

Parameters arg1 : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray, optional

if not supplied then will default to arg1 and produce pairwise output

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)

Frequency to conform the data to before computing the statistic. Specified as a fre-
quency string or DateOffset object.

pairwise : bool, default False

If False then only matching columns between arg1 and arg2 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 Panel in the case of DataFrame inputs. In the case of missing elements,
only complete pairwise observations will be used.

Returns y : type depends on inputs
DataFrame / DataFrame -> DataFrame (matches on columns) or Panel (pairwise)
DataFrame / Series -> Computes result for each column
Series / Series -> Series

**pandas.expanding_corr_pairwise**

```
pandas.expanding_corr_pairwise(df1, df2=None, min_periods=1, freq=None)
```

Deprecated. Use `expanding_corr(..., pairwise=True)` instead.

Pairwise expanding sample correlation

**Parameters**
- `df1` : DataFrame
- `df2` : DataFrame
- `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)
  Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**Returns**
- `y` : Panel whose items are `df1.index` values

**pandas.expanding_cov**

```
pandas.expanding_cov(arg1, arg2=None, min_periods=1, freq=None, pairwise=None, ddof=1)
```

Unbiased expanding covariance.

**Parameters**
- `arg1` : Series, DataFrame, or ndarray
- `arg2` : Series, DataFrame, or ndarray, optional
  if not supplied then will default to `arg1` and produce pairwise output
- `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)
  Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
- `pairwise` : bool, default False
  If False then only matching columns between `arg1` and `arg2` 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 Panel 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**
- `y` : type depends on inputs
  DataFrame / DataFrame -> DataFrame (matches on columns) or Panel (pairwise)
  DataFrame / Series -> Computes result for each column
  Series / Series -> Series
pandas: powerful Python data analysis toolkit, Release 0.16.2

**pandas.expanding_skew**

```
pandas.expanding_skew(arg, min_periods=1, freq=None, **kwargs)
```

Unbiased expanding skewness.

**Parameters**
- **arg**: Series, DataFrame
- **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)
  - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**Returns**
- **y**: type of input argument

**pandas.expanding_kurt**

```
pandas.expanding_kurt(arg, min_periods=1, freq=None, **kwargs)
```

Unbiased expanding kurtosis.

**Parameters**
- **arg**: Series, DataFrame
- **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)
  - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

**Returns**
- **y**: type of input argument

**pandas.expanding_apply**

```
pandas.expanding_apply(arg, func, min_periods=1, freq=None, args=(), kwargs={})
```

Generic expanding function application.

**Parameters**
- **arg**: Series, DataFrame
- **func**: function
  - Must produce a single value from an ndarray input
- **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)
  - Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.
- **args**: tuple
  - Passed on to `func`
kwargs : dict
    Passed on to func

Returns y : type of input argument

Notes

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).

pandas.expanding_quantile

pandas.expanding_quantile(arg, quantile, min_periods=1, freq=None)
    Expanding quantile.

Parameters arg : Series, DataFrame
    quantile : float
        0 <= quantile <= 1
    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)
        Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

Returns y : type of input argument

Notes

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).

33.2.7 Exponentially-weighted moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ewma(arg[, com, span, halflife, ...])</code></td>
<td>Exponentially-weighted moving average</td>
</tr>
<tr>
<td><code>ewmstd(arg[, com, span, halflife, ...])</code></td>
<td>Exponentially-weighted moving std</td>
</tr>
<tr>
<td><code>ewmvar(arg[, com, span, halflife, ...])</code></td>
<td>Exponentially-weighted moving variance</td>
</tr>
<tr>
<td><code>ewmcorr(arg1[, arg2, com, span, halflife, ...])</code></td>
<td>Exponentially-weighted moving correlation</td>
</tr>
<tr>
<td><code>ewmcov(arg1[, arg2, com, span, halflife, ...])</code></td>
<td>Exponentially-weighted moving covariance</td>
</tr>
</tbody>
</table>

pandas.ewma

pandas.ewma(arg, com=None, span=None, halflife=None, min_periods=0, freq=None, adjust=True, how=None, ignore_na=False)
    Exponentially-weighted moving average

Parameters arg : Series, DataFrame
**com**: float, optional

Center of mass: $$\alpha = 1/(1 + \text{com})$$.

**span**: float, optional

Specify decay in terms of span, $$\alpha = 2/(\text{span} + 1)$$.

**halflife**: float, optional

Specify decay in terms of halflife, $$\alpha = 1 - \exp(\log(0.5)/\text{halflife})$$.

**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

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).

**how**: string, default ‘mean’

Method for down- or re-sampling.

**ignore_na**: boolean, default False

Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior.

**Returns**

- **y**: type of input argument

**Notes**

Either center of mass or span must be specified.

EWMA is sometimes specified using a “span” parameter $$s$$, we have that the decay parameter $$\alpha$$ is related to the span as $$\alpha = 2/(s + 1) = 1/(1 + c)$$.

where $$c$$ is the center of mass. Given a span, the associated center of mass is $$c = (s - 1)/2$$.

So a “20-day EWMA” would have center 9.5.

**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**: $$(\text{weighted_average}[0] = \text{arg}[0];\text{weighted_average}[i] = (1-\alpha) \ast \text{weighted_average}[i-1] + \alpha \ast \text{arg}[i]).$$

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).
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.ewmstd
pandas.ewmstd(arg, com=None, span=None,
nore_na=False, adjust=True)
Exponentially-weighted moving std

halflife=None,

min_periods=0,

bias=False,

ig-

Parameters arg : Series, DataFrame
com : float. optional
Center of mass: 𝛼 = 1/(1 + 𝑐𝑜𝑚),
span : float, optional
Specify decay in terms of span, 𝛼 = 2/(𝑠𝑝𝑎𝑛 + 1)
halflife : float, optional
Specify decay in terms of halflife, 𝛼 = 1 − 𝑒𝑥𝑝(𝑙𝑜𝑔(0.5)/ℎ𝑎𝑙𝑓 𝑙𝑖𝑓 𝑒)
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
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)
how : string, default ‘mean’
Method for down- or re-sampling
ignore_na : boolean, default False
Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0
behavior
bias : boolean, default False
Use a standard estimation bias correction
Returns y : type of input argument
Notes

Either center of mass or span must be specified
EWMA is sometimes specified using a “span” parameter s, we have that the decay parameter 𝛼 is related to the
span as 𝛼 = 2/(𝑠 + 1) = 1/(1 + 𝑐)
where c is the center of mass. Given a span, the associated center of mass is 𝑐 = (𝑠 − 1)/2
So a “20-day EWMA” would have center 9.5.
When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1),
alpha)**(n-2), ..., 1-alpha, 1.
When adjust is False, weighted averages are calculated recursively as: weighted_average[0]
weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

33.2. General functions

=

(1arg[0];

943


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 \([x, \text{None}, y]\) 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 \([x, \text{None}, y]\) are 1-\alpha and 1 (if adjust is True), and 1-\alpha and alpha (if adjust is False).

**pandas.ewmvar**

```python
pandas.ewmvar(arg, com=None, span=None, halflife=None, min_periods=0, bias=False, freq=None, how=None, ignore_na=False, adjust=True)
```

Exponentially-weighted moving variance

**Parameters**

- **arg** : Series, DataFrame
  - **com** : float, optional
    - Center of mass: \(\alpha = 1/(1 + \text{com})\),
  - **span** : float, optional
    - Specify decay in terms of span, \(\alpha = 2/(\text{span} + 1)\)
  - **halflife** : float, optional
    - Specify decay in terms of halflife, \(\alpha = 1 - \exp(\log(0.5)/\text{halflife})\)
  - **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
    - 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)
  - **how** : string, default ‘mean’
    - Method for down- or re-sampling
  - **ignore_na** : boolean, default False
    - Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior
  - **bias** : boolean, default False
    - Use a standard estimation bias correction

**Returns**

- **y** : type of input argument

**Notes**

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter \(s\), we have that the decay parameter \(\alpha\) is related to the span as \(\alpha = 2/(s + 1) = 1/(1 + c)\)
where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$.

So a “20-day EWMA” would have center 9.5.

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:

\[
\text{weighted average}[0] = \text{arg}[0]; \\
\text{weighted average}[i] = (1-\alpha)\times\text{weighted average}[i-1] + \alpha\times\text{arg}[i].
\]

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).

correlation

\[
pandas.ewmcorr\(arg1, arg2=None, com=None, span=None, halflife=None, min_periods=0, freq=None, pairwise=None, how=None, ignore_na=False, adjust=True\)
\]

Exponentially-weighted moving correlation

**Parameters**

- **arg1**: Series, DataFrame, or ndarray
- **arg2**: Series, DataFrame, or ndarray, optional
  - If not supplied then will default to arg1 and produce pairwise output
- **com**: float, optional
  - Center of mass: $\alpha = 1/(1 + \text{com})$.
- **span**: float, optional
  - Specify decay in terms of span, $\alpha = 2/(\text{span} + 1)$
- **halflife**: float, optional
  - Specify decay in terms of halflife, $\alpha = 1 - \exp(\log(0.5)/\text{halflife})$
- **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
  - 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)
- **how**: string, default ‘mean’
  - Method for down- or re-sampling
- **ignore_na**: boolean, default False
  - Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior
- **pairwise**: bool, default False
If False then only matching columns between arg1 and arg2 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 Panel in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**Returns** y : type of input argument

**Notes**

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter $s$, we have that the decay parameter $\alpha$ is related to the span as $\alpha = 2/(s + 1) = 1/(1 + c)$

where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$

So a “20-day EWMA” would have center 9.5.

**When adjust is True (default), weighted averages are calculated using weights**

$(1-\alpha)^{\text{(n-1)}}, (1-\alpha)^{\text{(n-2)}}, ..., 1-\alpha, 1.$

**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).

### pandas.ewmcov

**Parameters**

arg1 : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray, optional

if not supplied then will default to arg1 and produce pairwise output

com : float, optional

Center of mass: $\alpha = 1/(1 + com)$

span : float, optional

Specify decay in terms of span, $\alpha = 2/(span + 1)$

halflife : float, optional

Specify decay in terms of halflife, $\alpha = 1 - \exp(\text{log}(0.5)/\text{halflife})$

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
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)

**how**: string, default ‘mean’

Method for down- or re-sampling

**ignore_na**: boolean, default False

Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

**pairwise**: bool, default False

If False then only matching columns between arg1 and arg2 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 Panel in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**Returns**  
y: type of input argument

**Notes**

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter $s$, we have that the decay parameter $\alpha$ is related to the span as $\alpha = \frac{2}{(s + 1)} = \frac{1}{1 + c}$

where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$

So a “20-day EWMA” would have center 9.5.

**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**:  
$\text{weighted_average}[0] = \text{arg}[0]$;  
$\text{weighted_average}[i] = (1-\alpha)*\text{weighted_average}[i-1] + \alpha*\text{arg}[i]$.  

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 $[x, \text{None}, y]$ 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 $[x, \text{None}, y]$ are $1-\alpha$ and 1 (if adjust is True), and $1-\alpha$ and $\alpha$ (if adjust is False).

### 33.3 Series

#### 33.3.1 Constructor

**Series**([data, index, dtype, name, copy, ...])  
One-dimensional ndarray with axis labels (including time series).
### 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 any 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 unique and hashable, same length as data. Index object (or other iterable of same length as data) Will default to np.arange(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**: boolean, default False
  
  Copy input data

**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>at</code></td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td><code>base</code></td>
<td>Internal property, property synonym for as_blocks()</td>
</tr>
<tr>
<td><code>blocks</code></td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td><code>dtype</code></td>
<td>return the dtype object of the underlying data</td>
</tr>
<tr>
<td><code>dtypes</code></td>
<td>return the dtypes object of the underlying data</td>
</tr>
<tr>
<td><code>empty</code></td>
<td>True if NDFrame is entirely empty [no items]</td>
</tr>
<tr>
<td><code>flags</code></td>
<td></td>
</tr>
<tr>
<td><code>ftype</code></td>
<td>return if the data is sparse</td>
</tr>
<tr>
<td><code>ftypes</code></td>
<td>return if the data is sparse</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>imag</code></td>
<td></td>
</tr>
<tr>
<td><code>is_time_series</code></td>
<td></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>ix</code></td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td><code>loc</code></td>
<td>Purely label-location based indexer for selection by label.</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, by definition 1</td>
</tr>
<tr>
<td><code>real</code></td>
<td></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>
</tbody>
</table>

*Continued on next page*
Table 33.21 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strides</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>values</td>
<td>Return Series as ndarray</td>
</tr>
</tbody>
</table>

**pandas.Series.T**

Series.T  
return the transpose, which is by definition self

**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

**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()

**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 dtypes object of the underlying data
pandas.Series.empty

Series.empty
   True if NDFrame is entirely empty [no items]

pandas.Series.flags

Series.flags

pandas.Series(ftype)

Series.ftype
   return if the data is sparseldense

pandas.Series(ftypes)

Series.ftypes
   return if the data is sparseldense

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.
   .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_time_series

Series.is_time_series

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.

.loc will raise a KeyError when the items are not found.

See more at Selection by Label

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

Returns arr: numpy.ndarray

Methods

abs()
Return an object with absolute value taken.

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.

align(other[, join, axis, level, copy, ...])
Align two object 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[, verify_integrity])
Concatenate two or more Series.

apply(func[, convert_dtype, args])
Invoke function on values of Series.

argmax([axis, out, skipna])
Index of first occurrence of maximum of values.

argmin([axis, out, skipna])
Index of first occurrence of minimum of values.

argsort([axis, kind, order])
Overrides ndarray.argsort.

as_blocks()
Convert the frame to a dict of dtype -> Constructor Types that each has a homoge
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as_matrix([columns])</code></td>
<td>Convert the frame to its Numpy-array representation.</td>
</tr>
<tr>
<td><code>asfreq(freq[, method, how, normalize])</code></td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset objects.</td>
</tr>
<tr>
<td><code>asof(where)</code></td>
<td>Return last good (non-NaN) value in TimeSeries if value is NaN for requested data point.</td>
</tr>
<tr>
<td><code>astype(dtype[, copy, raise_on_error])</code></td>
<td>Cast object to input numpy.dtype</td>
</tr>
<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>autocorr([lag])</code></td>
<td>Lag-N autocorrelation</td>
</tr>
<tr>
<td><code>between(left, right[, inclusive])</code></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right.</td>
</tr>
<tr>
<td><code>between_time(start_time, end_time[, ...])</code></td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
</tr>
<tr>
<td><code>bfill(axis, inplace, limit, downcast())</code></td>
<td>Return copy of the input with values below given value(s) truncated.</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>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><code>clip([lower, upper, out, axis])</code></td>
<td>Return copy of the input with values above given value(s) truncated.</td>
</tr>
<tr>
<td><code>clip_lower(threshold[, axis])</code></td>
<td>Perform elementwise binary operation on two Series using given function.</td>
</tr>
<tr>
<td><code>clip_upper(threshold[, axis])</code></td>
<td>Combine Series values, choosing the calling Series’s values first.</td>
</tr>
<tr>
<td><code>combine(other, func[, fill_value])</code></td>
<td>Return the compound percentage of the values for the requested axis.</td>
</tr>
<tr>
<td><code>combine_first(other)</code></td>
<td>Return selected slices of an array along given axis as a Series.</td>
</tr>
<tr>
<td><code>compound([axis, skipna, level])</code></td>
<td>Compute NDFrame with “consolidated” internals (data of each dtype grouped together).</td>
</tr>
<tr>
<td><code>compress(condition[, axis, out])</code></td>
<td>Attempt to infer better dtype for object columns.</td>
</tr>
<tr>
<td><code>consolidate([inplace])</code></td>
<td>Make a copy of this object.</td>
</tr>
<tr>
<td><code>convert_objects([convert_dates, ...])</code></td>
<td>Compute correlation with other Series, excluding missing values.</td>
</tr>
<tr>
<td><code>corr(other[, method, min_periods])</code></td>
<td>Return number of non-NA/null observations in the Series.</td>
</tr>
<tr>
<td><code>count([level])</code></td>
<td>Compute covariance with Series, excluding missing values.</td>
</tr>
<tr>
<td><code>cov(other[, min_periods])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>cummax([axis, dtype, out, skipna])</code></td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td><code>cummin([axis, dtype, out, skipna])</code></td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td><code>cumprod([axis, dtype, out, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td><code>cumsum([axis, dtype, out, skipna])</code></td>
<td>Generate various summary statistics, excluding NaN values.</td>
</tr>
<tr>
<td><code>describe([percentile_width, percentiles, ...])</code></td>
<td>1st discrete difference of object.</td>
</tr>
<tr>
<td><code>diff([periods])</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>div(other[, level, fill_value, axis])</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>divide(other[, level, fill_value, axis])</code></td>
<td>Matrix multiplication with DataFrame or inner-product with Series.</td>
</tr>
<tr>
<td><code>dot(other)</code></td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td><code>drop(labels[, axis, level, inplace, errors])</code></td>
<td>Return Series with duplicate values removed.</td>
</tr>
<tr>
<td><code>drop_duplicates([take_last, inplace])</code></td>
<td>Return Series without null values.</td>
</tr>
<tr>
<td><code>dropna([axis, inplace])</code></td>
<td>Return boolean Series denoting duplicate values.</td>
</tr>
<tr>
<td><code>dt</code></td>
<td>alias of CombinedDatetimelikeProperties</td>
</tr>
<tr>
<td><code>duplicated([take_last])</code></td>
<td>Return boolean Series denoting duplicate values.</td>
</tr>
<tr>
<td><code>eq(other[, axis])</code></td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td><code>equals(other)</code></td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><code>factorize([sort, na_sentinel])</code></td>
<td>Synonym for NDFrame.fillna(method='ffill').</td>
</tr>
<tr>
<td><code>ffill([axis, inplace, limit, downcast])</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>fillna([value, method, axis, inplace, ...])</code></td>
<td>Restrict the info axis to set of items or wildcard.</td>
</tr>
<tr>
<td><code>filter([items, like, regex, axis])</code></td>
<td>Convenience method for subsetting initial periods of time series data.</td>
</tr>
<tr>
<td><code>first(offset)</code></td>
<td>Return label for first non-NA/null value.</td>
</tr>
<tr>
<td><code>first_valid_index()</code></td>
<td>Integer division of series and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>floordiv(other[, level, fill_value, axis])</code></td>
<td>Read delimited file into Series.</td>
</tr>
<tr>
<td><code>from_array(arr[, level, fill_value, axis])</code></td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.).</td>
</tr>
<tr>
<td><code>from_csv(path[, index, name, dtypes, copy, ...])</code></td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td><code>ge(other[, axis])</code></td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.).</td>
</tr>
<tr>
<td><code>get(key[, default])</code></td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td><code>get_dtype_counts()</code></td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>get_ftype_counts()</code></td>
<td>Return the counts of ftypes in this object</td>
</tr>
<tr>
<td><code>get_value(label[, takeable])</code></td>
<td>Quickly retrieve single value at passed index label same as values (but handles sparseness conversions); is a view</td>
</tr>
<tr>
<td><code>get_values()</code></td>
<td>Group series using mapper (dict or key function, apply given function</td>
</tr>
<tr>
<td><code>groupby((by, axis, level, as_index, sort, ...))</code></td>
<td>return if I have any nans; enables various perf speedups</td>
</tr>
<tr>
<td><code>gt(other[, axis])</code></td>
<td>Returns first n rows</td>
</tr>
<tr>
<td><code>hist((by, ax, grid, xlabels, xlabel, xrot, ...))</code></td>
<td>Draw histogram of the input series using matplotlib</td>
</tr>
<tr>
<td><code>idxmax((axis, out, skipna))</code></td>
<td>Index of first occurrence of maximum of values.</td>
</tr>
<tr>
<td><code>idxmin((axis, out, skipna))</code></td>
<td>Index of first occurrence of minimum of values.</td>
</tr>
<tr>
<td><code>iget(i[, axis])</code></td>
<td>Return the i-th value or values in the Series by location</td>
</tr>
<tr>
<td><code>iget_value(i[, axis])</code></td>
<td>Return the i-th value or values in the Series by location</td>
</tr>
<tr>
<td><code>interpolate([method, axis, limit, inplace, ...])</code></td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td><code>irow(i[, axis])</code></td>
<td>Return the i-th value or values in the Series by location</td>
</tr>
<tr>
<td><code>isin(values)</code></td>
<td>Return a boolean Series showing whether each element in the Series is exist</td>
</tr>
<tr>
<td><code>isnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td><code>iters()</code></td>
<td>return the first element of the underlying data as a python scalar</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
<tr>
<td><code>kurt((axis, skipna, level, numeric_only))</code></td>
<td>Return unbiased kurtosis over requested axis using Fishers definition of kurtosis</td>
</tr>
<tr>
<td><code>kurtosis((axis, skipna, level, numeric_only))</code></td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td><code>last(offset)</code></td>
<td>Return label for last non-NA/null value</td>
</tr>
<tr>
<td><code>last_valid_index()</code></td>
<td>Return the last non-NA/null value</td>
</tr>
<tr>
<td><code>le(other[, axis])</code></td>
<td>Return a boolean Series showing whether each element in the Series is exist</td>
</tr>
<tr>
<td><code>load(path)</code></td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td><code>mad([axis, skipna, level])</code></td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td><code>map(arg[, na_action])</code></td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td><code>max([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>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><code>min([axis, skipna, level, numeric_only])</code></td>
<td>Modulo of series and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td><code>mode()</code></td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>mul(other[, level, fill_value, axis])</code></td>
<td>Returns the mode(s) of the dataset.</td>
</tr>
<tr>
<td><code>multiply(other[, level, fill_value, axis])</code></td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>nlargest([n, take_last])</code></td>
<td>Return the largest n elements.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td><code>nsmallest([n, take_last])</code></td>
<td>Return the smallest n elements.</td>
</tr>
<tr>
<td><code>nunique([dropna])</code></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><code>order([na_last, ascending, kind, ...])</code></td>
<td>Sorts Series object, by value, maintaining index-value link.</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>plot(data[, kind, ax, figsize, use_index, ...])</code></td>
<td>Make plots of Series using matplotlib / pylab.</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 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>
</tbody>
</table>
ptp([axis, out])
put(*args, **kwargs)
quantile([q])
radd(other[, level, fill_value, axis])
rank([method, na_option, ascending, pct])
ravel([order])
rdiv(other[, level, fill_value, axis])
reindex([index])
reindex_axis(labels[, axis])
reindex_like(other[, method, copy, limit])
rename([index])
rename_axis(mapper[, axis, copy, inplace])
reorder_levels(order)
repeat( reps)
replace(to_replace, value, inplace, limit, ...)
resample(rule[, how, axis, fill_method, ...])
reset_index([index])
reshape(*args, **kwargs)
rfloordiv( other[, level, fill_value, axis])
rmod( other[, level, fill_value, axis])
rmul( other[, level, fill_value, axis])
round([decimals, out])
rpow( other[, level, fill_value, axis])
rsub( other[, level, fill_value, axis])
rtruediv( other[, level, fill_value, axis])
sample([n, frac, replace, weights, ...])
save(path)
searchsorted(v[, side, sorter])
select(crit[, axis])
sem([axis, skipna, level, ddof, numeric_only])
set_axis(axis, labels)
set_value(label, value[, takeable])
shift([periods, freq, axis])
skew([axis, skipna, level, numeric_only])
slice_shift([periods, axis])
sort([axis, ascending, kind, na_position, ...])
sort_index([ascending])
sortlevel(i, j[, copy])
squeeze()
std([axis, skipna, level, ddof, numeric_only])
str
sub(other[, level, fill_value, axis])
subtract(other[, level, fill_value, axis])
sum([axis, skipna, level, numeric_only])
swapaxes(axis1, axis2[, copy])
swaplevel(i, j[, copy])
tail([n])
take(indices[, axis, convert, is_copy])
to_clipboard([excel, sep])
to_csv(path[, index, sep, na_rep, ...])
to_dense()
to_dict()
to_frame([name])
    Convert Series to DataFrame

to_hdf(path_or_buf, key, **kwargs)
    activate the HDFStore

to_json([path_or_buf, orient, date_format, ...])
    Convert the object to a JSON string.

to_msgpack([path_or_buf])
    msgpack (serialize) object to input file path

to_period([freq, copy])
    Convert TimeSeries from DatetimeIndex to PeriodIndex with desired

to_pickle(path)
    Pickle (serialize) object to input file path

to_sparse([kind, fill_value])
    Convert Series to SparseSeries

to_sql(name, con[, flavor, schema, ...])
    Write records stored in a DataFrame to a SQL database.

to_string([buf, na_rep, float_format, ...])
    Render a string representation of the Series

to_timestamp([freq, how, copy])
    Cast to datetimeindex of timestamps, at beginning of period

tolist()
    return the transpose, which is by definition self

truediv(other[, level, fill_value, axis])
    Floating division of series and other, element-wise (binary operator truediv).

truncate([before, after, axis, copy])
    Truncates a sorted NDFrame before and/or after some particular dates.

tshift([periods, freq, axis])
    Shift the time index, using the index’s frequency if available

tz_convert(tz[, axis, level, copy])
    Convert tz-aware axis to target time zone.

tz_localize(*args, **kwargs)
    Localize tz-naive TimeSeries to target time zone

unique()
    Return array of unique values in the object.

unstack([level])
    Unstack, a.k.a.

update(other)
    Modify Series in place using non-NA values from passed Series.

valid([inplace])

value_counts([normalize, sort, ascending, ...])
    Returns object containing counts of unique values.

var([axis, skipna, level, ddof, numeric_only])
    Return unbiased variance over requested axis.

view([dtype])

where(cond, other, inplace, axis, level, ...)
    Return an object of same shape as self and whose corresponding entries are from

xs(key[, axis, level, copy, drop_level])
    Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

pandas.Series.abs

    Series.abs()
        Return an object with absolute value taken. Only applicable to objects that are all numeric

        Returns abs: type of caller

pandas.Series.add

    Series.add(other, level=None, fill_value=None, axis=0)
        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)

                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.radd

**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.align**

Series.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)
Align two objects on their axes with the specified join method for each axis Index

**Parameters**
- other : DataFrame or Series
- join : {'outer', 'inner', 'left', 'right'}, default 'outer'
- axis : allowed axis of the other object, default None
  Align on index (0), columns (1), or both (None)
- level : int or level name, default None
  Broadcast across a level, matching Index values on the passed MultiIndex level
- copy : 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, 1}, default 0
  Filling axis, method and limit

**Returns**
- (left, right) : (type of input, type of other)
  Aligned objects
**pandas.Series.all**

Series.all \(\text{(axis=None, bool_only=None, skipna=None, level=None, **kwargs)}\)

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 data. If None, will attempt to use everything, then use only boolean data

**Returns**
- **all** : scalar or Series (if level specified)

**pandas.Series.any**

Series.any \(\text{(axis=None, bool_only=None, skipna=None, level=None, **kwargs)}\)

Return 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 data. If None, will attempt to use everything, then use only boolean data

**Returns**
- **any** : scalar or Series (if level specified)

**pandas.Series.append**

Series.append \(\text{(to_append, verify_integrity=False)}\)

Concatenate two or more Series.

**Parameters**
- **to_append** : Series or list/tuple of Series
  - **verify_integrity** : boolean, default False
    - If True, raise Exception on creating index with duplicates

**Returns**
- **appended** : Series
**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
  - `args`: tuple  
    Positional arguments to pass to function in addition to the value

- **Additional keyword arguments will be passed as keywords to the function**

- **Returns**
  - `y`: Series or DataFrame if func returns a Series

See also:

**Series.map** For element-wise operations

**pandas.Series.argmax**

`Series.argmax(axis=None, out=None, skipna=True)`  
Index of 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.

**pandas.Series.argmin**

`Series.argmin(axis=None, out=None, skipna=True)`  
Index of 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`.

**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'
    - Choice of sorting algorithm. See `np.sort` for more information. 'mergesort' is the only stable algorithm
  - `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()`  
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

**NOTE:** the dtypes of the blocks WILL BE PRESERVED HERE (unlike in `as_matrix`)

- **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.

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)
```

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**

- `freq` : DateOffset object, or string
- `method` : {'backfill', 'bfill', 'pad', 'ffill', 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 method
- `how` : {'start', 'end'}, default end
  For PeriodIndex only, see PeriodIndex.asfreq
- `normalize` : bool, default False
  Whether to reset output index to midnight

**Returns**

converted : type of caller

**pandas.Series.asof**

```
Series.asof(where)
```

Return last good (non-NaN) value in TimeSeries if value is NaN for requested date.

If there is no good value, NaN is returned.

**Parameters**

- `where` : date or array of dates

**Returns**

value or NaN

**Notes**

Dates are assumed to be sorted

**pandas.Series.astype**

```
Series.astype(dtype, copy=True, raise_on_error=True, **kwargs)
```

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

**Parameters**

- `dtype` : numpy.dtype or Python type
- `raise_on_error` : raise on invalid input
- `kwargs` : keyword arguments to pass on to the constructor

**Returns**

casted : type of caller
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=None, inplace=False, limit=None, downcast=None)
Synonym for NDFrame.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 a 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, out=None, axis=None)
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.

Returns
- clipped: Series

Examples

```python
>>> df
   0
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
0   1
```

33.3. Series
pandas.Series.clip_lower

Series.clip_lower (threshold, axis=None)

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.

Returns clipped : same type as input

See also:

clip

pandas.Series.clip_upper

Series.clip_upper (threshold, axis=None)

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.

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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**
- **compounded**: scalar or Series (if level specified)

### pandas.Series.compress

**Series.compress** *(condition, axis=0, out=None, **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)*

Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray). Mainly an internal API function, but available here to the savvy user

**Parameters**
- **inplace**: boolean, default False
If False return new object, otherwise modify existing object

**Returns** **consolidated** : type of caller

### pandas.Series.convert_objects

#### Series.convert_objects

```python
Series.convert_objects(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)
```

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 totimedelta 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

### pandas.Series.copy

#### Series.copy

```python
Series.copy(deep=True)
```

Make a copy of this object

**Parameters**

- **deep** : boolean or string, default True
  
  Make a deep copy, i.e. also copy data

**Returns** **copy** : type of caller

### pandas.Series.corr

#### Series.corr

```python
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, dtype=None, out=None, skipna=True, **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**

```
max : scalar
```

**pandas.Series.cummin**

```
Series.cummin (axis=None, dtype=None, out=None, skipna=True, **kwargs)
```

Return cumulative min 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**

```
min : scalar
```
pandas.Series.cumprod

`Series.cumprod(axis=None, dtype=None, out=None, skipna=True, **kwargs)`

Return cumulative prod 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**
- `prod`: scalar

pandas.Series.cumsum

`Series.cumsum(axis=None, dtype=None, out=None, skipna=True, **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**
- `sum`: scalar

pandas.Series.describe

`Series.describe(percentile_width=None, percentiles=None, include=None, exclude=None)`

Generate various summary statistics, excluding NaN values.

**Parameters**
- `percentile_width`: float, deprecated
  
  The `percentile_width` argument will be removed in a future version. Use `percentiles` instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75
- `percentiles`: array-like, optional
  
  The percentiles to include in the output. Should all be in the interval [0, 1]. By default `percentiles` is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.
- `include`, `exclude`: list-like, ‘all’, or None (default)
  
  Specify the form of the returned result. Either:
  
  - None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
  - A list of dtypes or strings to be included/excluded. To select all numeric types use `numpy numpy.number`. To select categorical objects use `type object`. See also the `select_dtypes` documentation. eg. df.describe(include=['O'])
  - If include is the string ‘all’, the output column-set will match the input one.

**Returns**
- `summary`: NDFrame of summary statistics

**See also:**

`DataFrame.select_dtypes`
Notes

The output DataFrame index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the count and most common pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.

**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).
pandas: powerful Python data analysis toolkit, Release 0.16.2

Equivalent to \texttt{series / other}, but with support to substitute a \texttt{fill\_value} for missing data in one of the inputs.

\textbf{Parameters} \texttt{other: Series or scalar value}

- \texttt{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

- \texttt{level}: int or name
  
  Broadcast across a level, matching Index values on the passed MultiIndex level

\textbf{Returns} \texttt{result: Series}

\textbf{See also:}

- \texttt{Series.rtruediv}

\textbf{pandas.Series.dot}

\texttt{Series.dot(other)}

Matrix multiplication with DataFrame or inner-product with Series objects

\textbf{Parameters} \texttt{other: Series or DataFrame}

\textbf{Returns} \texttt{dot\_product: scalar or Series}

\textbf{pandas.Series.drop}

\texttt{Series.drop(labels, axis=0, level=None, inplace=False, errors=’raise’)}

Return new object with labels in requested axis removed

\textbf{Parameters} \texttt{labels: single label or list-like}

- \texttt{axis}: int or axis name
- \texttt{level}: int or level name, default None
  
  For MultiIndex

- \texttt{inplace}: bool, default False
  
  If True, do operation inplace and return None.

- \texttt{errors}: {‘ignore’, ‘raise’}, default ‘raise’
  
  If ‘ignore’, suppress error and existing labels are dropped.
  
  New in version 0.16.1.

\textbf{Returns} \texttt{dropped: type of caller}

\textbf{pandas.Series.drop_duplicates}

\texttt{Series.drop_duplicates(\text{take\_last=False, inplace=False})}

Return Series with duplicate values removed

\textbf{Parameters} \texttt{take\_last}: boolean, default False

Take the last observed index in a group. Default first
inplace : boolean, default False

If True, performs operation in place 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
>>> s.dt.hour
>>> s.dt.second
>>> s.dt.quarter
```

Returns a Series indexed like the original Series. Raises TypeError if the Series does not contain datetimelike values.

pandas.Series.duplicated

Series.duplicated (take_last=False)

Return boolean Series denoting duplicate values

Parameters take_last : boolean, default False

Take the last observed index in a group. Default first

Returns duplicated : Series

pandas.Series.eq

Series.eq (other, axis=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.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 NDFrame.fillna(method=’ffill’)

pandas.Seriesfillna

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.

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

**pandas.Series.filter**

*Series.filter*(items=None, like=None, regex=None, axis=None)

Restrict the info axis to set of items or wildcard

**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 None
  
  The axis to filter on. By default this is the info axis. The “info axis” is the axis that is used when indexing with[]. For example, df = DataFrame({'a': [1, 2, 3, 4]})); df['a']. So, the DataFrame columns are the info axis.

**Notes**

Arguments are mutually exclusive, but this is not checked for

**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**

* ts.last('10D') -> First 10 days

**pandas.Series.first_valid_index**

*Series.first_valid_index*()

Return label for first non-NA/null value
pandas.Series.floordiv

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

classmethod Series.from_array(arr, index=None, name=None, dtype=None, copy=False, fast-
path=False)

pandas.Series.from_csv

classmethod Series.from_csv(path, sep=’,’, parse_dates=True, header=None, index_col=0, en-
coding=None, infer_datetime_format=False)
Read delimited file into Series

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 0
Row to use at 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 ver-
sions 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

**pandas.Series.ge**

```
Series.ge (other, axis=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

**Returns** value : type of items contained in object

**pandas.Series.get_dtypes**

```
Series.get_dtypes ()
```

Return the counts of dtypes in this object

**pandas.Series.get_ftypes**

```
Series.get_dtypes ()
```

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

**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)

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 / list of functions, dict, Series, or tuple / list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups.
- 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.
- 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()
```

```python
>>> data.groupby(["col1", "col2"])["col3"].mean()  
```

DataFrame with hierarchical index

```python
>>> data.groupby(["col1", "col2"]).mean() 
```

pandas.Series.gt

Series.gt(other, axis=None)

pandas.Series.hasnans

Series.hasnans()

return if I have any nans; enables various perf speedups
pandas.Series.head

Series.head\((n=5)\)

Returns first \(n\) rows

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\((axis=None, out=None, skipna=True)\)

Index of 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.

**pandas.Series.idxmin**

Series `idxmin (axis=None, out=None, skipna=True)`
Index of 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.

**pandas.Series.iget**

Series `iget (i, axis=0)`
Return the i-th value or values in the Series by location

**Parameters**
- `i` : int, slice, or sequence of integers

**Returns** `value` : scalar (int) or Series (slice, sequence)

**pandas.Series.iget_value**

Series `iget_value (i, axis=0)`
Return the i-th value or values in the Series by location

**Parameters**
- `i` : int, slice, or sequence of integers

**Returns** `value` : scalar (int) or Series (slice, sequence)

**pandas.Series.interpolate**

Series `interpolate (method='linear', axis=0, limit=None, inplace=False, downcast=None, **kwargs)`
Interpolate values according to different methods.
**Parameters** method : 


  - ‘linear’: ignore the index and treat the values as equally spaced. 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` with the order given both ‘polynomial’ and ‘spline’ require that you also specify and order (int) e.g. df.interpolate(method='polynomial', order=4)
  - ‘krogh’, ‘piecewise_polynomial’, ‘spline’, and ‘pchip’ are all wrappers around the scipy interpolation methods of similar names. See the scipy documentation for more on their behavior: http://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-interpolation

**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.

**inplace** : bool, default False

Update the NDFrame in place if possible.

**downcast** : optional, ‘infer’ or None, defaults to None

Downcast dtypes if possible.

**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.Series.irow

```python
Series.irow(i, axis=0)
```

Return the i-th value or values in the Series by location

**Parameters**
- `i` : int, slice, or sequence of integers

**Returns**
- `value` : scalar (int) or Series (slice, sequence)

pandas.Series.isin

```python
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` : list-like
  - The sequence of values to test. Passing in a single string will raise a TypeError. Instead, turn a single string into a list of one element.

**Returns**
- `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.isnull

```python
Series.isnull()
```

Return a boolean same-sized object indicating if the values are null

**See also:**
- `notnull` boolean inverse of isnull
pandas.Series.item

```
Series.item()
```

Return the first element of the underlying data as a python scalar

pandas.Series.iteritems

```
Series.iteritems()
```

Lazily iterate over (index, value) tuples

pandas.Series.iterkv

```
Series.iterkv(*args, **kwargs)
```

Iteritems alias used to get around 2to3. Deprecated

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 Fishers 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, 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 data. If None, will attempt to use everything, then use only numeric data

  - **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 Fishers 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, 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 data. If None, will attempt to use everything, then use only numeric data

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

ts.last('5M') -> Last 5 months

pandas.Series.last_valid_index

Series.last_valid_index()
   Return label for last non-NA/null value

pandas.Series.le

Series.le(other, axis=None)

pandas.Series.load

Series.load(path)
   Deprecated. Use read_pickle instead.

pandas.Series.lt

Series.lt(other, axis=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, 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 data. If None, will attempt to use everything, then use only numeric data

**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

**Returns y** : Series

same index as caller

**Examples**

```python
def x
    one 1
    two 2
    three 3

>>> y
    1 foo
    2 bar
    3 baz

>>> x.map(y)
    one foo
    two bar
    three baz
```

### pandas.Series.mask

**Series.mask**(cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array

**other** : scalar or NDFrame

**inplace** : boolean, default False

Whether to perform the operation in place on the data
**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use
    only numeric data

**Returns**

- **median**: scalar or Series (if level specified)

**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, 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 data. If None, will attempt to use everything, then use
    only numeric data

**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 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.rmod

pandas.Series.mode

Series.mode ()

Returns the mode(s) of the dataset.

Empty if nothing occurs at least 2 times. Always returns Series even if only one value.

Parameters sort : bool, default True

If True, will lexicographically sort values, if False skips sorting. Result ordering when sort=False is not defined.

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*, *axis=None*)

**pandas.Series.nlargest**

Series.nlargest(*n=5*, *take_last=False*)

Return the largest n elements.

**Parameters**  
n : int

Return this many descending sorted values

take_last : bool

Where there are duplicate values, take the last duplicate

**Returns**  
top_n : Series

The n largest values in the Series, in sorted order

**See also:**

`Series.nsmallest`

**Notes**

Faster than .order(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(1e6))
>>> s.nlargest(10)  # only sorts up to the N requested
```

**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
>>> s = pd.Series([0, 3, 0, 4])
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
1  3
3  4
dtype: int64
```

**pandas.Series.notnull**

*Series.notnull()*

Return a boolean same-sized object indicating if the values are not null

**See also:**

`isnull` boolean inverse of notnull

**pandas.Series.nsmallest**

*Series.nsmallest*(n=5, take_last=False)

Return the smallest n elements.

**Parameters**

- **n**: int
  
  Return this many ascending sorted values

- **take_last**: bool
  
  Where there are duplicate values, take the last duplicate

**Returns**

- **bottom_n**: Series
  
  The n smallest values in the Series, in sorted order

**See also:**

`Series.nlargest`

**Notes**

Faster than `.order().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(1e6))
>>> s.nsmallest(10)  # only sorts up to the N requested
```

**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.order**

Series.order(na_last=None, ascending=True, kind='quicksort', na_position='last', inplace=False)

Sorts Series object, by value, maintaining index-value link. This will return a new Series by default. Series.sort is the equivalent but as an inplace method.

**Parameters**

- **na_last** : boolean (optional, default=True) (DEPRECATED; use na_position)
  
  Put NaN’s at beginning or end

- **ascending** : boolean, default True
  
  Sort ascending. Passing False sorts descending

- **kind** : {'mergesort', 'quicksort', 'heapsort'}, default ‘quicksort’
  
  Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm

- **na_position** : {'first', 'last'} (optional, default='last')
  
  ‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end

- **inplace** : boolean, default False
  
  Do operation in place.

**Returns**

- **y** : Series

See also:

- Series.sort

**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)`

New in version 0.16.2.

**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**: positional arguments passed into `func`.

- **kwargs**: 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 on `Series` or `DataFrames`. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (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.Series.plot**

`Series.plot(data, kind='line', ax=None, figsize=None, use_index=True, title=None, grid=None, legend=False, 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, label=None, secondary_y=False, **kwds)`

Make plots of Series using matplotlib / pylab.

**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

  Title to use for the plot

- **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)

**layout** : tuple (optional)
(rows, columns) for the layout of the plot

**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.

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, 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 data. If None, will attempt to use everything, then use only numeric data
pandas: powerful Python data analysis toolkit, Release 0.16.2

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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns **prod** : scalar or Series (if level specified)

**pandas.Series.ptp**

Series.ptp(*axis=None, out=None*)

**pandas.Series.put**

Series.put(*args, **kwargs*)

return a ndarray with the values put

See also:

    numpy.ndarray.put

**pandas.Series.quantile**

Series.quantile(*q=0.5*)

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

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
>>> 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

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

Series.rank(method='average', na_option='keep', ascending=True, pct=False)
Compute data ranks (1 through n). Equal values are assigned a rank that is the average of the ranks of
those values

Parameters  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

na_option : {'keep'}
keep: leave NA values where they are

ascending : boolean, default True
False for ranks by high (1) to low (N)

pct : boolean, default False
Computes percentage rank of data
Returns ranks : Series

pandas.Series.ravel

Series.ravel(order='C')
Return the flattened underlying data as an ndarray
See also:
numpy.ndarray.ravel

pandas.Series.rdiv

Series.rdiv(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.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 (can be specified in order, or as keywords) New labels / index to conform to. Preferably an Index object to avoid duplicating data
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

**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 size gap to forward or backward fill

**Returns reindexed**: Series

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

**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)

return an object with matching indices to myself

**Parameters other**: Object

**method**: string or None

**copy**: boolean, default True

**limit**: int, default None

Maximum size gap to forward or backward fill

**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 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.

**Parameters index**: dict-like or function, optional

Transformation to apply to that axis values

**copy**: boolean, default True
pandas: powerful Python data analysis toolkit, Release 0.16.2

Also copy underlying data

inplace : boolean, default False

Whether to return a new Series. If True then value of copy is ignored.

Returns renamed : Series (new object)

pandas.Series.rename_axis

Series.rename_axis (mapper, axis=0, copy=True, inplace=False)

Alter index and / or columns using 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.

Parameters mapper : dict-like or function, optional

axis : int or string, default 0

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Returns renamed : type of caller

pandas.Series.reorder_levels

Series.reorder_levels (order)

Rearrange index levels using input order. May not drop or duplicate levels

Parameters order : list of int representing new level order.

(reference level by number or key)

axis : where to reorder levels

Returns type of caller (new object)

pandas.Series.repeat

Series.repeat (reps)

return a new Series with the values repeated reps times

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 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 form 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**
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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)*

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters**

- `rule`: string
  - the offset string or object representing target conversion
- `how`: string
  - method for down- or re-sampling, default to `‘mean’` for downsampling
- `axis`: int, optional, default 0
- `fill_method`: string, default None
  - fill_method for upsampling
- `closed`: {'right', 'left'}
  - Which side of bin interval is closed
- `label`: {'right', 'left'}
  - Which bin edge label to label bucket with
- `convention`: {'start', 'end', 's', 'e'}
- `kind`: “period”/”timestamp”
- `loffset`: timedelta
  - Adjust the resampled time labels
- `limit`: int, default None
Maximum size gap to when reindexing with `fill_method`

```markdown
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

---

### 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

---

### pandas.Series.reshape

```
Series.reshape(*args, **kwargs)
```

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.take`

---

### 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.round

Series.round (decimals=0, out=None)
Return a with each element rounded to the given number of decimals.
Refer to numpy.around for full documentation.

See also:
**numpy.around** equivalent function

**pandas.Series.rpow**

`Series.rpow(other, level=None, fill_value=None, axis=0)`

Exponential power of series and other, element-wise (binary operator `rpow`).

Equivalent to `other ** series`, but with support to substitute a `fill_value` for missing data in 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.

New in version 0.16.1.

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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

pandas.Series.save

Series.save(path)

Deprecated. Use to_pickle instead
**pandas.Series.searchsorted**

*pandas.Series.searchsorted*(v, 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 *v* were inserted before the indices, the order of *self* would be preserved.

**Parameters**

- **v**: array_like
  Values to insert into *a*.

- **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 *a*).

- **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 *v*.

**See also:**

*Series.sort, Series.order, 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.searchsorted([1, 2], side='right', sorter=[0, 2, 1])
array([1, 3])
```
**pandas.Series.select**

```
Series.select (crit, axis=0)
```

Return data corresponding to axis labels matching criteria

**Parameters**
- **crit** : function
  - To be called on each index (label). Should return True or False
- **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.

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
- **numeric_only** : boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**
- **sem** : scalar or Series (if level specified)

**pandas.Series.set_axis**

```
Series.set_axis (axis, labels)
```

public version of axis assignment

**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

**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, **kwargs)
```
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 datetools 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, 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 data. If None, will attempt to use everything, then use only numeric data

**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**

```python
Series.sort(axis=0, ascending=True, kind='quicksort', na_position='last', inplace=True)
```

Sort values and index labels by value. This is an inplace sort by default. Series.order is the equivalent but returns a new Series.

**Parameters**
- **axis** : int (can only be zero)
- **ascending** : boolean, default True
  - Sort ascending. Passing False sorts descending
- **kind** : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'
  - Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm
- **na_position** : {'first', 'last'} (optional, default='last')
  - ‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end
- **inplace** : boolean, default True
  - Do operation in place.

**See also:**
- `Series.order`

**pandas.Series.sort_index**

```python
Series.sort_index(ascending=True)
```

Sort object by labels (along an axis)

**Parameters**
- **ascending** : boolean or list, default True
  - Sort ascending vs. descending. Specify list for multiple sort orders

**Returns**
- **sorted_obj** : Series

**Examples**

```python
>>> result1 = s.sort_index(ascending=False)
>>> result2 = s.sort_index(ascending=[1, 0])
```

**pandas.Series.sortlevel**

```python
Series.sortlevel(level=0, ascending=True, sort_remaining=True)
```

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

**pandas.Series.squeeze**

Series.squeeze()
squeeze length 1 dimensions

**pandas.Series.std**

Series.std(\text{axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs})
Return unbiased 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
  - **numeric_only**: boolean, default None
    - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

- **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(\text{other, level=None, fill_value=None, axis=0})
Subtraction of series and other, element-wise (binary operator sub).

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.rsub*

**pandas.Series.subtract**

**Series.subtract** *(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 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*

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, j, 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.

Returns swapped: Series

**pandas.Series.tail**

Series.tail(n=5)
Returns last n rows

**pandas.Series.take**

Series.take(indices, axis=0, convert=True, is_copy=False)
return Series corresponding to requested indices

Parameters indices: list / array of ints
convert: translate negative to positive indices (default)

Returns taken: Series

See also:
numpy.ndarray.take

**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 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.Series.to_csv
```

```
Series.to_csv(path, index=True, sep=',', na_rep='', float_format=None, header=False, index_label=None, mode='w', nanRep=None, encoding=None, date_format=None, decimal='').
```

Write Series to a comma-separated values (csv) file

Parameters:
- **path**: string file path or file handle / StringIO. 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()
Convert Series to {label -> value} dict

Returns value_dict : dict

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)
activate the HDFStore

Parameters path_or_buf : the path (string) or buffer to put the store

key : string
    identifier for the group in the store

mode : optional, {'a', 'w', 'r', 'r+'}, default 'a'
      'r' Read-only; no data can be modified.
      '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

complevel : int, 1-9, default 0
If a complib is specified compression will be applied where possible

`complib` : {'zlib', 'bzip2', 'lz4', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

`fletcher32` : bool, default False

If applying compression use the fletcher32 checksum

---

**pandas.Series.to_json**

Series.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

**date_format** : {'epoch', 'iso'}

Type of date conversion. *epoch* = epoch milliseconds, *iso* = ISO8601, 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 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.

Returns same type as input object with filtered info axis

pandas.Series.to_msgpack

Series.to_msgpack *(path_or_buf=None, **kwargs)*

msgpack (serialize) object to input file path

**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.Series.to_period

Series.to_period *(freq=None, copy=True)*

Convert TimeSeries from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

**Parameters**

- **freq**: string, default

**Returns**

- **ts**: TimeSeries with PeriodIndex

pandas.Series.to_pickle

Series.to_pickle *(path)*

Pickle (serialize) object to input file path

**Parameters**

- **path**: string

  File path

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='sqlite', 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', 'mysql'}, default 'sqlite'
  - The flavor of SQL to use. Ignored when using SQLAlchemy engine. ‘mysql’ is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.
- **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, 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)

**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**: TimeSeries with DatetimeIndex

### pandas.Series.tolist

**Series.tolist()**

Convert Series to a nested list

### pandas.Series.transpose

**Series.transpose()**

return the transpose, which is by definition self

### pandas.Series.truediv

**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

Series.truncate(before=None, after=None, axis=None, copy=True)
Truncates a sorted NDFrame before and/or after some particular dates.

Parameters before : date
Truncate before date
after : date
Truncate after date
axis : the truncation axis, defaults to the stat axis
copy : boolean, default is True,
return a copy of the truncated section

Returns truncated : type of caller

pandas.Series.tshift

Series.tshift(periods=1, freq=None, axis=0, **kwargs)
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 datetools 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 (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(*args, **kwargs)
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

Raises
- TypeError
  If the TimeSeries is tz-aware and tz is not None.

pandas.Series.unique

Series.unique()
Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

Returns
- uniques : ndarray
pandas.Series.unstack

Series.unstack(level=-1)
    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

    Returns unstacked : DataFrame

Examples

    >>> s
    one  a  1.
    one  b  2.
    two a  3.
    two b  4.

    >>> s.unstack(level=-1)
    a  b
    one 1. 2.
    two 3. 4.

    >>> s.unstack(level=0)
    one  two
    a  1. 2.
    b  3. 4.

pandas.Series.update

Series.update(other)
    Modify Series in place using non-NA values from passed Series. Aligns on index

    Parameters other : Series

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
numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

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, try_cast=False, raise_on_error=True)
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 or array
other : scalar or NDFrame
inplace : boolean, default False
Whether to perform the operation in place on the data
pandas: powerful Python data analysis toolkit, Release 0.16.2

axis : alignment axis if needed, default None
level : alignment level if needed, default None
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)

Returns wh : same type as caller

pandas.Series.xs

Series.xs (key, axis=0, level=None, copy=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.
copy : boolean [deprecated]
  Whether to make a copy of the data
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 functionality, 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 1  4  1  8  9
bar two  1  7  5  5  0
baz one  1  6  6  8  0
three  2  5  3  5  3

>>> df.xs(('baz', 'three'))
   A  B  C  D
third
2  5  3  5  3

>>> df.xs('one', level=1)
   A  B  C  D
first 3rd
bar one  1  4  1  8  9
baz one  1  6  6  8  0

>>> df.xs(('baz', 2), level=[0, 'third'])
   A  B  C  D
second
 3  3  5  3

33.3.2 Attributes

Axes

- **index**: axis labels

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.values</strong></td>
<td>Return Series as ndarray</td>
</tr>
<tr>
<td><strong>Series.dtype</strong></td>
<td>return the dtype object of the underlying data</td>
</tr>
<tr>
<td><strong>Series.ffi</strong></td>
<td>return if the data is sparse</td>
</tr>
<tr>
<td><strong>Series.shape</strong></td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td><strong>Series.nbytes</strong></td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td><strong>Series.ndim</strong></td>
<td>return the number of dimensions of the underlying data, by definition 1</td>
</tr>
<tr>
<td><strong>Series.size</strong></td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td><strong>Series.strides</strong></td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td><strong>Series.itemsize</strong></td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td><strong>Series.base</strong></td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td><strong>Series.T</strong></td>
<td>return the transpose, which is by definition self</td>
</tr>
</tbody>
</table>

**pandas.Series.values**

Series.values
Return Series as ndarray

Returns arr : numpy.ndarray
pandas.Series.dtype

Series.dtype
    return the dtype object of the underlying data

pandas.Series.dtype

Series.dtype
    return if the data is sparse|dense

pandas.Series.shape

Series.shape
    return a tuple of the shape of the underlying data

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.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.itemsize

Series.itemsize
    return the size of the dtype of the item of the underlying data

pandas.Series.base

Series.base
    return the base object if the memory of the underlying data is shared
pandas.Series.T

Series.T
return the transpose, which is by definition self

33.3.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.astype(dtype[, copy, raise_on_error])</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>Series.copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>Series.isnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>Series.notnull()</td>
<td>Return a boolean same-sized object indicating if the values are not null</td>
</tr>
</tbody>
</table>

**pandas.Series.astype**

Series.astype (dtype, copy= True, raise_on_error= True, **kwargs)
Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

- **Parameters**
  - dtype : numpy.dtype or Python type
  - raise_on_error : raise on invalid input
  - kwargs : keyword arguments to pass on to the constructor

- **Returns**
  - casted : type of caller

**pandas.Series.copy**

Series.copy (deep=True)
Make a copy of this object

- **Parameters**
  - deep : boolean or string, default True
    - Make a deep copy, i.e. also copy data

- **Returns**
  - copy : type of caller

**pandas.Series.isnull**

Series.isnull()
Return a boolean same-sized object indicating if the values are null

See also:

- notnull boolean inverse of isnull

**pandas.Series.notnull**

Series.notnull()
Return a boolean same-sized object indicating if the values are not null

See also:

- isnull boolean inverse of notnull
33.3.4 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.get(key[, default])</code></td>
<td>Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.</td>
</tr>
<tr>
<td><code>Series.at</code></td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td><code>Series.iat</code></td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td><code>Series.ix</code></td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td><code>Series.loc</code></td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td><code>Series.iloc</code></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><code>Series.__iter__()</code></td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
<tr>
<td><code>Series.iteritems()</code></td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
</tbody>
</table>

**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

**Returns**
- value : type of items contained in object

**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.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.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 hierachical 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.

`.loc` will raise a `KeyError` when the items are not found.

See more at Selection by Label

**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.

`.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.__iter__**

**Series.__iter__()**

**pandas.Series.iteritems**

**Series.iteritems()**

Lazily iterate over (index, value) tuples

For more information on `.at`, `.iat`, `.ix`, `.loc`, and `.iloc`, see the *indexing documentation*.

### 33.3.5 Binary operator functions
Series.add(other[, level, fill_value, axis])
Series.sub(other[, level, fill_value, axis])
Series.mul(other[, level, fill_value, axis])
Series.div(other[, level, fill_value, axis])
Series.truediv(other[, level, fill_value, axis])
Series.floordiv(other[, level, fill_value, axis])
Series.mod(other[, level, fill_value, axis])
Series.pow(other[, level, fill_value, axis])
Series.radd(other[, level, fill_value, axis])
Series.rsub(other[, level, fill_value, axis])
Series.rmul(other[, level, fill_value, axis])
Series.rdiv(other[, level, fill_value, axis])
Series.rtruediv(other[, level, fill_value, axis])
Series.rfloordiv(other[, level, fill_value, axis])
Series.rmod(other[, level, fill_value, axis])
Series.rpow(other[, level, fill_value, axis])
Series.combine(other, func[, fill_value])
Series.combine_first(other)
Series.round([decimals, out])
Series.lt(other[, axis])
Series.gt(other[, axis])
Series.le(other[, axis])
Series.ge(other[, axis])
Series.ne(other[, axis])
Series.eq(other[, axis])

pandas.Series.add

Series.add(other, level=None, fill_value=None, axis=0)
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)
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.radd

pandas.Series.sub

Series.sub(other, level=None, fill_value=None, axis=0)
Subtraction of series and other, element-wise (binary operator sub).

Addition of series and other, element-wise (binary operator add).
Subtraction of series and other, element-wise (binary operator sub).
Multiplication of series and other, element-wise (binary operator mul).
Floating division of series and other, element-wise (binary operator truediv).
Floating division of series and other, element-wise (binary operator truediv).
Integer division of series and other, element-wise (binary operator floordiv).
Modulo of series and other, element-wise (binary operator mod).
Exponential power of series and other, element-wise (binary operator pow).
Addition of series and other, element-wise (binary operator radd).
Subtraction of series and other, element-wise (binary operator rsub).
Multiplication of series and other, element-wise (binary operator rmul).
Floating division of series and other, element-wise (binary operator rtruediv).
Floating division of series and other, element-wise (binary operator rtruediv).
Integer division of series and other, element-wise (binary operator rfloordiv).
Modulo of series and other, element-wise (binary operator rmod).
Exponential power of series and other, element-wise (binary operator rpow).
Perform elementwise binary operation on two Series using given function
Combine Series values, choosing the calling Series’s values first.
Return a with each element rounded to the given number of decimals.
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`

### 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.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 \( \text{result} \): Series

See also:

\[ \text{Series.rtruediv} \]

\textbf{pandas.Series.truediv}

\texttt{Series.truediv(\text{other}, level=\text{None}, fill_value=\text{None}, axis=\text{0})}

Floating division of series and other, element-wise (binary operator \text{truediv}).

Equivalent to \texttt{series / other}, but with support to substitute a \text{fill_value} for missing data in one of the inputs.

\begin{description}
\item[Parameters] \text{other}: Series or scalar value
\item[fill_value]: None or float value, default None (NaN)
\item[level]: int or name
\end{description}

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns \( \text{result} \): Series

See also:

\[ \text{Series.rtruediv} \]

\textbf{pandas.Series.floordiv}

\texttt{Series.floordiv(\text{other}, level=\text{None}, fill_value=\text{None}, axis=\text{0})}

Integer division of series and other, element-wise (binary operator \text{floordiv}).

Equivalent to \texttt{series // other}, but with support to substitute a \text{fill_value} for missing data in one of the inputs.

\begin{description}
\item[Parameters] \text{other}: Series or scalar value
\item[fill_value]: None or float value, default None (NaN)
\item[level]: int or name
\end{description}

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns \( \text{result} \): Series

See also:

\[ \text{Series.rfloordiv} \]

\textbf{pandas.Series.mod}

\texttt{Series.mod(\text{other}, level=\text{None}, fill_value=\text{None}, axis=\text{0})}

Modulo of series and other, element-wise (binary operator \text{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.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.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.
**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.add

**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.rdiv**

Series.rdiv (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.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.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
pandas: powerful Python data analysis toolkit, Release 0.16.2

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.rpow

Series.rpow(other, level=None, fill_value=None, axis=0)
Exponential power of series and other, element-wise (binary operator rpow).
Equivalent to other ** series, but with support to substitute a fill_value for missing data in 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.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.round**

Series . round (decimals=0, out=None)
Return a with each element rounded to the given number of decimals.
Refer to numpy.around for full documentation.
See also:

    numpy.around equivalent function

**pandas.Series.lt**

Series . lt (other, axis=None)

**pandas.Series.gt**

Series . gt (other, axis=None)

**pandas.Series.le**

Series . le (other, axis=None)

**pandas.Series.ge**

Series . ge (other, axis=None)

**pandas.Series.ne**

Series . ne (other, axis=None)

**pandas.Series.eq**

Series . eq (other, axis=None)
33.3.6 Function application, GroupBy

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.apply(func[, convert_dtype, args])</code></td>
<td>Invoke function on values of Series.</td>
</tr>
<tr>
<td><code>Series.map(arg[, na_action])</code></td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td></td>
<td>Group series using mapper (dict or key function, apply given function)</td>
</tr>
</tbody>
</table>

**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
- **args**: tuple  
  Positional arguments to pass to function in addition to the value

**Additional keyword arguments will be passed as keywords to the function**

**Returns**

- **y**: Series or DataFrame if func returns a Series

**See also:**

- `Series.map` For element-wise operations

**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

**Returns**

- **y**: Series  
  same index as caller

**Examples**

```python
>>> x
   one  1
   two  2
   three  3
>>> y
   1  foo
   2  bar
   3  baz
```
pandas.Series.groupby

Series.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False)

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 / list of functions, dict, Series, or tuple / list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups.
- **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
- **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()
```

33.3.7 Computations / Descriptive Stats

**Series.abs()**

Return an object with absolute value taken.

33.3. Series
### pandas.Series.abs

**Series.abs()**

Return an object with absolute value taken. Only applicable to objects that are all numeric

**Returns**

abs: type of caller

### pandas.Series.all

**Series.all([axis, bool_only, skipna, level])**

Return whether all elements are True over requested axis

**Parameters**

axis : {index (0)}

### pandas.Series.any

**Series.any([axis, bool_only, skipna, level])**

Return whether any element is True over requested axis

**Parameters**

axis : {index (0)}

### pandas.Series.autocorr

**Series.autocorr([lag])**

Lag-N autocorrelation

### pandas.Series.between

**Series.between(left, right[, inclusive])**

Return boolean Series equivalent to left <= series <= right.

### pandas.Series.clip

**Series.clip([lower, upper, out, axis])**

Trim values at input threshold(s)

### pandas.Series.clip_lower

**Series.clip_lower(threshold[, axis])**

Return copy of the input with values below given value(s) truncated

### pandas.Series.clip_upper

**Series.clip_upper(threshold[, axis])**

Return copy of input with values above given value(s) truncated

### pandas.Series.corr

**Series.corr(other[, method, min_periods])**

Compute correlation with other Series, excluding missing values

### pandas.Series.count

**Series.count([level])**

Return number of non-NA/null observations in the Series

### pandas.Series.cov

**Series.cov(other[, min_periods])**

Generate various summary statistics, excluding NaN values.

### pandas.Series.cummax

**Series.cummax([axis, dtype, out, skipna])**

1st discrete difference of object

### pandas.Series.cummin

**Series.cummin([axis, dtype, out, skipna])**

Encode the object as an enumerated type or categorical variable

### pandas.Series.cumprod

**Series.cumprod([axis, dtype, out, skipna])**

Return cumulative prod over requested axis

### pandas.Series.cumsum

**Series.cumsum([axis, dtype, out, skipna])**

Return cumulative max over requested axis

### pandas.Series.describe

**Series.describe([percentile_width, ...])**

Return cumulative min over requested axis

### pandas.Series.diff

**Series.diff([periods])**

Return cumulative cummax over requested axis

### pandas.Series.factorize

**Series.factorize([sort, na_sentinel])**

This method returns the maximum of the values in the object.

### pandas.Series.kurt

**Series.kurt([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.mad

**Series.mad([axis, skipna, level])**

This method returns the minimum of the values in the object.

### pandas.Series.max

**Series.max([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.mean

**Series.mean([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.median

**Series.median([axis, skipna, level, ...])**

This method returns the minimum of the values in the object.

### pandas.Series.min

**Series.min([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.mode

**Series.mode()**

This method returns the minimum of the values in the object.

### pandas.Series.nlargest

**Series.nlargest([n, take_last])**

This method returns the minimum of the values in the object.

### pandas.Series.nsmallest

**Series.nsmallest([n, take_last])**

This method returns the minimum of the values in the object.

### pandas.Series.pct_change

**Series.pct_change([periods, fill_method, ...])**

This method returns the minimum of the values in the object.

### pandas.Series.prod

**Series.prod([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.quantile

**Series.quantile([q])**

This method returns the minimum of the values in the object.

### pandas.Series.rank

**Series.rank([method, na_option, ascending, pct])**

This method returns the minimum of the values in the object.

### pandas.Series.sem

**Series.sem([axis, skipna, level, ddof, ...])**

This method returns the minimum of the values in the object.

### pandas.Series.skew

**Series.skew([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.std

**Series.std([axis, skipna, level, ddof, ...])**

This method returns the minimum of the values in the object.

### pandas.Series.sum

**Series.sum([axis, skipna, level, numeric_only])**

This method returns the minimum of the values in the object.

### pandas.Series.unique

**Series.unique()**

This method returns the minimum of the values in the object.

### pandas.Series.nunique

**Series.nunique([dropna])**

This method returns the minimum of the values in the object.

### pandas.Series.value_counts

**Series.value_counts([normalize, sort, ...])**

This method returns the minimum of the values in the object.
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 data. If None, will attempt to use everything, then use only
    boolean data

Returns all : scalar or Series (if level specified)

pandas.Series.any

Series.any (axis=None, bool_only=None, skipna=None, level=None, **kwargs)
    Return 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 data. If None, will attempt to use everything, then use only
        boolean data

Returns any : scalar or Series (if level specified)

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.clip**

Series.clip(lower=None, upper=None, out=None, axis=None)

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.

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
 1 -0.2
 2 -0.1
 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.Series.clip_lower**

Series.clip_lower(threshold, axis=None)

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.
Returns clipped: same type as input

See also:
clip

**pandas.Series.clip_upper**

Series.clip_upper(threshold, axis=None)
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.

Returns clipped: same type as input

See also:
clip

**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, dtype=None, out=None, skipna=True, **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
 max : scalar

pandas.Series.cummin

Series.cummin (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative min 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
 min : scalar

pandas.Series.cumprod

Series.cumprod (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative prod 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
 prod : scalar

pandas.Series.cumsum

Series.cumsum (axis=None, dtype=None, out=None, skipna=True, **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 sum: scalar

**pandas.Series.describe**

Series.describe(percentile_width=None, percentiles=None, include=None, exclude=None)

Generate various summary statistics, excluding NaN values.

Parameters percentile_width: float, deprecated

The percentile_width argument will be removed in a future version. Use percentiles instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

percentiles: array-like, optional

The percentiles to include in the output. Should all be in the interval [0, 1]. By default percentiles is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

include, exclude: list-like, ‘all’, or None (default)

Specify the form of the returned result. Either:

• None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.

• A list of dtypes or strings to be included/excluded. To select all numeric types use numpy numpy.number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])

• If include is the string ‘all’, the output column-set will match the input one.

Returns summary: NDFrame of summary statistics

See also:

DataFrame.select_dtypes

Notes

The output DataFrame index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the count and most common pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.
**pandas.Series.diff**

Series.diff\( \text{periods}=1 \)

1st discrete difference of object

**Parameters**

- **periods**: int, default 1
  - Periods to shift for forming difference

**Returns**

- **diffed**: Series

**pandas.Series.factorize**

Series.factorize\( \text{sort=\text{False}, na_sentinel=-1} \)

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.kurt**

Series.kurt\( \text{axis=\text{None}, skipna=\text{None}, level=\text{None}, numeric_only=\text{None}, **\text{kwargs}} \)

Return unbiased kurtosis over requested axis using Fishers 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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **kurt**: scalar or Series (if level specified)

**pandas.Series.mad**

Series.mad\( \text{axis=\text{None}, skipna=\text{None}, level=\text{None}} \)

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, 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 data. If None, will attempt to use everything, then
use only numeric data

Returns mad : scalar or Series (if level specified)

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, 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 data. If None, will attempt to use everything, then
use only numeric data

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, 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 data. If None, will attempt to use everything, then
use only numeric data

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)}
  - :class:`int`
    - default: `None`
    - If `None`, use the index

- **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

- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If `None`, will attempt to use everything, then use only numeric data

Returns

- **median**: scalar or Series (if level specified)

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)}
  - :class:`int`
    - default: `None`
    - If `None`, use the index

- **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

- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If `None`, will attempt to use everything, then use only numeric data

Returns

- **min**: scalar or Series (if level specified)

pandas.Series.mode

Series.mode ()

Returns the mode(s) of the dataset.

Empty if nothing occurs at least 2 times. Always returns Series even if only one value.

Parameters

- **sort**: bool, default True
  - If True, will lexicographically sort values, if False skips sorting. Result ordering when sort=False is not defined.

Returns

- **modes**: Series (sorted)
**pandas.Series.nlargest**

Series.\texttt{nlargest}(n=5, \texttt{take\_last}=False)
Return the largest \(n\) elements.

**Parameters**
\begin{itemize}
\item \texttt{n}: int
\end{itemize}

\begin{itemize}
\item Return this many descending sorted values
\item \texttt{take\_last}: bool
\end{itemize}

\begin{itemize}
\item Where there are duplicate values, take the last duplicate
\end{itemize}

**Returns**
\begin{itemize}
\item \texttt{top\_n}: Series
\end{itemize}

The \(n\) largest values in the Series, in sorted order

**See also:**
Series.\texttt{nsmallest}

**Notes**

Faster than \texttt{.order(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(1e6))
>>> s.nlargest(10)  # only sorts up to the N requested
```

**pandas.Series.nsmallest**

Series.\texttt{nsmallest}(n=5, \texttt{take\_last}=False)
Return the smallest \(n\) elements.

**Parameters**
\begin{itemize}
\item \texttt{n}: int
\end{itemize}

\begin{itemize}
\item Return this many ascending sorted values
\item \texttt{take\_last}: bool
\end{itemize}

\begin{itemize}
\item Where there are duplicate values, take the last duplicate
\end{itemize}

**Returns**
\begin{itemize}
\item \texttt{bottom\_n}: Series
\end{itemize}

The \(n\) smallest values in the Series, in sorted order

**See also:**
Series.\texttt{nlargest}

**Notes**

Faster than \texttt{.order().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(1e6))
>>> s.nsmallest(10)  # only sorts up to the N requested
```

**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.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)}
  
  Exclude NA/null values. If an entire row/column is NA, the result will be NA
  
  - **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
  
  - **numeric_only**: boolean, default None
    
    Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

- **Returns**
  - **prod**: scalar or Series (if level specified)
pandas.Series.quantile

Series.quantile(q=0.5)
Return value at the given quantile, a la numpy.percentile.

**Parameters**
- q : float or array-like, default 0.5 (50% quantile)
  
  $0 \leq q \leq 1$, the quantile(s) to compute

**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
```

pandas.Series.rank

Series.rank(method='average', na_option='keep', ascending=True, pct=False)
Compute data ranks (1 through n). Equal values are assigned a rank that is the average of the ranks of those values

**Parameters**
- 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

- na_option : {'keep'}
  
  keep: leave NA values where they are

- ascending : boolean, default True
  
  False for ranks by high (1) to low (N)

- pct : boolean, default False
  
  Computes percentage rank of data

**Returns**
- ranks : Series
**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.

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
  - `numeric_only`: boolean, default None
    - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- `sem`: scalar or Series (if level specified)

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- `skew`: scalar or Series (if level specified)

**pandas.Series.std**

`Series.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`

Return unbiased 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
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 data. If None, will attempt to use everything, then use only numeric data

**Returns std**: scalar or Series (if level specified)

### 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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns sum**: scalar or Series (if level specified)

### 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

**numeric_only**: boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns var**: scalar or Series (if level specified)
pandas.Series.unique

Series.unique()  
Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

Returns uniques : ndarray

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.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

33.3.8 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.align(other[, join, axis, level, ...])</td>
<td>Align two object on their axes with the</td>
</tr>
<tr>
<td>Series.drop(labels[, axis, level, inplace, ...])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>Series.drop_duplicates((take_last, inplace))</td>
<td>Return Series with duplicate values removed</td>
</tr>
<tr>
<td>Series.duplicated((take_last))</td>
<td>Return boolean Series denoting duplicate values</td>
</tr>
</tbody>
</table>
### pandas.Series.align

**Series**.align(\(other\), join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)

Align two object 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, 1\}, default 0
  - Filling axis, method and limit

**Returns** (left, right) : (type of input, type of other)

Aligned objects
pandas.Series.drop

Series.drop (labels, axis=0, level=None, inplace=False, errors='raise')
Return new object with labels in requested axis removed

Parameters labels : single label or list-like
axis : int or axis name
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.
New in version 0.16.1.

Returns dropped : type of caller

pandas.Series.drop_duplicates

Series.drop_duplicates (take_last=False, inplace=False)
Return Series with duplicate values removed

Parameters take_last : boolean, default False
    Take the last observed index in a group. Default first
inplace : boolean, default False
    If True, performs operation inplace and returns None.

Returns deduplicated : Series

pandas.Series.duplicated

Series.duplicated (take_last=False)
Return boolean Series denoting duplicate values

Parameters take_last : boolean, default False
    Take the last observed index in a group. Default first

Returns duplicated : Series

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.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**

ts.last('10D') -> First 10 days

**pandas.Series.head**

Series.head(n=5)
Returns first n rows

**pandas.Series.idxmax**

Series.idxmax(axis=None, out=None, skipna=True)
Index of 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 numpy.ndarray.argmax.

**pandas.Series.idxmin**

Series.idxmin(axis=None, out=None, skipna=True)
Index of 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 numpy.ndarray.argmin.
**pandas.Series.isin**

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 : list-like  
The sequence of values to test. Passing in a single string will raise a TypeError. Instead, turn a single string into a list of one element.

**Returns** 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.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.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 (can be specified in order, or as 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**:
  
  - 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 size gap to forward or backward fill

**Returns**

- **reindexed**: Series

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

---

**pandas.Series.reindex_like**

- **Series.reindex_like**(other, method=None, copy=True, limit=None)
  
  return an object with matching indicies to myself

**Parameters**

- **other**: Object
  
  - **method**: string or None
  - **copy**: boolean, default True
  - **limit**: int, default None
  
  Maximum size gap to forward or backward fill

**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 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.

**Parameters**
- **index**: dict-like or function, optional
  - Transformation to apply to that axis values
- **copy**: boolean, default True
  - Also copy underlying data
- **inplace**: boolean, default False
  - Whether to return a new Series. If True then value of copy is ignored.

**Returns**
- **renamed**: Series (new object)

**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

**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.

New in version 0.16.1.

**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 called on a DataFrame, will accept the name of a column when axis = 0. 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** Same type as caller.

### pandas.Series.select

**Series.select** *(crit, axis=0)*

Return data corresponding to axis labels matching criteria

**Parameters**

- **crit**: function
  
  To be called on each index (label). Should return True or False

- **axis**: int

**Returns** **selection**: type of caller

### pandas.Series.take

**Series.take** *(indices, axis=0, convert=True, is_copy=False)*

return Series corresponding to requested indices

**Parameters**

- **indices**: list / array of ints

- **convert**: translate negative to positive indices (default)

**Returns** **taken**: Series

**See also:**

numpy.ndarray.take

### pandas.Series.tail

**Series.tail** *(n=5)*

Returns last n rows

### pandas.Series.truncate

**Series.truncate** *(before=None, after=None, axis=None, copy=True)*

Truncates a sorted NDFrame before and/or after some particular dates.

**Parameters**

- **before**: date
  
  Truncate before date

- **after**: date
Truncate after date

axis: the truncation axis, defaults to the stat axis

copy: boolean, default is True,
      return a copy of the truncated section

Returns truncated: type of caller

pandas.Series.where

Series.where(cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array

other: scalar or NDFrame

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

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)

Returns wh: same type as caller

pandas.Series.mask

Series.mask(cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array

other: scalar or NDFrame

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

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)
33.3.9 Missing data handling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.dropna()</code></td>
<td>Return Series without null values</td>
</tr>
<tr>
<td><code>Series.fillna()</code></td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td><code>Series.interpolate()</code></td>
<td>Interpolate values according to different methods.</td>
</tr>
</tbody>
</table>

**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.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.

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

pandas.Series.interpolate

Series.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast=None, **kwargs)

Interpolate values according to different methods.

Parameters method : {'linear', 'time', 'index', 'values', 'nearest', 'zero',
'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline'
'piecewise_polynomial', 'pchip'}

- ‘linear’: ignore the index and treat the values as equally spaced. 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
is passed to scipy.interpolate.interp1d with the order given both ‘poly-
nomial’ and ‘spline’ require that you also specify and order (int) e.g.
df.interpolate(method='polynomial', order=4)
- ‘krogh’, ‘piecewise_polynomial’, ‘spline’, and ‘pchip’ are all
wrappers around the scipy interpolation methods of similar
names. See the scipy documentation for more on their behavior:
http://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-

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.

inplace : bool, default False

Update the NDFrame in place if possible.

downcast : optional, ‘infer’ or None, defaults to None

Downcast dtypes if possible.

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
```

### 33.3.10 Reshaping, sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Series.argsort</td>
<td>Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values.</td>
</tr>
<tr>
<td>pandas.Series.order</td>
<td>Sorts Series object, by value, maintaining index-value link.</td>
</tr>
<tr>
<td>pandas.Series.reorder_levels</td>
<td>Rearranges index levels using input order.</td>
</tr>
<tr>
<td>pandas.Series.sort</td>
<td>Sort values and index labels by value.</td>
</tr>
<tr>
<td>pandas.Series.sort_index</td>
<td>Sorts Series object by labels (along an axis).</td>
</tr>
<tr>
<td>pandas.Series.sortlevel</td>
<td>Sorts Series with MultiIndex by chosen level.</td>
</tr>
<tr>
<td>pandas.Series.swaplevel</td>
<td>Swap levels i and j in a MultiIndex.</td>
</tr>
<tr>
<td>pandas.Series.unstack</td>
<td>Unstack, a.k.a.</td>
</tr>
<tr>
<td>pandas.Series.searchsorted</td>
<td>Finds indices where elements should be inserted to maintain order.</td>
</tr>
</tbody>
</table>

#### 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’`
    - Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm
  - `order : ignored`

- **Returns**
  - `argsorted : Series, with -1 indicated where nan values are present`

#### pandas.Series.order

`Series.order(na_last=None, ascending=True, kind='quicksort', na_position='last', inplace=False)`

Sorts Series object, by value, maintaining index-value link. This will return a new Series by default. Series.sort is the equivalent but as an inplace method.

- **Parameters**
  - `na_last : boolean (optional, default=True)` (DEPRECATED; use na_position)
    - Put NaN’s at beginning or end
  - `ascending : boolean, default True`
    - Sort ascending. Passing False sorts descending
  - `kind : {'mergesort', 'quicksort', 'heapsort'}, default ‘quicksort’`
Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm

na_position : {'first', 'last'} (optional, default='last')
‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end

inplace : boolean, default False
Do operation in place.

Returns y : Series

See also:
Series.sort

pandas.Series.reorder_levels

Series.reorder_levels(order)
Rearrange index levels using input order. May not drop or duplicate levels

Parameters order: list of int representing new level order.
(reference level by number or key)
axis: where to reorder levels

Returns type of caller (new object)

pandas.Series.sort

Series.sort(axis=0, ascending=True, kind='quicksort', na_position='last', inplace=True)
Sort values and index labels by value. This is an inplace sort by default. Series.order is the equivalent but returns a new Series.

Parameters axis : int (can only be zero)
ascending : boolean, default True
Sort ascending. Passing False sorts descending
kind : {'mergesort', 'quicksort', 'heapsort'}, default ‘quicksort’
Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm
na_position : {'first', 'last'} (optional, default='last')
‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end
inplace : boolean, default True
Do operation in place.

See also:
Series.order
**pandas.Series.sort_index**

Series.sort_index(ascending=True)

Sort object by labels (along an axis)

**Parameters**

- **ascending**: boolean or list, default True
  
  Sort ascending vs. descending. Specify list for multiple sort orders

**Returns**

- **sorted_obj**: Series

**Examples**

```python
>>> result1 = s.sort_index(ascending=False)
>>> result2 = s.sort_index(ascending=[1, 0])
```

**pandas.Series.sortlevel**

Series.sortlevel(level=0, ascending=True, sort_remaining=True)

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

**pandas.Series.swaplevel**

Series.swaplevel(i, j, 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.

**Returns**

- **swapped**: Series

**pandas.Series.unstack**

Series.unstack(level=-1)

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

**Returns**

- **unstacked**: DataFrame

**Examples**
>>> s
one a 1.
one b 2.
two a 3.
two b 4.

>>> s.unstack(level=-1)
a  b
one 1. 2.
two 3. 4.

>>> s.unstack(level=0)
one two
a 1. 2.
b 3. 4.

pandas.Series.searchsorted

Series.searchsorted(v, 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 v were inserted before the indices, the order of self would be preserved.

Parameters
v : array_like
   Values to insert into a.
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 a).

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 v.

See also:
Series.sort, Series.order, numpy.searchsorted

Notes

Binary search is used to find the required insertion points.

Examples

>>> x = pd.Series([1, 2, 3])
>>> x
0  1
1  2
2  3
```python
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.searchsorted([1, 2], side='right', sorter=[0, 2, 1])
array([1, 3])
```

### 33.3.11 Combining / joining / merging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.append(to_append[, verify_integrity])</code></td>
<td>Concatenate two or more Series.</td>
</tr>
<tr>
<td><code>Series.replace([to_replace, value, inplace, ...])</code></td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td><code>Series.update(other)</code></td>
<td>Modify Series in place using non-NA values from passed Series.</td>
</tr>
</tbody>
</table>

**pandas.Series.append**

```python
Series.append(to_append[, verify_integrity=False])
```

Concatenate two or more Series.

**Parameters**

- **to_append**: Series or list/tuple of Series
- **verify_integrity**: boolean, default False
  
  If True, raise Exception on creating index with duplicates

**Returns**

- **appended**: Series

**pandas.Series.replace**

```python
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: regexes 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 form 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.Series.update**

Series.update(other)

Modify Series in place using non-NA values from passed Series. Aligns on index

**Parameters** other : Series

### 33.3.12 Time series-related

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.asfreq(freq[, method, how, normalize])</td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.</td>
</tr>
<tr>
<td>Series.shift(where)</td>
<td>Return last good (non-NaN) value in TimeSeries if value is NaN for requested dt</td>
</tr>
<tr>
<td>Series.first_valid_index()</td>
<td>Return label for first non-NA/null value</td>
</tr>
<tr>
<td>Series.last_valid_index()</td>
<td>Return label for last non-NA/null value</td>
</tr>
<tr>
<td>Series.resample(rule[, how, axis, ...])</td>
<td>Convenience method for frequency conversion and resampling of regular time-series data.</td>
</tr>
<tr>
<td>Series.tz_convert(tz[, axis, level, copy])</td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td>Series.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
</tbody>
</table>

**pandas.Series.asfreq**

Series.asfreq(freq=None, method=None, how=None, normalize=False)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**

- freq : DateOffset object, or string
  
  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

- how : ‘start’, ‘end’, default end
  
  For PeriodIndex only, see PeriodIndex.asfreq

- normalize : bool, default False
  
  Whether to reset output index to midnight

**Returns** converted : type of caller
**pandas.Series.asof**

Series.asof(where)

Return last good (non-NaN) value in TimeSeries if value is NaN for requested date.

If there is no good value, NaN is returned.

**Parameters** where : date or array of dates

**Returns** value or NaN

**Notes**

Dates are assumed to be sorted

**pandas.Series.shift**

Series.shift(periods=1, freq=None, axis=0, **kwargs)

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 datetools 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.first_valid_index**

Series.first_valid_index()

Return label for first non-NA/null value

**pandas.Series.last_valid_index**

Series.last_valid_index()

Return label for last non-NA/null value

**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)

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** rule : string
the offset string or object representing target conversion

**how** : string
   method for down- or re-sampling, default to ‘mean’ for downsampling

**axis** : int, optional, default 0

**fill_method** : string, default None
   fill_method for upsampling

**closed** : {'right', 'left'}
   Which side of bin interval is closed

**label** : {'right', 'left'}
   Which bin edge label to label bucket with

**convention** : {'start', 'end', 's', 'e'}

**kind** : “period”/”timestamp”

**offset** : timedelta
   Adjust the resampled time labels

**limit** : int, default None
   Maximum size gap to when reindexing with fill_method

**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

**pandas.Series.tz_convert**

Series.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 is 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(*args, **kwargs)
   Localize tz-naive TimeSeries to target time zone

33.3. Series
Parameters  
\texttt{tz} : string or pytz.timezone object 
\texttt{axis} : the axis to localize 
\texttt{level} : int, str, default None 
If axis is a MultiIndex, localize a specific level. Otherwise must be None 
\texttt{copy} : boolean, default True 
Also make a copy of the underlying data 
\texttt{ambiguous} : ‘infer’, bool-ndarray, ‘NaT’, default ‘raise’ 
\begin{itemize}
  \item ‘infer’ will attempt to infer fall dst-transition hours based on order 
  \item bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times) 
  \item ‘NaT’ will return NaT where there are ambiguous times 
  \item ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times 
\end{itemize}
\texttt{infer\_dst} : boolean, default False (DEPRECATED) 
Attempt to infer fall dst-transition hours based on order 

\textbf{Raises}  
\textbf{TypeError} 
If the TimeSeries is tz-aware and tz is not None.

### 33.3.13 Datetimelike Properties

\texttt{Series.dt} can be used to access the values of the series as datetimelike and return several properties. These can be accessed like \texttt{Series.dt.<property>}. 

\textbf{Datet ime Properties} 

| \texttt{Series.dt.date} | Returns numpy array of datetime.date. |
| \texttt{Series.dt.time} | Returns numpy array of datetime.time. |
| \texttt{Series.dt.year} | The year of the datetime |
| \texttt{Series.dt.month} | The month as January=1, December=12 |
| \texttt{Series.dt.day} | The days of the datetime |
| \texttt{Series.dt.hour} | The hours of the datetime |
| \texttt{Series.dt.minute} | The minutes of the datetime |
| \texttt{Series.dt.second} | The seconds of the datetime |
| \texttt{Series.dt.microsecond} | The microseconds of the datetime |
| \texttt{Series.dt.nanosecond} | The nanoseconds of the datetime |
| \texttt{Series.dt.quarter} | The quarter of the date |
| \texttt{Series.dt.is\_month\_start} | Logical indicating if first day of month (defined by frequency) |
| \texttt{Series.dt.is\_month\_end} | Logical indicating if last day of month (defined by frequency) |
| \texttt{Series.dt.is\_quarter\_start} | Logical indicating if first day of quarter (defined by frequency) |
| \texttt{Series.dt.is\_quarter\_end} | Logical indicating if last day of quarter (defined by frequency) |
| \texttt{Series.dt.is\_year\_start} | Logical indicating if first day of year (defined by frequency) |

Continued on next page
Table 33.34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.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>get/set the frequency of the Index</td>
</tr>
<tr>
<td>Series.dt.freq</td>
<td></td>
</tr>
</tbody>
</table>

**pandas.Series.dt.date**

Series.dt.date

Returns numpy array of datetime.date. The date part of the Timestamps.

**pandas.Series.dt.time**

Series.dt.time

Returns numpy array of datetime.time. The time part of the Timestamps.

**pandas.Series.dt.year**

Series.dt.year

The year of the datetime

**pandas.Series.dt.month**

Series.dt.month

The month as January=1, December=12

**pandas.Series.dt.day**

Series.dt.day

The days of the datetime

**pandas.Series.dt.hour**

Series.dt.hour

The hours of the datetime

**pandas.Series.dt.minute**

Series.dt.minute

The minutes of the datetime

**pandas.Series.dt.second**

Series.dt.second

The seconds of the datetime
pandas.Series.dt.microsecond

Series.dt.microsecond
The microseconds of the datetime

pandas.Series.dt.nanosecond

Series.dt.nanosecond
The nanoseconds of the datetime

pandas.Series.dt.second

Series.dt.second
The seconds of the datetime

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.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)

pandas.Series.dt.is_quarter_end

Series.dt.is_quarter_end
Logical indicating if last day of quarter (defined by frequency)

pandas.Series.dt.is_year_start

Series.dt.is_year_start
Logical indicating if first day of year (defined by frequency)

pandas.Series.dt.is_year_end

Series.dt.is_year_end
Logical indicating if last day of year (defined by frequency)

pandas.Series.dt.daysinmonth

Series.dt.daysinmonth
The number of days in the month
New in version 0.16.0.

pandas.Series.dt.days_in_month

Series.dt.days_in_month
The number of days in the month
New in version 0.16.0.

pandas.Series.dt.tz

Series.dt.tz
### pandas.Series.dt.freq

Series.dt.freq
get/set the frequency of the Index

#### 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 pytz/dateutil), or remove timezone from tz-aware DatetimeIndex</td>
</tr>
<tr>
<td>Series.dt.tz_convert(*args, **kwargs)</td>
<td>Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)</td>
</tr>
<tr>
<td>Series.dt.normalize(*args, **kwargs)</td>
<td>Return DatetimeIndex with times to midnight.</td>
</tr>
</tbody>
</table>

### pandas.Series.dt.to_period

Series.dt.to_period(*args, **kwargs)
Cast to PeriodIndex at a particular frequency

### pandas.Series.dt.to_pydatetime

Series.dt.to_pydatetime()

### 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

**Returns**

- **localized**: DatetimeIndex

**Raises**

- **TypeError**
  
  If the DatetimeIndex is tz-aware and tz is not None.
**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

**Raises**

- TypeError
  
  If DatetimeIndex is tz-naive.

**pandas.Series.dt.normalize**

Series.dt.normalize(*args, **kwargs)

Return DatetimeIndex with times to midnight. Length is unaltered

**Returns**

- `normalized`: DatetimeIndex

**Timedelta Properties**

- `Series.dt.days`  
  Number of days for each element.

- `Series.dt.seconds`  
  Number of seconds (>= 0 and less than 1 day) for each element.

- `Series.dt.microseconds`  
  Number of microseconds (>= 0 and less than 1 second) for each element.

- `Series.dt.nanoseconds`  
  Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

- `Series.dt.components`  
  Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds).

**pandas.Series.dt.days**

Series.dt.days

Number of days for each element.

**pandas.Series.dt.seconds**

Series.dt.seconds

Number of seconds (>= 0 and less than 1 day) for each element.

**pandas.Series.dt.microseconds**

Series.dt.microseconds

Number of microseconds (>= 0 and less than 1 second) for each element.

**pandas.Series.dt.nanoseconds**

Series.dt.nanoseconds

Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.


**pandas.Series.dt.components**

Series.dt.components

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns a DataFrame

**Timedelta Methods**

- **Series.dt.to_pytimedelta()**

**pandas.Series.dt.to_pytimedelta**

Series.dt.to_pytimedelta()

33.3.14 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><code>Series.str.capitalize()</code></td>
<td>Convert strings in the Series/Index to be capitalized.</td>
</tr>
<tr>
<td><code>Series.str.cat([others, sep, na_rep])</code></td>
<td>Concatenate strings in the Series/Index with given separator.</td>
</tr>
<tr>
<td><code>Series.str.center(width[, fillchar])</code></td>
<td>Filling left and right side of strings in the Series/Index with an additional character.</td>
</tr>
<tr>
<td><code>Series.str.contains(pat[, case, flags, na, ...])</code></td>
<td>Count occurrences of pattern in each string of the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.decode(encoding[, errors])</code></td>
<td>Decode character string in the Series/Index to unicode using indicated encoding.</td>
</tr>
<tr>
<td><code>Series.str.encode(encoding[, errors])</code></td>
<td>Encode character string in the Series/Index to some other encoding using indicated encoding.</td>
</tr>
<tr>
<td><code>Series.str.endswith(pat[, na])</code></td>
<td>Return boolean Series indicating whether each string in the Series/Index ends with passed pattern.</td>
</tr>
<tr>
<td><code>Series.str.extract(pat[, flags])</code></td>
<td>Find groups in each string in the Series using passed regular expression.</td>
</tr>
<tr>
<td><code>Series.str.find(sub[, start, end])</code></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><code>Series.str.findall(pat[, flags])</code></td>
<td>Find all occurrences of pattern or regular expression in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.get(i)</code></td>
<td>Extract element from lists, tuples, or strings in each element in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.index(sub[, start, end])</code></td>
<td>Return lowest indexes in each strings where the substring is fully contained between [start:end].</td>
</tr>
<tr>
<td><code>Series.str.join(sep)</code></td>
<td>Join lists contained as elements in the Series/Index with passed delimiter.</td>
</tr>
<tr>
<td><code>Series.str.len()</code></td>
<td>Compute length of each string in the Series/Index.</td>
</tr>
<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.lstrip([to_strip])</code></td>
<td>Convert strings in the Series/Index to lowercase.</td>
</tr>
<tr>
<td><code>Series.str.match(pat[, case, flags, na, ...])</code></td>
<td>Strip whitespace (including newlines) from each string in the Series/Index from left side.</td>
</tr>
<tr>
<td><code>Series.str.normalize(form)</code></td>
<td>Deprecated: Find groups in each string in the Series/Index using passed regular expression.</td>
</tr>
<tr>
<td><code>Series.str.pad(width[, side, fillchar])</code></td>
<td>Return the Unicode normal form for the strings in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.partition([pat, expand])</code></td>
<td>Pad strings in the Series/Index with an additional character to specified side.</td>
</tr>
<tr>
<td><code>Series.str.repeat(repeats)</code></td>
<td>Split the string at the first occurrence of sep, and return 3 elements containing the strings of the Series/Index by indicated number of times.</td>
</tr>
<tr>
<td><code>Series.str.replace(pat, repl[, n, case, flags])</code></td>
<td>Duplicate each string in the Series/Index by specified side.</td>
</tr>
<tr>
<td><code>Series.str.rfind(sub[, start, end])</code></td>
<td>Replace occurrences of pattern/regex in the Series/Index with some other string.</td>
</tr>
<tr>
<td><code>Series.str.rindex(sub[, start, end])</code></td>
<td>Return highest indexes in each strings in the Series/Index where the substring is fully contained.</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 strings of the Series/Index by specified side.</td>
</tr>
<tr>
<td><code>Series.str.rstrip([to_strip])</code></td>
<td>Slice substrings from each element in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.slice((start, stop, step))</code></td>
<td>Replace a slice of each string in the Series/Index with another string.</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>
</tbody>
</table>
### pandas.Series.str.split

`Series.str.split(*args, **kwargs)`  
Split each string (in the Series/Inx) by given pattern, propagating NA values.

`Series.str.rsplit([pat, n, expand])`  
Split each string in the Series/Inx by the given delimiter string, starting at the end and working to the front.

`Series.str.strip([to_strip])`  
Strip whitespace (including newlines) from each string in the Series/Inx from left and right sides.

`Series.str.startswith(pat[, na])`  
Return boolean Series/Array indicating whether each string in the Series/Inx starts with passed pattern.

`Series.str.strip()`  
Strip whitespace (including newlines) from each string in the Series/Inx from left and right sides.

`Series.str.swapcase()`  
Convert strings in the Series/Inx to swapcased.

`Series.str.title()`  
Convert strings in the Series/Inx to titlecase.

`Series.str.translate(table[, deletechars])`  
Map all characters in the string through the given mapping table.

`Series.str.upper()`  
Convert strings in the Series/Inx to uppercase.

`Series.str.wrap(width, **kwargs)`  
Wrap long strings in the Series/Inx to be formatted in paragraphs with length less than a given width.

`Series.str.zfill(width)`  
Zfill each string with leading zeros to a given width.

### pandas.Series.str.isalnum

`Series.str.isalnum()`  
Check whether all characters in each string in the Series/Inx are alphanumeric.

### pandas.Series.str.isalpha

`Series.str.isalpha()`  
Check whether all characters in each string in the Series/Inx are alphabetic.

### pandas.Series.str.isdigit

`Series.str.isdigit()`  
Check whether all characters in each string in the Series/Inx are digits.

### pandas.Series.str.isspace

`Series.str.isspace()`  
Check whether all characters in each string in the Series/Inx are whitespace.

### pandas.Series.str.islower

`Series.str.islower()`  
Check whether all characters in each string in the Series/Inx are lowercase.

### pandas.Series.str.isupper

`Series.str.isupper()`  
Check whether all characters in each string in the Series/Inx are uppercase.

### pandas.Series.str.istitle

`Series.str.istitle()`  
Check whether all characters in each string in the Series/Inx are titlecase.

### pandas.Series.str.isnumeric

`Series.str.isnumeric()`  
Check whether all characters in each string in the Series/Inx are numeric.

### pandas.Series.str.isdecimal

`Series.str.isdecimal()`  
Check whether all characters in each string in the Series/Inx are decimal.

### 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.

```python
>>> Series(["a", "b", "c"]).str.cat(["A", "B", "C"], sep=",")
0 a,A
1 b,B
2 c,C
```

33.3. Series

1079
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
```

**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

**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
**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

**pandas.Series.str.decode**

Series.str.decode(encoding, errors='strict')

Decode character string in the Series/Index to unicode using indicated encoding. Equivalent to str.decode().

**Parameters**

- **encoding**: string
- **errors**: string

**Returns**

- **decoded**: Series/Index of objects

**pandas.Series.str.encode**

Series.str.encode(encoding, errors='strict')

Encode character string in the Series/Index to some other encoding using indicated encoding. Equivalent to str.encode().

**Parameters**

- **encoding**: string
- **errors**: string

**Returns**

- **encoded**: Series/Index of objects

**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

**pandas.Series.str.extract**

Series.str.extract(pat, flags=0)

Find groups in each string in the Series using passed regular expression.

**Parameters**

- **pat**: string
  - Pattern or regular expression
flags : int, default 0 (no flags)
re module flags, e.g. re.IGNORECASE

Returns extracted groups : Series (one group) or DataFrame (multiple groups)
Note that dtype of the result is always object, even when no match is found and the
result is a Series or DataFrame containing only NaN values.

Examples

A pattern with one group will return a Series. Non-matches will be NaN.

```python
>>> Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)')
0   1
1   2
2   NaN
dtype: object
```

A pattern with more than one group will return a DataFrame.

```python
>>> Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
   0  1
0   a  1
1   b  2
2  NaN NaN
```

A pattern may contain optional groups.

```python
>>> Series(['a1', 'b2', 'c3']).str.extract('([ab])?\d)')
   0  1
0   a  1
1   b  2
2  NaN  3
```

Named groups will become column names in the result.

```python
>>> Series(['a1', 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)')
    letter digit
0   a       1
1   b       2
2  NaN      NaN
```

**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
Subtring 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

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

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

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
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

pandas.Series.str.len

Series.str.len()
Compute length of each string in the Series/Index.

Returns lengths : Series/Index of integer values

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.right().

Parameters width : int
Minimum width of resulting string; additional characters will be filled with fillchar

fillchar : str
Additional character for filling, default is whitespace

Returns filled : Series/Index of objects

pandas.Series.str.lower

Series.str.lower()
Convert strings in the Series/Index to lowercase. Equivalent to str.lower().

Returns converted : Series/Index of objects

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

pandas.Series.str.match

Series.str.match(pat, case=True, flags=0, na=nan, as_indexer=False)
Deprecated: Find groups in each string in the Series/Index using passed regular expression. If as_indexer=True, 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 : False, by default, gives deprecated behavior better achieved using str_extract. True return boolean indexer.

Returns Series/array of boolean values
  if as_indexer=True
Series/Index of tuples
  if as_indexer=False, default but deprecated

See also:

contains analogous, but less strict, relying on re.search instead of re.match
extract now preferred to the deprecated usage of match (as_indexer=False)

Notes

To extract matched groups, which is the deprecated behavior of match, use str.extract.

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

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
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
```

pandas.Series.str.repeat

Series.str.repeat(repeats)
Duplicate 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
**pandas.Series.str.replace**

Series.str.replace(pat, repl, n=-1, case=True, flags=0)
Replace occurrences of pattern/regex in the Series/Index with some other string. Equivalent to
str.replace() or re.sub().

**Parameters**

- **pat**: string
  Character sequence or regular expression
- **repl**: string
  Replacement sequence
- **n**: int, default -1 (all)
  Number of replacements to make from start
- **case**: boolean, default True
  If True, case sensitive
- **flags**: int, default 0 (no flags)
  re module flags, e.g. re.IGNORECASE

**Returns**

- **replaced**: Series/Index of objects

**pandas.Series.str.rfind**

Series.str.rfind(sub, start=0, end=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 str.rfind().

**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:

- find Return lowest indexes in each strings

**pandas.Series.str.rindex**

Series.str.rindex(sub, start=0, end=None)
Return highest indexes in each strings where the substring is fully contained between [start:end]. This is the same as str.rfind except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard str.rindex.

**Parameters**

- **sub**: str
  Substring being searched
pandas: powerful Python data analysis toolkit, Release 0.16.2

start : int
Left edge index

end : int
Right edge index

Returns found : Series/Index of objects

See also:

index Return lowest indexes in each strings

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.left().

Parameters width : int
Minimum width of resulting string; additional characters will be filled with fillchar

fillchar : str
Additional character for filling, default is whitespace

Returns filled : Series/Index of objects

pandas.Series.str.rpartition

Series.str.rpartition(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

**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

**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

**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

**pandas.Series.str.split**

Series.str\[split\](*args, **kwargs)

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.
    New in version 0.16.1.
return_type : deprecated, use expand

Returns split : Series/Index or DataFrame/MultiIndex of objects

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().
    New in version 0.16.2.

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

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

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
**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

**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

**pandas.Series.str.translate**

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

**pandas.Series.str.upper**

Series.str.upper()

Convert strings in the Series/Index to uppercase. Equivalent to str.upper().

Returns converted : Series/Index of objects

**pandas.Series.str.wrap**

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 com-
pound words, as it is customary in English. If false, only whitespaces will be consid-
ered as potentially good places for line breaks, but you need to set break_long_words
to false if you want truly insecable 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

>>> s = pd.Series(['line to be wrapped', 'another line to be wrapped'])
>>> s.str.wrap(12)
0  line to be\nwrapped
1  another line\n4wrapped

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
**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

**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

**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

**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

**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

**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

**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
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

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

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()
da  b  c
0  1  1  0
1  1  0  0
2  1  0  1

>>> Series(['a|b', np.nan, 'a|c']).str.get_dummies()
da  b  c
0  1  1  0
1  0  0  0
2  1  0  1
```

33.3.15 Categorical

If the Series is of dtype category, Series.cat can be used to change the 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>Series.cat.categories</th>
<th>The categories of this categorical.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.cat.ordered</td>
<td>Gets the ordered attribute</td>
</tr>
<tr>
<td>Series.cat.codes</td>
<td></td>
</tr>
</tbody>
</table>
**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 a inplace operation!

Raises ValueError

If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

rename_categories, reorder_categories, add_categories, remove_categories, remove_unused_categories, set_categories

**pandas.Series.cat.ordered**

Series.cat.ordered

Gets the ordered attribute

**pandas.Series.cat.codes**

Series.cat.codes

**pandas.Series.cat.rename_categories**

Series.cat.rename_categories(*args, **kwargs)

Renames categories.

The new categories 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.

Parameters new_categories : Index-like

The renamed categories.

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 with renamed categories added or None if inplace.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename_categories(*args, **kwargs)</td>
<td>Renames categories.</td>
</tr>
<tr>
<td>reorder_categories(*args, **kwargs)</td>
<td>Reorders categories as specified in new_categories.</td>
</tr>
<tr>
<td>add_categories(*args, **kwargs)</td>
<td>Add new categories.</td>
</tr>
<tr>
<td>remove_categories(*args, **kwargs)</td>
<td>Removes the specified categories.</td>
</tr>
<tr>
<td>remove_unused_categories(*args, ...)</td>
<td>Removes categories which are not used.</td>
</tr>
<tr>
<td>set_categories(*args, ...)</td>
<td>Sets the categories to the specified new_categories.</td>
</tr>
<tr>
<td>as_ordered(*args, **kwargs)</td>
<td>Sets the Categorical to be ordered</td>
</tr>
<tr>
<td>as_unordered(*args, **kwargs)</td>
<td>Sets the Categorical to be unordered</td>
</tr>
</tbody>
</table>

33.3. Series
**pandas.Series.cat.reorder_categories**

Series.cat.reorder_categories(*args, **kwargs)

Reorders categories as specified in new_categories.

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

**pandas.Series.cat.add_categories**

Series.cat.add_categories(*args, **kwargs)

Add new categories.

Parameters:
- **new_categories** : category or list-like of category
  The new categories to be included.
- **inplace** : 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, add_categories, remove_categories, remove_unused_categories, set_categories
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, remove_categories, remove_unused_categories, set_categories

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

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 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

### 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

### 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:

**Categorical(values, categories, ordered, ...)**: Represents a categorical variable in classic R / S-plus fashion
pandas.Categorical

class pandas.Categorical(values, categories=None, ordered=False, name=None, fastpath=False, levels=None)

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.

name : str, optional

Name for the Categorical variable. If name is None, will attempt to infer from values.

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.

Examples

```python
>>> from pandas import Categorical
>>> Categorical([1, 2, 3, 1, 2, 3])
[1, 2, 3, 1, 2, 3]
Categories (3, int64): [1 < 2 < 3]
```

```python
>>> Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
[a, b, c, a, b, c]
Categories (3, object): [a < b < c]
```

```python
>>> a = Categorical(['a','b','c','a','b','c'], ['c', 'b', 'a'])
>>> a.min()
'c'
```

Attributes

<table>
<thead>
<tr>
<th>categories</th>
<th>The categories of this categorical.</th>
</tr>
</thead>
</table>

Continued on next page
Table 33.42 – continued from previous page

<table>
<thead>
<tr>
<th>codes</th>
<th>The category codes of this categorical.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>Gets the ordered attribute</td>
</tr>
</tbody>
</table>

| name          | (string) The name of this Categorical. |

Methods

- add_categories(new_categories[, inplace])
- argsort([ascending])
- as_ordered([inplace])
- as_unordered([inplace])
- astype(dtype)
- check_for_ordered(op)
- copy()
- describe()
- dropna()
- equals(other)
- fillna(*args, **kwargs)
- from_array(data, **kwargs)
- from_codes(codes, categories[, ordered, name])
- get_values()
- is_dtype_equal(other)
- isnan()
- max([numeric_only])
- min([numeric_only])
- mode()
- notnull()
- order([inplace, ascending, na_position])
- ravel([order])
- remove_categories(removals[, inplace])
- remove_unused_categories([inplace])
- rename_categories(new_categories[, inplace])
- reorder_categories(new_categories[, ...])
- repeat(repeats)
- reshape(new_shape, **kwargs)
- searchsorted(v[, side, sorter])
- set_categories(new_categories[, ordered, ...])
- set_ordered(value[, inplace])
- sort([inplace, ascending, na_position])
- take(indexer[, allow_fill, fill_value])
- take_nd(indexer[, allow_fill, fill_value])
- to_dense()
- unique()
- value_counts([dropna])
- view()

pandas.Categorical.from_codes

```
classmethod Categorical.from_codes(codes, categories, ordered=False, name=None)

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.

- **name**: str, optional
  - Name for the Categorical variable.

Categorical.from_codes(codes, categories[, ...]) Make a Categorical type from codes and categories arrays.

**pandas.Categorical.from_codes**

*classmethod*

Categorical.from_codes(codes, categories, ordered=False, name=None)

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.

- **name**: str, optional
  - Name for the Categorical variable.

np.asarray(categorical) works by implementing the array interface. Be aware, that this converts the Categorical back to a numpy array, so levels and order information is not preserved!

Categorical.__array__((dtype)) The numpy array interface.

**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

### 33.3.16 Plotting

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.hist()</code></td>
<td>Draw histogram of the input series using matplotlib</td>
</tr>
<tr>
<td><code>Series.plot()</code></td>
<td>Make plots of Series using matplotlib / pylab.</td>
</tr>
</tbody>
</table>

#### pandas.Series.hist

`Series.hist(by=None, ax=None, grid=True, xlabels, ...)`  
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
- **xlabels** : int, default None  
  If specified changes the x-axis label size
- **xrot** : float, default None  
  Rotation of x axis labels
- **ylabels** : 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.plot

Series.plot (data, kind='line', ax=None, figsize=None, use_index=True, title=None, grid=None, legend=False, 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, label=None, secondary_y=False, **kwds)

Make plots of Series using matplotlib / pylab.

**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
  Title to use for the plot

- **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
  33.3. Series 1103
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)

**layout**: tuple (optional)

(rows, columns) for the layout of the plot

**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)

33.3.17 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Series.from_csv(path[, sep, parse_dates, ...])</td>
<td>Read delimited file into Series</td>
</tr>
<tr>
<td>pandas.Series.to_pickle(path)</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>pandas.Series.to_csv(path[, index, sep, na_rep, ...])</td>
<td>Convert Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>pandas.Series.to_dict()</td>
<td>Convert Series to {label -&gt; value} dict</td>
</tr>
<tr>
<td>pandas.Series.to_frame([name])</td>
<td>Convert Series to DataFrame</td>
</tr>
<tr>
<td>pandas.Series.to_hdf(path_or_buf, key, **kwargs)</td>
<td>activate the HDFStore</td>
</tr>
<tr>
<td>pandas.Series.to_sql(name, con[, flavor, schema, ...])</td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td>pandas.Series.to_msgpack([path_or_buf])</td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td>pandas.Series.to_json([path_or_buf])</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>pandas.Series.to_sparse([kind, fill_value])</td>
<td>Convert Series to SparseSeries</td>
</tr>
<tr>
<td>pandas.Series.to_dense()</td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td>pandas.Series.to_string([buf, na_rep, ...])</td>
<td>Attempt to write text representation of object to the system clipboard This can be</td>
</tr>
</tbody>
</table>

pandas.Series.from_csv

classmethod pandas.Series.from_csv(path, sep=';', parse_dates=True, header=None, index_col=0, encoding=None, infer_datetime_format=False)

Read delimited file into Series

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 0  
Row to use at 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

**pandas.Series.to_pickle**

Series.$\text{to\_pickle}$ ($path$)

Pickle (serialize) object to input file path

**Parameters**

- **path**: string
  - File path

**pandas.Series.to_csv**

Series.$\text{to\_csv}$ ($path$, $index=True$, $sep=\',\$', $na\_rep='\$'$

Write Series to a comma-separated values (csv) file

**Parameters**

- **path**: string file path or file handle / StringIO. 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_dict

Series.to_dict()
Convert Series to {label -> value} dict

Returns value_dict : dict

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)
activate the HDFStore

Parameters path_or_buf : the path (string) or buffer to put the store

key : string
identifier for the group in the store

mode : optional, {'a', 'w', 'r', 'r+'}, default 'a'
‘r’ Read-only; no data can be modified.
‘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

complevel : int, 1-9, default 0
If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None
If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False
If applying compression use the fletcher32 checksum

**pandas.Series.to_sql**

```python
Series.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'
  The flavor of SQL to use. Ignored when using SQLAlchemy engine. ‘mysql’ is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.

- **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_msgpack**

```python
Series.to_msgpack(path_or_buf=None, **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.Series.to_json**

Series.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

- **date_format** : {'epoch', ‘iso’}

  Type of date conversion. epoch = epoch milliseconds, iso’ = ISO8601, 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 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.

**Returns** same type as input object with filtered info axis

### 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_dense

Series.to_dense()

Return dense representation of NDFrame (as opposed to sparse)

### pandas.Series.to_string

Series.to_string( *buf*=None, *na_rep*='NaN', *float_format*=None, *header*=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)
- **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_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 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

### 33.3.18 Sparse methods

- **SparseSeries.to_coo**
  - Creates a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.

- **SparseSeries.from_coo**
  - Creates a SparseSeries from a scipy.sparse.coo_matrix.

**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).

New in version 0.16.0.

- **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**
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)]

pandas.SparseSeries.from_coo

classmethod SparseSeries.from_coo (A, dense_index=False)
Create a SparseSeries from a scipy.sparse.coo_matrix.

New in version 0.16.0.

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

from scipy import sparse
A = sparse.coo_matrix(((3.0, 1.0, 2.0), ([1, 0, 0], [0, 2, 3])),
                      shape=(3, 4))

A
<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)

33.4 DataFrame

33.4.1 Constructor

```
DataFrame(data, index, columns, dtype, copy)
```

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
- **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, otherwise 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_csv` from CSV files
- `DataFrame.from_items` from sequence of (key, value) pairs

pandas.read_csv, pandas.read_table, pandas.read_clipboard
Examples

```python
>>> d = {'col1': ts1, 'col2': ts2}
>>> df = DataFrame(data=d, index=index)
>>> df2 = DataFrame(np.random.randn(10, 5))
>>> df3 = DataFrame(np.random.randn(10, 5),
...                 columns=['a', 'b', 'c', 'd', 'e'])
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td>at</td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td>axes</td>
<td></td>
</tr>
<tr>
<td>blocks</td>
<td>Internal property, property synonym for as_blocks()</td>
</tr>
<tr>
<td>dtypes</td>
<td>Return the dtypes in this object</td>
</tr>
<tr>
<td>empty</td>
<td>True if NDFrame is entirely empty [no items]</td>
</tr>
<tr>
<td>ftypes</td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<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>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></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.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

**pandas.DataFrame.blocks**

DataFrame.blocks

Internal property, property synonym for as_blocks()
**pandas.DataFrame.dtypes**

DataFrame.dtypes
return the dtypes in this object

**pandas.DataFrame.empty**

DataFrame.empty
True if NDFrame is entirely empty [no items]

**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.
.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.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 hierachical 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.

.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

pandas.DataFrame.size

DataFrame.size

number of elements in the NDFrame

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 upcase to int32.

Methods

abs()  Return an object with absolute value taken.
add(other[, axis, level, fill_value])  Addition of dataframe 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
align(other[, join, axis, level, copy, ...])  Align two object 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(other[, ignore_index, verify_integrity])  Append rows of other to the end of this frame, returning a new object.
apply(func[, axis, broadcast, raw, reduce, args])  Applies function along input axis of DataFrame.
applymap(func)  Apply a function to a DataFrame that is intended to operate elementwise, i.e.
as_blocks()  as_matrix([columns])  asfreq(freq[, method, how, normalize])  asof()  assign(**kwargs)  assign_new_columns([columns])  astype(dtype[, copy, raise_on_error])  at_time(time[, asof])  between_time(start_time, end_time[, ...])  bfill([axis, inplace, limit, downcast])  bool()  boxplot([column, by, ax, fontsize, rot, ...])  clip([lower, upper, out, axis])  clip_lower(threshold[, axis])  clip_upper(threshold[, axis])  combine(other, func[, fill_value, overwrite])  combineAdd(other)  combineMul(other)  combine_first(other)  compound([axis, skipna, level])  consolidate(inplace)  convert_objects([convert_dates, ...])  copy([deep])  corr([method, min_periods])  corrwith(other[, axis, drop])  count([axis, level, numeric_only])  cov([min_periods])  cummax([axis, dtype, out, skipna])  cummin([axis, dtype, out, skipna])  cumprod([axis, dtype, out, skipna])  cumsum([axis, dtype, out, skipna])  

Continued on next page
describe([percentile_width, percentiles, ...])

generate various summary statistics, excluding NaN values.
1st discrete difference of object
Floating division of dataframe and other, element-wise (binary operator true_div)
Floating division of dataframe and other, element-wise (binary operator true_div)
Matrix multiplication with DataFrame or Series objects
Return new object with labels in requested axis removed
Return DataFrame with duplicate rows removed, optionally only
Return object with labels on given axis omitted where alternately any
Return boolean Series denoting duplicate rows, optionally only
Wrapper for flexible comparison methods eq
Determines if two NDFrame objects contain the same elements.
Evaluate an expression in the context of the calling DataFrame instance.
Synonym for NDFrame.fillna(method='ffill')
Fill NA/NaN values using the specified method
Restrict the info axis to set of items or wildcard
Convenience method for subsetting initial periods of time series data
Return label for first non-NA/null value
Integer division of dataframe and other, element-wise (binary operator floor_div)
Read delimited file into DataFrame
Construct DataFrame from dict of array-like or dicts
Convert (key, value) pairs to DataFrame.
Convert structured or record ndarray to DataFrame
Wrapper for flexible comparison methods ge
Get item from object for given key (DataFrame column, Panel slice, etc.).
Return the counts of dtypes in this object
Return the counts of ftypes in this object
Quickly retrieve single value at passed column and index
same as values (but handles sparseness conversions)
Group series using mapper (dict or key function, apply given function
Wrapper for flexible comparison methods gt
Returns first n rows
Draw histogram of the DataFrame’s series using matplotlib / pylab.
Return index of first occurrence of maximum over requested axis.
Return index of first occurrence of minimum over requested axis.
Concise summary of a DataFrame.
Insert column into DataFrame at specified location.
Interpolate values according to different methods.
Return boolean DataFrame showing whether each element in the DataFrame is
Return a boolean same-sized object indicating if the values are null
Iterator over (column, series) pairs
iterator alias used to get around 2to3. Deprecated
Iterator over rows of DataFrame as (index, Series) pairs.
Iterator over rows of DataFrame as tuples, with index value
Join columns with other DataFrame either on index or on a key column.
Get the ‘info’ axis (see Indexing for more)
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Convenience method for subsetting final periods of time series data
Return label for last non-NA/null value
### DataFrame

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>le(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods le</td>
</tr>
<tr>
<td><code>load(path)</code></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><code>lookup(row_labels, col_labels)</code></td>
<td>Label-based “fancy indexing” function for DataFrame.</td>
</tr>
<tr>
<td><code>lt(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods 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>mask(cond[, other, inplace, axis, level, ...])</code></td>
<td>Return an object of same shape as self and whose corresponding entries are false.</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>memory_usage([index])</code></td>
<td>Memory usage of DataFrame columns.</td>
</tr>
<tr>
<td><code>merge(right[, how, on, left_on, right_on, ...])</code></td>
<td>Merge DataFrame objects by performing a database-style join operation by column labels.</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>mode([axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td><code>mul([other, axis, level, fill_value])</code></td>
<td>Gets the mode(s) of each element along the axis selected.</td>
</tr>
<tr>
<td><code>multiply(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>ne(other[, axis, level])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Wrapper for flexible comparison methods ne</td>
</tr>
<tr>
<td><code>pct_change([periods, fill_method, limit, freq])</code></td>
<td>Return a boolean same-sized object indicating if the values are 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(data[, values, index, columns, ...])</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
</tr>
<tr>
<td><code>plot(data[, x, y, kind, ax, subplots, ...])</code></td>
<td>Make plots of DataFrame using matplotlib / pylab.</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])</code></td>
<td>Return values at the given quantile over requested axis, a la numpy.percentile.</td>
</tr>
<tr>
<td><code>query(expr, **kwargs)</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>rand([axis, numeric_only, method, ...])</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(index[, columns])</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([index, columns])</code></td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><code>rename_axis(mapper[, axis, copy, inplace])</code></td>
<td>Alter index and / or columns using input function or functions.</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 regular time-series data.</td>
</tr>
<tr>
<td><code>reset_index([level[, drop, inplace, ...]])</code></td>
<td>For DataFrame with multi-level index, return new DataFrame with labeling info.</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>Exponential power of dataframe and other, element-wise (binary operator rmod).</td>
</tr>
<tr>
<td><code>rpow(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator rsub).</td>
</tr>
<tr>
<td><code>rsub(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe 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>
<tr>
<td><code>save(path)</code></td>
<td>Deprecated.</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 dtype.</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>
</tbody>
</table>
**pandas: powerful Python data analysis toolkit, Release 0.16.2**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_axis</code></td>
<td>Set public version of axis assignment</td>
</tr>
<tr>
<td><code>set_index</code></td>
<td>Set the DataFrame index (row labels) using one or more existing columns.</td>
</tr>
<tr>
<td><code>set_value</code></td>
<td>Put single value at passed column and index</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>Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td><code>sort_index</code></td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td><code>sortlevel</code></td>
<td>Sort multilevel index by chosen axis and primary level.</td>
</tr>
<tr>
<td><code>squeeze</code></td>
<td>Squeeze length 1 dimensions</td>
</tr>
<tr>
<td><code>transpose</code></td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame</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 DataFrame to dictionary.</td>
</tr>
<tr>
<td><code>to_excel</code></td>
<td>Write DataFrame to a excel sheet.</td>
</tr>
<tr>
<td><code>to_gbq</code></td>
<td>Write a DataFrame to a Google BigQuery table.</td>
</tr>
<tr>
<td><code>to_hdf</code></td>
<td>Write DataFrame to a comma-separated values (csv) file</td>
</tr>
<tr>
<td><code>to_csv</code></td>
<td>Write DataFrame to a HDFStore.</td>
</tr>
<tr>
<td><code>to_msgpack</code></td>
<td>Render a DataFrame to a tabular environment table.</td>
</tr>
<tr>
<td><code>to_json</code></td>
<td>Render a DataFrame to a console-friendly tabular output.</td>
</tr>
<tr>
<td><code>to_period</code></td>
<td>Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.</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 DataFrame from DatetimeIndex to PeriodIndex with desired Pickle</td>
</tr>
<tr>
<td><code>to_timestamp</code></td>
<td>Convert DataFrame from DatetimeIndex to PeriodIndex with desired Pickle</td>
</tr>
<tr>
<td><code>to_clipboard</code></td>
<td>Render a DataFrame as an HTML table.</td>
</tr>
<tr>
<td><code>to_html</code></td>
<td>Render the object to the system clipboard This can be pasted into Excel,</td>
</tr>
<tr>
<td></td>
<td>for example. This method is equivalent to Excel's copy (Ctrl+C) method.</td>
</tr>
<tr>
<td><code>to_latex</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>to_stata</code></td>
<td>Render a DataFrame to a console-friendly tabular output.</td>
</tr>
<tr>
<td><code>to_sql</code></td>
<td>Cast to DatetimeIndex of timestamps, at <code>beginning</code> of period</td>
</tr>
<tr>
<td><code>to_panel</code></td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td><code>true_div</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator)</td>
</tr>
<tr>
<td><code>trunc</code></td>
<td>Truncates a sorted NDFrame before and/or after some particular dates.</td>
</tr>
<tr>
<td><code>tz_convert</code></td>
<td>Shift the time index, using the index’s frequency if available</td>
</tr>
<tr>
<td><code>tz_localize</code></td>
<td>Localize tz-naive TimeSeries to target time zone</td>
</tr>
<tr>
<td><code>update</code></td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame</td>
</tr>
<tr>
<td><code>var</code></td>
<td>Return unbiased variance over requested axis</td>
</tr>
<tr>
<td><code>xs</code></td>
<td>Return an object of same shape as self and whose corresponding entries are</td>
</tr>
<tr>
<td></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

33.4. DataFrame
pandas.DataFrame.align

DataFrame.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)
Align two object on their axes with the specified join method for each axis Index

Parameters
other : DataFrame or Series
    Align on index (0), columns (1), or both (None)
join : {'outer', 'inner', 'left', 'right'}, default 'outer'
    Allowed axis of the other object, default None
axis : allowed axis of the other object, default 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
    Filling axis, method and limit
limit : int, default None
fill_axis : {0, 1}, default 0
    Returns (left, right) : (type of input, 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 data. If None, will attempt to use everything, then use only boolean data

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 data. If None, will attempt to use everything, then use only boolean data

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.

Examples

>>> 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
0  5  6
1  7  8

With `ignore_index` set to True:

>>> df.append(df2, ignore_index=True)

A  B
0  1  2
1  3  4
2  5  6
3  7  8

**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
- **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 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.
- **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
- **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

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

`pandas.DataFrame.as_blocks`

`DataFrame.as_blocks()`

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

**NOTE:** the dtypes of the blocks WILL BE PRESERVED HERE (unlike in `as_matrix`)

**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.

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)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters freq : DateOffset object, or string

method : {'backfill', 'bfill', 'pad', 'ffill', 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 method

how : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

normalize : bool, default False

Whether to reset output index to midnight

Returns converted : type of caller

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.

New in version 0.16.0.

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. 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

Since `kwargs` is a dictionary, the order of your arguments may not be preserved. The make things predictable, 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))
```

A B  ln_A
0 1 0.426905 0.000000
1 2 -0.780949 0.693147
2 3 -0.418711 1.098612
3 4 -0.269708 1.386294
4 5 -0.274002 1.609438
5 6 -0.500792 1.791759
6 7 1.649697 1.945910
7 8 -1.495604 2.079442
8 9 0.549296 2.197225
9 10 -0.758542 2.302585
```

Where the value already exists and is inserted:

```python
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
```

A B  ln_A
0 1 0.426905 0.000000
1 2 -0.780949 0.693147
2 3 -0.418711 1.098612
3 4 -0.269708 1.386294
4 5 -0.274002 1.609438
5 6 -0.500792 1.791759
6 7 1.649697 1.945910
7 8 -1.495604 2.079442
8 9 0.549296 2.197225
9 10 -0.758542 2.302585
```

**pandas.DataFrame.astype**

DataFrame.astype(*dtype*, *copy=True*, *raise_on_error=True*, **kwargs)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

**Parameters**

- *dtype*: numpy.dtype or Python type
- *raise_on_error*: raise on invalid input
- *kwargs*: keyword arguments to pass on to the constructor

**Returns**

- *casted*: type of caller
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 NDFrame.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 : {‘axes’, ‘dict’, ‘both’}, default ‘dict’
    The kind of object to return. ‘dict’ returns a dictionary whose values are the
    matplotlib Lines of the boxplot; ‘axes’ returns the matplotlib axes the boxplot is
drawn on; ‘both’ returns a namedtuple with the axes and dict.

    When grouping with by, a dict mapping columns to return_type is returned.

kwds : other plotting keyword arguments to be passed to matplotlib boxplot
    function

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=lower, upper=upper, out=out, axis=axis)
    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.

Returns clipped : Series

Examples

>>> 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
```python
4  0.500000  0.500000
>>> t
0 -0.3
1 -0.2
2 -0.1
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.DataFrame.clip_lower**

DataFrame.clip_lower(threshold, axis=None)

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.

**Returns**
- **clipped**: same type as input

See also:

clip

**pandas.DataFrame.clip_upper**

DataFrame.clip_upper(threshold, axis=None)

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.

**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.combineAdd**  

DataFrame.**combineAdd**(other)  
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  

**Returns**  
DataFrame  

---  

**pandas.DataFrame.combineMult**  

DataFrame.**combineMult**(other)  
Multiply 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  

**Returns**  
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, 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 data. If None, will attempt to use everything, then
   use only numeric data

Returns compounded : Series or DataFrame (if level specified)

pandas.DataFrame.consolidate

DataFrame.consolidate(inplace=False)
   Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndar-
   ray). Mainly an internal API function, but available here to the savvy user

   Parameters inplace : boolean, default False
      If False return new object, otherwise modify existing object

   Returns consolidated : type of caller

pandas.DataFrame.convert_objects

DataFrame.convert_objects(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)
   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

pandas.DataFrame.copy

DataFrame.copy(deep=True)
   Make a copy of this object

   Parameters deep : boolean or string, default True
Make a deep copy, i.e. also copy 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, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative max over requested axis.

Parameters

axis : {index (0), columns (1)}
skipna : boolean, default True
Excluding NA/null values. If an entire row/column is NA, the result will be NA.

Returns

max : Series

pandas.DataFrame.cummin

DataFrame.cummin(axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative min over requested axis.

Parameters

axis : {index (0), columns (1)}
skipna : boolean, default True
Excluding NA/null values. If an entire row/column is NA, the result will be NA.

Returns

min : Series

pandas.DataFrame.cumprod

DataFrame.cumprod(axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative prod over requested axis.

Parameters

axis : {index (0), columns (1)}
skipna : boolean, default True
Excluding NA/null values. If an entire row/column is NA, the result will be NA.

Returns

prod : Series
pandas.DataFrame.cumsum

DataFrame.cumsum(axis=None, dtype=None, out=None, skipna=True, **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
sum : Series

pandas.DataFrame.describe

DataFrame.describe(percentile_width=None, percentiles=None, include=None, exclude=None)
Generate various summary statistics, excluding NaN values.

Parameters
percentile_width : float, deprecated
   The percentile_width argument will be removed in a future version. Use percentiles instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75
percentiles : array-like, optional
   The percentiles to include in the output. Should all be in the interval [0, 1]. By default percentiles is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.
include, exclude : list-like, ‘all’, or None (default)
   Specify the form of the returned result. Either:
   • None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
   • A list of dtypes or strings to be included/excluded. To select all numeric types use numpy numpy.number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
   • If include is the string ‘all’, the output column-set will match the input one.

Returns
summary: NDFrame of summary statistics

See also:
DataFrame.select_dtypes

Notes

The output DataFrame index depends on the requested dtypes:
For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.
For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.
For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.
If multiple values have the highest count, then the count and most common pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.

**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 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.divide**

DataFrame.divide(*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.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, axis=0, level=None, inplace=False, errors='raise')`  
Return new object with labels in requested axis removed  

**Parameters**  
labels : single label or list-like  
axis : int or axis name  
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. New in version 0.16.1.

**Returns**  
dropped : type of caller
**pandas.DataFrame.drop_duplicates**

DataFrame.drop_duplicates(*args, **kwargs)

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
- **take_last**: boolean, default False
  - Take the last observed row in a row. Defaults to the first row
- **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 inplace and return None.

**Returns**
- **dropped**: DataFrame

**pandas.DataFrame.duplicated**

DataFrame.duplicated(*args, **kwargs)

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
- **take_last**: boolean, default False
  - Take the last observed row in a row. Defaults to the first row

**Returns**
- **duplicated**: DataFrame
For a set of distinct duplicate rows, flag all but the last row as duplicated. Default is for all but the first row to be flagged.

**cols** : kwargs only argument of subset [deprecated]

**Returns duplicated** : Series

**pandas.DataFrame.eq**

DataFrame.eq(*other*, axis='columns', level=None)
   Wrapper for flexible comparison methods eq

**pandas.DataFrame.equals**

DataFrame.equals(*other*)
   Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

**pandas.DataFrame.eval**

DataFrame.eval(*expr*, **kwargs)
   Evaluate an expression in the context of the calling DataFrame instance.

   **Parameters expr** : string
       The expression string to evaluate.

   **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.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.ffill

**DataFrame.ffill** *(axis=None, inplace=False, limit=None, downcast=None)*

| Synonym for NDFrame.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 / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap
- **axis** : {0, 1, ‘index’, ‘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.
- **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

pandas.DataFrame.filter

**DataFrame.filter** *(items=None, like=None, regex=None, axis=None)*

Restrict the info axis to set of items or wildcard

**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 None

The axis to filter on. By default this is the info axis. The “info axis” is the axis that is used when indexing with []. For example, df = DataFrame({'a': [1, 2, 3, 4]}); df['a']. So, the DataFrame columns are the info axis.

**Notes**

Arguments are mutually exclusive, but this is not checked for

**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.last('10D') -> First 10 days
```

**pandas.DataFrame.first_valid_index**

Dataframe.**first_valid_index**()

Return label for first non-NA/null value

**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

**class method**  
DataFrame.from_csv(*path*,  
** header=0,  
** sep=',',  
** index_col=0,  
** parse_dates=True,  
** encoding=None,  
** tupleize_cols=False,  
** infer_datetime_format=False *)

Read delimited file into DataFrame

**Parameters**  
path : string file path or file handle / StringIO

header : int, default 0

Row to use at 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

**Notes**

Preferable to use read_table for most general purposes but from_csv makes for an easy roundtrip to and from file, especially with a DataFrame of time series data
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_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 to 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.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

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)
    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 / list of functions, dict, Series, or tuple / list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups
- **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
- **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()
```

```python
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```python
>>> 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)
Returns first n rows

pandas.DataFrame.hist

DataFrame.hist(data, column=None, by=None, grid=True, xlabels=column, yrot=None, xrot=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

xlabels : int, default None
    If specified changes the x-axis label size

xrot : float, default None
    rotation of x axis labels

ylabels : 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: (optional) a tuple (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

```python
pandas.DataFrame.icol
```

```python
DataFrame.icol(i)
```

```python
pandas.DataFrame.idxmax
```

```python
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`.

```python
pandas.DataFrame.idxmin
```

```python
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`.

```python
pandas.DataFrame.iget_value
```

```python
DataFrame.iget_value(i, j)
```
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, 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. 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.

If allow_duplicates is False, raises Exception if column is already contained in the DataFrame.

Parameters loc : int
Must have 0 <= loc <= len(columns)

column : object

column : object

column : object

value : int, Series, or array-like

pandas.DataFrame.interpolate

DataFrame.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast=None, **kwargs)
Interpolate values according to different methods.

Parameters method : {'linear', 'time', 'index', 'values', 'nearest', 'zero',
'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline'
'piecewise_polynomial', 'pchip'}

• ‘linear’: ignore the index and treat the values as equally spaced. 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
  is passed to scipy.interpolate.interp1d with the order given both ‘poly-
  nomial’ and ‘spline’ require that you also specify and order (int) e.g.
  df.interpolate(method='polynomial', order=4)
• ‘krogh’, ‘piecewise_polynomial’, ‘spline’, and ‘pchip’ are all
  wrappers around the scipy interpolation methods of similar
  names. See the scipy documentation for more on their behavior:
  http://docs.scipy.org/doc/scipy/reference/interpolate.html#univari-
  ate-interpolation http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html

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.

inplace : bool, default False
  Update the DataFrame in place if possible.

downcast : optional, ‘infer’ or None, defaults to None
  Downcast dtypes if possible.

Returns  Series or DataFrame of same shape interpolated at the NaNs

See also:
  reindex, replace,fillna

Examples

Filling in NaNs

>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0 0
1 1
2 2
3 3
dtype: float64

pandas.DataFrame.irow

DataFrame.irow(i, copy=False)

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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 # Column A in 'other' has a 3, but not at index 1.
 2  True  True
```

**pandas.DataFrame.isnull**

DataFrame.isnull()

Return a boolean same-sized object indicating if the values are null

**See also:**

notnull boolean inverse of isnull

**pandas.DataFrame.iteritems**

DataFrame.iteritems()

Iterator over (column, series) pairs

**pandas.DataFrame.iterkv**

DataFrame.iterkv(*args, **kwargs)

iteritems alias used to get around 2to3. Deprecated
pandas.DataFrame.iterrows

DataFrame.iterrows()
Iterate over rows of DataFrame as (index, Series) pairs.

Returns it : generator
A generator that iterates over the rows of the frame.

Notes

• iterrows does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```python
gf = DataFrame([[1, 1.0]], columns=['x', 'y'])
row = next(df.iterrows())[1]
print(row['x'].dtype)
float64
print(df['x'].dtype)
int64
```

pandas.DataFrame.itertuples

DataFrame.itertuples(index=True)
Iterate over rows of DataFrame as tuples, with index value as first element of the tuple

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) to use for joining, otherwise join 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'}
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

e suffix : string
Suffix to use from right frame’s overlapping columns

sort : boolean, default False
Order result DataFrame lexicographically by the join key. If False, preserves the
index order of the calling (left) DataFrame

Returns joined : DataFrame

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

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.

pandas.DataFrame.kurt

DataFrame.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fishers 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, 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 data. If None, will attempt to use everything, then
use only numeric data

Returns kurt : Series or DataFrame (if level specified)

pandas.DataFrame.kurtosis

DataFrame.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fishers 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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

**kurt**: Series or DataFrame (if level specified)

**pandas.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**

ts.last('5M') -> Last 5 months

**pandas.DataFrame.last_valid_index**

 DataFrame.last_valid_index()

Return label for last non-NA/null value

**pandas.DataFrame.le**

 DataFrame.le(other, axis='columns', level=None)

Wrapper for flexible comparison methods le

**pandas.DataFrame.load**

 DataFrame.load(path)

Deprecated. Use read_pickle instead.

**pandas.DataFrame.lookup**

 DataFrame.lookup(row_labels, col_labels)

Label-based “fancy indexing” function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

**Parameters**

**row_labels**: sequence

The row labels to use for lookup

**col_labels**: sequence
The column labels to use for lookup

Notes

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

Examples

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, 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 data. If None, will attempt to use everything, then
        use only numeric data

Returns mad : Series or DataFrame (if level specified)
```

pandas.DataFrame.mask

```
DataFrame.mask (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False,
    raise_on_error=True)
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 or array
    other : scalar or NDFrame
    inplace : boolean, default False
        Whether to perform the operation in place on the data
```
**pandas.DataFrame.max**

DataFrame.max(\texttt{axis=\text{None}, \text{skipna}=\text{None}, \text{level}=\text{None}, \text{numeric\_only}=\text{None}, **\text{kwargs}})

This method returns the maximum of the values in the object. If you want the index of the maximum, use \texttt{idxmax}. This is the equivalent of the numpy.ndarray method \texttt{argmax}.

**Parameters**

- **axis**: \{index (0), columns (1)}
  
  - \text{skipna}: boolean, default True  
    
    Exclude NA/null values. If an entire row/column is NA, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

\text{max}: Series or DataFrame (if level specified)

**pandas.DataFrame.mean**

DataFrame.mean(\texttt{axis=\text{None}, \text{skipna}=\text{None}, \text{level}=\text{None}, \text{numeric\_only}=\text{None}, **\text{kwargs}})

Return the mean of the values for the requested axis

**Parameters**

- **axis**: \{index (0), columns (1)}
  
  - \text{skipna}: boolean, default True
    
    Exclude NA/null values. If an entire row/column is NA, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

\text{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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **median**: Series or DataFrame (if level specified)

**pandas.DataFrame.memory_usage**

DataFrame.memory_usage \((index=False)\)

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.

**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.

**pandas.DataFrame.merge**

DataFrame.merge \(\text{right, how=’inner’, on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('x', '_y'), copy=True}\)

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 (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)

**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

**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

**Returns merged**: DataFrame

The output type will the be same as ‘left’, if it is a subclass of DataFrame.

**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

>>> merge(A, 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
```
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.

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
- numeric_only : boolean, default None
  Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns

- min : Series or DataFrame (if level specified)

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. Empty if nothing has 2+ occurrences. 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.

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.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.notnull**

`DataFrame.notnull()`

Return a boolean same-sized object indicating if the values are not null

**See also:**

`isnull` boolean inverse of `notnull`
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
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

date : str, 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)
New in version 0.16.2.

Parameters

func : 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 : positional arguments passed into func.

kwargs : 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 on Series or DataFrames. 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=arg2, arg3=c)
... )
```

Examples

```python
>>> df
   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.
```

```python
>>> df.pivot('foo', 'bar', 'baz')
   A  B  C
one 1  2  3
two 4  5  6
```
```python
>>> df.pivot('foo', 'bar')['baz']
A  B  C
one 1  2  3
two 4  5  6
```

### pandas.DataFrame.pivot_table

`DataFrame.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True)`

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

**Parameters**

- `data` : DataFrame
  - `values` : column to aggregate, optional
  - `index` : a column, Grouper, array which has the same length as data, or list of them.
  - `columns` : a column, Grouper, array which has the same length as data, or list of them.
  - `aggfunc` : function, default numpy.mean, or list of functions
    - 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

**Returns**

- `table` : DataFrame

### Examples

```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
```
>>> table = pivot_table(df, values='D', index=['A', 'B'],  
...                     columns=['C'], aggfunc=np.sum)

>>> table

    small  large
  foo    one  1   4
       two  6   NaN
  bar    one  5   4
       two  6   7

pandas.DataFrame.plot

DataFrame.plot(data, 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.

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
Title to use for the plot

**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)

**layout**: tuple (optional)
(rows, columns) for the layout of the plot

**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.
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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns

prod : Series or DataFrame (if level specified)

pandas.DataFrame.product

DataFrame.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), 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
  - **numeric_only**: boolean, default None
    - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **prod**: Series or DataFrame (if level specified)

**pandas.DataFrame.quantile**

```
pandas.DataFrame.quantile(q=0.5, axis=0, numeric_only=True)
```

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

**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**

```
>>> df = DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                  columns=['a', 'b'])
>>> df.quantile(.1)
adtype: float64
>>> df.quantile([.1, .5])
adtype: float64
```

**pandas.DataFrame.query**

```
pandas.DataFrame.query(expr, **kwargs)
```

Query the columns of a frame with a boolean expression.

- New in version 0.13.

- **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.

**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
```

DataFrame.radd

Addition of dataframe and other, element-wise (binary operator `radd`). This is 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, numeric_only=None, method='average', 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

Ranks over columns (0) or rows (1)

numeric_only : boolean, default None

Include only float, int, boolean data


- 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

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 : DataFrame
pandas.DataFrame.rdiv

DataFrame.rdiv \(\text{other, axis} = \text{columns}, \text{level} = \text{None}, \text{fill_value} = \text{None}\)
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 \(\text{index} = \text{None}, \text{columns} = \text{None}, **\text{kwargs}\)\)
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**

- **index, columns**: array-like, optional (can be specified in order, or as 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:
  - 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 size gap to forward or backward fill

**Returns reindexed** : DataFrame

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

**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, 1, ‘index’, ‘columns’}

**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 size gap to forward or backward fill

**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)

return an object with matching indicies to myself

**Parameters**

- **other**: Object
  - **method**: string or None
  - **copy**: boolean, default True
  - **limit**: int, default None
    - Maximum size gap to forward or backward fill

**Returns**

- **reindexed**: same as input

**Notes**

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

**pandas.DataFrame.rename**

DataFrame.rename (index=None, columns=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.

**Parameters**

- **index, columns**: dict-like or function, optional
- **copy**: boolean, default True
- **inplace**: boolean, default False
  - Whether to return a new DataFrame. If True then value of copy is ignored.

**Returns**

- **renamed**: DataFrame (new object)

**pandas.DataFrame.rename_axis**

DataFrame.rename_axis (mapper, axis=0, copy=True, inplace=False)

Alter index and / or columns using 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.

**Parameters**

- **mapper**: dict-like or function, optional
  - **axis**: int or string, default 0
  - **copy**: boolean, default True
    - Also copy underlying data
**pandas: powerful Python data analysis toolkit, Release 0.16.2**

```python
inplace : boolean, default False

Returns renamed : type of caller
```

**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 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 form 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)

Convenience method for frequency conversion and resampling of regular time-series data.

Parameters:
- **rule**: string
  - the offset string or object representing target conversion
- **how**: string
  - method for down- or re-sampling, default to \'mean\' for downsampling
- **axis**: int, optional, default 0
- **fill_method**: string, default None
  - fill_method for upsampling
- **closed**: \{'right', \'left'\}
  - Which side of bin interval is closed
- **label**: \{'right', \'left'\}
  - Which bin edge label to label bucket with
- **convention**: \{'start', \'end', \'s', \'e'\}
- **kind**: \"period"/\"timestamp\"
- **loffset**: timedelta
  - Adjust the resampled time labels
- **limit**: int, default None
  - Maximum size gap to when reindexing with fill_method
- **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

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

**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

**params**

- **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.rmull

DataFrame.matmul(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.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: powerful Python data analysis toolkit, Release 0.16.2

**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.

New in version 0.16.1.

**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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

**pandas.DataFrame.save**

DataFrame.save(path)

Deprecated. Use to_pickle instead

**pandas.DataFrame.select**

DataFrame.select(crit, axis=0)

Return data corresponding to axis labels matching criteria

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 : list-like

A list of dtypes or strings to be included/excluded. You must pass in a non-empty sequence for at least one of these.

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.

TypeError

- If either of include or exclude is not a sequence

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 Pandas categorical dtypes, use `category`

Examples

```python
def = pd.DataFrame({'a': np.random.randn(6).astype('f4'),
                 'b': [True, False] * 3,
                 'c': [1.0, 2.0] * 3})
def
   a   b   c
0  0.3962  True  1
1  0.1459 False  2
2  0.2623  True  1
3  0.0764 False  2
4 -0.9703  True  1
5 -1.2094 False  2
def.select_dtypes(include=['float64'])
c
0  1
1  2
2  1
3  2
4  1
5  2
def.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

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data
Returns `sem`: Series or DataFrame (if level specified)

**pandas.DataFrame.set_axis**

DataFrame.set_axis(axis, labels)

public version of axis assignment

**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**

```python
gold_df = df.set_index(['A', 'B'])
gold_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
gold_df3 = df.set_index([0, 1, 2, 0, 1, 2])
```

**pandas.DataFrame.set_value**

DataFrame.set_value(index, col, value, takeable=False)

Put single value at passed column and index

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(periods=1, freq=None, axis=0, **kwargs)
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 datetools module or time rule (e.g. ‘EOM’). See Notes.
- **axis**: {0, 1, ‘index’, ‘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, 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 data. If None, will attempt to use everything, then use only numeric data

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**

```python
DataFrame.sort(columns=None, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

Sort DataFrame either by labels (along either axis) or by the values in column(s)

**Parameters**

- `columns` : object
  
  Column name(s) in frame. Accepts a column name or a list for a nested sort. A tuple will be interpreted as the levels of a multi-index.

- `ascending` : boolean or list, default True
  
  Sort ascending vs. descending. Specify list for multiple sort orders

- `axis` : {0 or ‘index’, 1 or ‘columns’}, default 0
  
  Sort index/rows versus columns

- `inplace` : boolean, default False
  
  Sort the DataFrame without creating a new instance

- `kind` : {'quicksort', ‘mergesort’, ‘heapsort’}, optional
  
  This option is only applied when sorting on a single column or label.

- `na_position` : {'first', ‘last’} (optional, default='last')
  
  ‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end

**Returns**

- `sorted` : DataFrame

**Examples**

```python
>>> result = df.sort(['A', 'B'], ascending=[1, 0])
```

**pandas.DataFrame.sort_index**

```python
DataFrame.sort_index(axis=0, by=None, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

Sort DataFrame either by labels (along either axis) or by the values in a column

**Parameters**

- `axis` : {0 or ‘index’, 1 or ‘columns’}, default 0
  
  Sort index/rows versus columns

- `by` : object
  
  Column name(s) in frame. Accepts a column name or a list for a nested sort. A tuple will be interpreted as the levels of a multi-index.

- `ascending` : boolean or list, default True
  
  Sort ascending vs. descending. Specify list for multiple sort orders

- `inplace` : boolean, default False
Sort the DataFrame without creating a new instance

    na_position : {'first', 'last'} (optional, default='last')
        'first' puts NaNs at the beginning 'last' puts NaNs at the end
    kind : {'quicksort', 'mergesort', 'heapsort'}, optional
        This option is only applied when sorting on a single column or label.

Returns sorted : DataFrame

Examples

>>> result = df.sort_index(by=['A', 'B'], ascending=[True, False])

pandas.DataFrame.sortlevel

DataFrame.sortlevel (level=0, axis=0, ascending=True, inplace=False, sort_remaining=True)
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

pandas.DataFrame.squeeze

DataFrame.squeeze ()
squeeze length 1 dimensions

pandas.DataFrame.stack

DataFrame.stack (level=-1, dropna=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
    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  b
one 1
    2
two 3
    4
```

**pandas.DataFrame.std**

`DataFrame.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`

Return unbiased 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
  - `numeric_only`: boolean, default None
    - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**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
**pandas.DataFrame.subtract**

DataFrame.subtract(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

**Notes**
Mismatches indices will be unioned together

---

**pandas.DataFrame.sum**

DataFrame.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), 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
  - **numeric_only**: boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**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 \( (i, j, axis=0) \)  
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)

### pandas.DataFrame.tail

DataFrame.tail \( (n=5) \)  
Returns last n rows

### pandas.DataFrame.take

DataFrame.take \( (indices, axis=0, convert=True, is_copy=True) \)  
Analogous to ndarray.take  

**Parameters**  
indices : list / array of ints  
axis : int, default 0  
convert : translate neg to pos indices (default)  
is_copy : mark the returned frame as a copy  

**Returns**  
taken : type of caller

### 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='\n', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding=None, quoting=None, quotechar='"', line_terminator='
', chunksize=None, tupleize_cols=False, date_format=None, doublequote=True, escapechar=None, decimal='. ', **kwds)

Write DataFrame to a comma-separated values (csv) file

Parameters

path_or_buf: string or file handle, default None

File path or object, if None is provided the result is returned as a string.

sep: character, default ","

Field delimiter for the output file.

na_rep: string, default "\n"

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 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, 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

nanRep: None

deprecated, use na_rep

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.

**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

**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

write multi_index columns as a list of tuples (if True) or new (expanded format) 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

New in version 0.16.0.

### `pandas.DataFrame.to_dense`

**DataFrame.to_dense()**

Return dense representation of NDFrame (as opposed to sparse)

### `pandas.DataFrame.to_dict`

**DataFrame.to_dict(*args, **kwargs)**

Convert DataFrame to dictionary.

**Parameters**


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}]

Abbreviations are allowed. s indicates series and sp indicates split.

Returns result : dict like {column -> {index -> value}}

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')

Write DataFrame to a 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 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

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)

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 = ExcelWriter('output.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
```

**pandas.DataFrame.to_gbq**

DataFrame.to_gbq (destination_table, project_id=None, chunksize=10000, verbose=True, reauth=False)

Write a DataFrame to a Google BigQuery table.

THIS IS AN EXPERIMENTAL LIBRARY

If the table exists, the dataframe will be written to the table using the defined table schema and column types. For simplicity, this method uses the Google BigQuery streaming API. The to_gbq method chunks data into a default chunk size of 10,000. Failures return the complete error response which can be quite long depending on the size of the insert. There are several important limitations of the Google streaming API which are detailed at: https://developers.google.com/bigquery/streaming-data-into-bigquery.

**Parameters dataframe**: DataFrame

- **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.
DataFrame\_to\_hdf

**DataFrame\_to\_hdf** *(path\_or\_buf, key, **kwargs)*

activate the HDFStore

**Parameters**

path\_or\_buf : the path (string) or buffer to put the store

key : string

indentifier for the group in the store

mode : optional, \{'a', 'w', 'r', 'r+'\}, default 'a'

'r' Read-only; no data can be modified.

'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

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzma', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dataframe\_to\_html

**DataFrame\_to\_html** *(buf=None, columns=None, col\_space=None, colSpace=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)*

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.

**Parameters**

- **frame** : DataFrame
  - object to render
- **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
- **justify** : {'left', 'right'}, default None
  - Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box.
- **index_names** : bool, optional
  - Prints the names of the indexes, default True
- **force_unicode** : bool, default False
  - Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

**Returns**

- **formatted** : string (or unicode, depending on data and options)
pandas.DataFrame.to_json

pandas.DataFrame.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

date_format : {‘epoch’, ’iso’}
  Type of date conversion. epoch = epoch milliseconds, iso’ = ISO8601, 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 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.

Returns
  same type as input object with filtered info axis
**pandas.DataFrame.to_latex**

`DataFrame.to_latex(buf=None, columns=None, col_space=None, colSpace=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=True, longtable=False, escape=True)`

Render a DataFrame to a tabular environment table. You can splice this into a LaTeX document. Requires `usepackage{booktabs}`.

*to_latex*-specific options:

- **bold_rows** [boolean, default True] Make the row labels bold in the output

- **longtable** [boolean, default False] Use a `longtable` environment instead of tabular. Requires adding a `usepackage{longtable}` to your LaTeX preamble.

- **escape** [boolean, default True] When set to False prevents from escaping latex special characters in column names.

**Parameters**

- **frame** : DataFrame
  - object to render

- **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

- **justify** : {'left', 'right'}, default None
Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box.

index_names : bool, optional
  Prints the names of the indexes, default True

force_unicode : bool, default False
  Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

Returns formatted : string (or unicode, depending on data and options)

pandas.DataFrame.to_msgpack

DataFrame.to_msgpack (path_or_buf=None, **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_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)

Pickle (serialize) object to input file path

**Parameters**
- path : string
  - File path

**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

**pandas.DataFrame.to_sql**

DataFrame.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'
  - The flavor of SQL to use. Ignored when using SQLAlchemy engine. 'mysql' is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.
- 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.DataFrame.to_stata

Dataframe.to_stata(fname = None, convert_dates=None, write_index=True, encoding='latin-1', byteorder=None, time_stamp=None, data_label=None)
A class for writing Stata binary dta files from array-like objects

Parameters
fname : file path or buffer
Where to save the dta file.

convert_dates : dict
Dictionary mapping column of datetime types to the stata internal format that you want to use for the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either a number or a name.

encoding : str
Default is latin-1. Note that Stata does not support unicode.

byteorder : str
Can be “>”, “<”, “little”, or “big”. The default is None which uses sys.byteorder

Examples

>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()

Or with dates

>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
pandas.DataFrame.to_string

DataFrame.to_string(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, line_width=None, max_rows=None, max_cols=None, show_dimensions=False)

Render a DataFrame to a console-friendly tabular output.

**Parameters**

- **frame**: DataFrame
  - object to render
- **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
- **justify**: {'left', 'right'}, default None
  - Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box.
- **index_names**: bool, optional
  - Prints the names of the indexes, default True
- **force_unicode**: bool, default False
  - Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.
**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_wide**

`DataFrame.to_wide(*args, **kwargs)`

**pandas.DataFrame.transpose**

`DataFrame.transpose()`

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 NDFrame before and/or after some particular dates.

**Parameters**

- **before** : date
  - Truncate before date
- **after** : date
  - Truncate after date
- **axis** : the truncation axis, defaults to the stat axis
- **copy** : boolean, default is True,
  - return a copy of the truncated section

**Returns**

truncated : type of caller

**pandas.DataFrame.tshift**

DataFrame.tshift(periods=1, freq=None, axis=0, **kwargs)

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 datetools 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

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

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)
        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

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

**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
    b    2
two    a    3
    b    4
dtype: float64

>>> s.unstack(level=-1)
      a  b
one  1  2
two  3  4

>>> s.unstack(level=0)
     one  two
      a  1  3
      b  2  4

>>> df = s.unstack(level=0)
>>> df.unstack()
     one  two
      a  1.
      b  3.
     two  a  2.
      b  4.
```

**DataFrame.update**

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.

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

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns var : Series or DataFrame (if level specified)

pandas.DataFrame.where

DataFrame.where (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array

other : scalar or NDFrame

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

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)

Returns wh : same type as caller
pandas.DataFrame.xs

DataFrame.xs(key, axis=0, level=None, copy=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.

copy : boolean [deprecated]
Whether to make a copy of the data

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 functionality, see MultiIndex 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
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  1  7  5  5  0
```
```python
>>> df.xs(('baz', 'three'))
A   B   C   D
third
2 5 3 5 3
>>> df.xs('one', level=1)
A   B   C   D
first third
bar 1 4 1 8 9
baz 1 6 6 8 0
>>> df.xs(('baz', 2), level=[0, 'third'))
A   B   C   D
second
three 5 3 5 3
```

### 33.4.2 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>DataFrame.as_matrix([columns])</td>
<td>Convert the frame to its Numpy-array representation.</td>
</tr>
<tr>
<td>DataFrame.dtypes</td>
<td>Return the dtypes in this object.</td>
</tr>
<tr>
<td>DataFrame.ftypes</td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<tr>
<td>DataFrame.get_dtype_counts()</td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>DataFrame.get_ftype_counts()</td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td>DataFrame.select_dtypes([include, exclude])</td>
<td>Return a subset of a DataFrame including/excluding columns based on their</td>
</tr>
<tr>
<td></td>
<td>Numpy representation of NDFrame</td>
</tr>
<tr>
<td>DataFrame.axes</td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td>DataFrame.ndim</td>
<td>number of elements in the NDFrame</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.as_matrix**

```python
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 upcast to int32.

This method is provided for backwards compatibility. Generally, it is recommended to use `.values`.

### pandas.DataFrame.dtypes

DataFrame.dtypes

Return the dtypes in this object

### pandas.DataFrame.ftypes

DataFrame.ftypes

Return the ftypes (indication of sparse/dense and dtype) in this object.

### pandas.DataFrame.get_dtype_counts

DataFrame.get_dtype_counts()

Return the counts of dtypes in this object

### pandas.DataFrame.get_ftype_counts

DataFrame.get_ftype_counts()

Return the counts of ftypes in this object

### 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**: list-like
  A list of dtypes or strings to be included/excluded. You must pass in a non-empty sequence for at least one of these.

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.
TypeError

- If either of include or exclude is not a sequence

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 Pandas categorical dtypes, use ‘category’

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
1  0.1459 False  2
2  0.2623 True  1
3  0.0764 False  2
4 -0.9703 True  1
5 -1.2094 False  2
>>> df.select_dtypes(include=['float64'])
   c
0  1
1  2
2  1
3  2
4  1
5  2
>>> df.select_dtypes(exclude=['floating'])
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

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 upcase to int32.

**pandas.DataFrame.axes**

DataFrame.\texttt{axes}

**pandas.DataFrame.ndim**

DataFrame.\texttt{ndim}

Number of axes / array dimensions

**pandas.DataFrame.size**

DataFrame.\texttt{size}

number of elements in the NDFrame

**pandas.DataFrame.shape**

DataFrame.\texttt{shape}

33.4.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.\texttt{astype}(dtype[, copy, raise_on_error])</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>DataFrame.\texttt{convert_objects}([convert_dates, ...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>DataFrame.\texttt{copy}([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>DataFrame.\texttt{isnull}()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>DataFrame.\texttt{notnull}()</td>
<td>Return a boolean same-sized object indicating if the values are true</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.astype**

DataFrame.\texttt{astype}(dtype, copy=True, raise_on_error=True, **kwargs)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

**Parameters**

dtype : numpy.dtype or Python type
raise_on_error : raise on invalid input
kwargs : keyword arguments to pass on to the constructor

**Returns**
casted : type of caller

**pandas.DataFrame.convert_objects**

DataFrame.\texttt{convert_objects}(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)

Attempt to infer better dtype for object columns

**Parameters**

correct_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

### pandas.DataFrame.copy

DataFrame.copy (deep=True)

Make a copy of this object

**Parameters**

- **deep** : boolean or string, default True

  Make a deep copy, i.e. also copy data

**Returns**

- **copy** : type of caller

### pandas.DataFrame.isnull

DataFrame.isnull()

Return a boolean same-sized object indicating if the values are null

See also:

- **notnull** boolean inverse of isnull

### pandas.DataFrame.notnull

DataFrame.notnull()

Return a boolean same-sized object indicating if the values are not null

See also:

- **isnull** boolean inverse of notnull

#### 33.4.4 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.head([n])</td>
<td>Returns first n rows</td>
</tr>
<tr>
<td>DataFrame.at</td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td>DataFrame.iat</td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td>DataFrame.ix</td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
</tbody>
</table>
Table 33.54 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.loc</td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td>DataFrame.iloc</td>
<td>Purely integer-location based indexer for selection by position.</td>
</tr>
<tr>
<td>DataFrame.insert</td>
<td>Insert column into DataFrame at specified location.</td>
</tr>
<tr>
<td>DataFrame.<strong>iter</strong></td>
<td>Iterate over index axis.</td>
</tr>
<tr>
<td>DataFrame.iteritems()</td>
<td>Iterate over (column, series) pairs.</td>
</tr>
<tr>
<td>DataFrame.iterrows()</td>
<td>Iterate over rows of DataFrame as (index, Series) pairs.</td>
</tr>
<tr>
<td>DataFrame.lookup</td>
<td>Label-based “fancy indexing” function for DataFrame.</td>
</tr>
<tr>
<td>DataFrame.pop(item)</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>DataFrame.tail(n)</td>
<td>Returns last n rows</td>
</tr>
<tr>
<td>DataFrame.xs(key[, axis, level, copy, ...])</td>
<td>Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.</td>
</tr>
<tr>
<td>DataFrame.isin(values)</td>
<td>Return boolean DataFrame showing whether each element in the DataFrame is in values.</td>
</tr>
<tr>
<td>DataFrame.where(cond[, other, inplace, ...])</td>
<td>Return an object of same shape as self and whose corresponding entries are from cond.</td>
</tr>
<tr>
<td>DataFrame.mask(cond[, other, inplace, axis, ...])</td>
<td>Return an object of same shape as self and whose corresponding entries are from not cond.</td>
</tr>
<tr>
<td>DataFrame.query(expr, **kwargs)</td>
<td>Query the columns of a frame with a boolean expression.</td>
</tr>
</tbody>
</table>

pandas.DataFrame.head

DataFrame.head(n=5)
Returns first n rows.

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.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.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.

.loc will raise a KeyError when the items are not found.

See more at Selection by Label

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.

.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.insert

DataFrame.insert(loc, column, value, allow_duplicates=False)
Insert column into DataFrame at specified location.

If allow_duplicates is False, raises Exception if column is already contained in the DataFrame.

Parameters
loc : int
    Must have 0 <= loc <= len(columns)

column : object

value : int, Series, or array-like
pandas.DataFrame.__iter__

DataFrame.__iter__()  
Iterate over index axis

pandas.DataFrame.iteritems

DataFrame.iteritems()  
Iterator over (column, series) pairs

pandas.DataFrame.iterrows

DataFrame.iterrows()  
Iterate over rows of DataFrame as (index, Series) pairs.

Returns  
A generator that iterates over the rows of the frame.

Notes

*iterrows does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```python
>>> df = DataFrame([[1, 1.0]], columns=['x', 'y'])
>>> row = next(df.iterrows())[1]
>>> print(row['x'].dtype)
float64
>>> print(df['x'].dtype)
int64
```

pandas.DataFrame.itertuples

DataFrame.itertuples(index=True)  
Iterate over rows of DataFrame as tuples, with index value as first element of the tuple

pandas.DataFrame.lookup

DataFrame.lookup(row_labels, col_labels)  
Label-based “fancy indexing” function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

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

values [ndarray] The found values

**pandas.DataFrame.pop**

DataFrame.pop(item)

Return item and drop from frame. Raise KeyError if not found.

**pandas.DataFrame.tail**

DataFrame.tail(n=5)

Returns last n rows

**pandas.DataFrame.xs**

DataFrame.xs(key, axis=0, level=None, copy=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.
- **copy**: boolean [deprecated]
  Whether to make a copy of the data
- **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 functionality, see MultiIndex 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  2
   b  9
   c  3
Name: C
```

```python
>>> 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'))
     A  B  C  D
    third
   2  5  3  5  3
>>> df.xs('one', level=1)
     A  B  C  D
    first third
   bar  1  4  1  8  9
   baz  1  6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
     A  B  C  D
    second
   three  5  3  5  3
```

**pandas.DataFrame.isin**

Dataframe.[isin](\textit{values})

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

**Parameters** \textit{values} : iterable, Series, DataFrame or dictionary

The result will only be true at a location if all the labels match. If \textit{values} is a Series, that’s the index. If \textit{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:

```python
df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
df.isin([1, 3, 12, 'a'])
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True  True</td>
</tr>
<tr>
<td>1</td>
<td>False False</td>
</tr>
<tr>
<td>2</td>
<td>True  False</td>
</tr>
</tbody>
</table>

When `values` is a dict:

```python
df = DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
df.isin({'A': [1, 3], 'B': [4, 7, 12]})
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True False # Note that B didn't match the 1 here.</td>
</tr>
<tr>
<td>1</td>
<td>False  True</td>
</tr>
<tr>
<td>2</td>
<td>True  True</td>
</tr>
</tbody>
</table>

When `values` is a Series or DataFrame:

```python
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)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True False</td>
</tr>
<tr>
<td>1</td>
<td>False False # Column A in 'other' has a 3, but not at index 1.</td>
</tr>
<tr>
<td>2</td>
<td>True  True</td>
</tr>
</tbody>
</table>

### pandas.DataFrame.where

DataFrame.where(cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array
- **other** : scalar or NDFrame
- **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
- **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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

Returns wh : same type as caller

pandas.DataFrame.mask

DataFrame.mask (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array

other : scalar or NDFrame

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

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)

Returns wh : same type as caller

pandas.DataFrame.query

DataFrame.query (expr, **kwargs)

Query the columns of a frame with a boolean expression.

New in version 0.13.

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.

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
```

For more information on `.at`, `.iat`, `.ix`, `.loc`, and `.iloc`, see the `indexing` documentation.

### 33.4.5 Binary operator functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.add(other[, axis, level, fill_value])</code></td>
<td>Addition of dataframe and other, element-wise (binary operator <code>add</code>).</td>
</tr>
<tr>
<td><code>DataFrame.sub(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.mul(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.div(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.floordiv(other[, axis, level, ...])</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.mod(other[, axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.pow(other[, axis, level, fill_value])</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.radd(other[, axis, level, fill_value])</code></td>
<td>Addition of dataframe and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rsub(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmul(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rdiv(other[, axis, level, ...])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rfloordiv(other[, axis, level, ...])</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmod(other[, axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rpow(other[, axis, level, fill_value])</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.lt(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>lt</code>.</td>
</tr>
<tr>
<td><code>DataFrame.gt(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>gt</code>.</td>
</tr>
<tr>
<td><code>DataFrame.le(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>le</code>.</td>
</tr>
<tr>
<td><code>DataFrame.ge(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>ge</code>.</td>
</tr>
<tr>
<td><code>DataFrame.ne(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>ne</code>.</td>
</tr>
<tr>
<td><code>DataFrame.eq(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods <code>eq</code>.</td>
</tr>
<tr>
<td><code>DataFrame.combine(other, func[, fill_value, ...])</code></td>
<td>Add two DataFrame objects and do not propagate NaN values, so if for a DataFrame combine the method.</td>
</tr>
<tr>
<td><code>DataFrame.combineAdd(other)</code></td>
<td>Add two DataFrame objects and default to non-null values in frame calling the method.</td>
</tr>
<tr>
<td><code>DataFrame.combine_first(other)</code></td>
<td>Combine two DataFrame objects and default to non-null values in frame calling the method.</td>
</tr>
</tbody>
</table>

Continued
DataFramecombineMult

Multiply two DataFrame objects and do not propagate NaN values, so if

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

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

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.

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

dataframe.div

DataFrame.div(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.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.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
**pandas** powerful Python data analysis toolkit, Release 0.16.2


**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.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.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.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.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.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.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.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.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.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.lt

DataFrame.lt(other, axis='columns', level=None)

Wrapper for flexible comparison methods lt

pandas.DataFrame.gt

DataFrame.gt(other, axis='columns', level=None)

Wrapper for flexible comparison methods gt
pandas.DataFrame.le

DataFrame.le\( (other, axis='columns', level=None) \)
Wrapper for flexible comparison methods le

pandas.DataFrame.ge

DataFrame.ge\( (other, axis='columns', level=None) \)
Wrapper for flexible comparison methods ge

pandas.DataFrame.ne

DataFrame.ne\( (other, axis='columns', level=None) \)
Wrapper for flexible comparison methods ne

pandas.DataFrame.eq

DataFrame.eq\( (other, axis='columns', level=None) \)
Wrapper for flexible comparison methods eq

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.combineAdd

DataFrame.combineAdd\( (other) \)
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

Returns

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.combineMult**

`DataFrame.combineMult(other)`

Multiply 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

Returns `DataFrame`

### 33.4.6 Function application, GroupBy

**DataFrame.apply(func[, axis, broadcast, ...])**

Applies function along input axis of DataFrame.

**DataFrame.applymap(func)**

Apply a function to a DataFrame that is intended to operate elementwise, i.e.

**DataFrame.groupby([by, axis, level, ...])**

Group series using mapper (dict or key function, apply given function

**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

- `reduce` : boolean or None, default False
  - 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 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.
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.

args : tuple

Positional arguments to pass to function in addition to the array/series

Additional keyword arguments will be passed as keywords to the function

Returns applied : Series or DataFrame

See also:

**DataFrame.applymap** For elementwise 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

**pandas.DataFrame.groupby**

Dataframe.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False)

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 / list of functions, dict, Series, or tuple /
list of column names. Called on each element of the object index to determine the
groups. If a dict or Series is passed, the Series or dict VALUES will be used to
determine the groups.

**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

**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()
```

### 33.4.7 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.</td>
</tr>
<tr>
<td><code>DataFrame.all()</code></td>
<td>Return whether all elements are True over requested axis</td>
</tr>
<tr>
<td><code>DataFrame.any()</code></td>
<td>Return whether any element is True over requested axis</td>
</tr>
<tr>
<td><code>DataFrame.clip()</code></td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td><code>DataFrame.clip_lower()</code></td>
<td>Return copy of the input with values below given value(s) truncated</td>
</tr>
<tr>
<td><code>DataFrame.clip_upper()</code></td>
<td>Return copy of input with values above given value(s) truncated</td>
</tr>
<tr>
<td><code>DataFrame.corr()</code></td>
<td>Compute pairwise correlation of columns, excluding NA/null values</td>
</tr>
<tr>
<td><code>DataFrame.corrwith()</code></td>
<td>Compute pairwise correlation between rows or columns of two DataFrame objects</td>
</tr>
<tr>
<td><code>DataFrame.count()</code></td>
<td>Return Series with number of non-NA/null observations over requested axis</td>
</tr>
<tr>
<td><code>DataFrame.cov()</code></td>
<td>Compute pairwise covariance of columns, excluding NA/null values</td>
</tr>
<tr>
<td><code>DataFrame.cummax()</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.cummin()</code></td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.cumprod()</code></td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.cumsum()</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
</tbody>
</table>

---

33.4. DataFrame 1233
Table 33.57 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.describe</td>
<td>Generate various summary statistics, excluding NaN values.</td>
</tr>
<tr>
<td>DataFrame.diff</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>DataFrame.eval</td>
<td>Evaluate an expression in the context of the calling DataFrame instance.</td>
</tr>
<tr>
<td>DataFrame.kurt</td>
<td>Return unbiased kurtosis over requested axis using Fishers definition of kurtosis.</td>
</tr>
<tr>
<td>DataFrame.mad</td>
<td>This method returns the maximum of the values in the object.*********************************************************************************</td>
</tr>
<tr>
<td>DataFrame.max</td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td>DataFrame.mean</td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td>DataFrame.median</td>
<td>Gets the mode(s) of each element along the axis selected.</td>
</tr>
<tr>
<td>DataFrame.pct_change</td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td>DataFrame.prod</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>DataFrame.quantile</td>
<td>Return values at the given quantile over requested axis, a la numpy.percentile.</td>
</tr>
<tr>
<td>DataFrame.rank</td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td>DataFrame.sem</td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td>DataFrame.skew</td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td>DataFrame.sum</td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td>DataFrame.std</td>
<td>Return unbiased standard deviation over requested axis</td>
</tr>
<tr>
<td>DataFrame.var</td>
<td>Return unbiased variance over requested axis</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.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  
level : int or level name, default None  
bool_only : boolean, default None  

**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 data. If None, will attempt to use everything, then use only boolean data

**Returns any**: Series or DataFrame (if level specified)

---

**pandas.DataFrame.clip**

DataFrame.clip(lower=None, upper=None, out=None, axis=None)

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.

**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  -0.3
 1  -0.2
 2  -0.1
 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
```
pandas.DataFrame.clip_lower

DataFrame.clip_lower(threshold, axis=None)
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.

Returns
- clipped : same type as input

See also:
- clip

pandas.DataFrame.clip_upper

DataFrame.clip_upper(threshold, axis=None)
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.

Returns
- clipped : same type as input

See also:
- clip

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, dtype=None, out=None, skipna=True, **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:
  max : Series

pandas.DataFrame.cummin

DataFrame.cummin (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative min 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:
  min : Series

pandas.DataFrame.cumprod

DataFrame.cumprod (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative prod 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:
  prod : Series

pandas.DataFrame.cumsum

DataFrame.cumsum (axis=None, dtype=None, out=None, skipna=True, **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:
  sum : Series

pandas.DataFrame.describe

DataFrame.describe (percentile_width=None, percentiles=None, include=None, exclude=None)
Generate various summary statistics, excluding NaN values.

Parameters:
  percentile_width : float, deprecated
The `percentile_width` argument will be removed in a future version. Use `percentiles` instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

**percentiles** : array-like, optional

The percentiles to include in the output. Should all be in the interval [0, 1]. By default `percentiles` is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

**include, exclude** : list-like, ‘all’, or None (default)

Specify the form of the returned result. Either:

- None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
- A list of dtypes or strings to be included/excluded. To select all numeric types use `numpy` `numpy.number`. To select categorical objects use `type` object. See also the `select_dtypes` documentation. eg. `df.describe(include=['O'])`
- If include is the string ‘all’, the output column-set will match the input one.

**Returns** summary: `NDFrame` of summary statistics

**See also:**

`DataFrame.select_dtypes`

**Notes**

The output `DataFrame` index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with `NaN`. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the `count` and `most common` pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for `Series`.

**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.eval

DataFrame.eval(expr, **kwargs)
Evaluate an expression in the context of the calling DataFrame instance.

Parameters expr : string
The expression string to evaluate.

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.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')
```

pandas.DataFrame.kurt

DataFrame.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis using Fishers 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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns kurt : Series or DataFrame (if level specified)
**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- *mad*: Series or DataFrame (if level specified)

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**  **median** : Series or DataFrame (if level specified)

### 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.

**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

**numeric_only** : boolean, default None

   Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**  **min** : Series or DataFrame (if level specified)

### pandas.DataFrame.mode

**DataFrame.mode**(axis=0, numeric_only=False)

Gets the mode(s) of each element along the axis selected. Empty if nothing has 2+ occurrences. 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
def = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
def.mode()
```

```
      A
0   1
1   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
  - 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.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
Excluding 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

numeric_only: boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns prod: Series or DataFrame (if level specified)

pandas.DataFrame.quantile

DataFrame.quantile(q=0.5, axis=0, numeric_only=True)
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

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  2.5  55.0
```

pandas.DataFrame.rank

DataFrame.rank(axis=0, numeric_only=None, method='average', 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
Ranks over columns (0) or rows (1)
**numeric_only** : boolean, default None
Include only float, int, boolean data

**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

**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** : DataFrame

---

**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
- **numeric_only** : boolean, default None
  Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **sem** : Series or DataFrame (if level specified)

---

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **skew**: Series or DataFrame (if level specified)

---

**pandas.DataFrame.sum**

DataFrame.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), 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
  - **numeric_only**: boolean, default None
    
    Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **sum**: Series or DataFrame (if level specified)

---

**pandas.DataFrame.std**

DataFrame.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased 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
  - **numeric_only**: boolean, default None
    
    Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **std**: Series or DataFrame (if level specified)
**Returns std**: Series or DataFrame (if level specified)

**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
- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns var**: Series or DataFrame (if level specified)

### 33.4.8 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.add_prefix(prefix)</td>
<td>Concatenate prefix string with panel items names.</td>
</tr>
<tr>
<td>DataFrame.add_suffix(suffix)</td>
<td>Concatenate suffix string with panel items names.</td>
</tr>
<tr>
<td>DataFrame.align(other[, join, axis, level, ...])</td>
<td>Align two object on their axes with the.</td>
</tr>
<tr>
<td>DataFrame.drop(labels[, axis, level, ...])</td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td>DataFrame.drop_duplicates(*args, **kwargs)</td>
<td>Return DataFrame with duplicate rows removed, optionally only.</td>
</tr>
<tr>
<td>DataFrame.duplicated(*args, **kwargs)</td>
<td>Return boolean Series denoting duplicate rows, optionally only.</td>
</tr>
<tr>
<td>DataFrame.equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td>DataFrame.filter([items, like, regex, axis])</td>
<td>Restrict the info axis to set of items or wildcard.</td>
</tr>
<tr>
<td>DataFrame.first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data.</td>
</tr>
<tr>
<td>DataFrame.head([n])</td>
<td>Return first n rows.</td>
</tr>
<tr>
<td>DataFrame.idxmax([axis, skipna])</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td>DataFrame.idxmin([axis, skipna])</td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td>DataFrame.last(offset)</td>
<td>Convenience method for subsetting final periods of time series data.</td>
</tr>
<tr>
<td>DataFrame.reindex([index, columns])</td>
<td>Conform DataFrame to new index with optional filling logic, placing NA/NAN.</td>
</tr>
<tr>
<td>DataFrame.reindex_axis(labels[, axis, ...])</td>
<td>Conform input object to new index with optional filling logic, placing NA/NAN.</td>
</tr>
<tr>
<td>DataFrame.rename([index, columns])</td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td>DataFrame.reset_index([level, drop, ...])</td>
<td>For DataFrame with multi-level index, return new DataFrame with labeling.</td>
</tr>
<tr>
<td>DataFrame.sample([n, frac, replace, ...])</td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td>DataFrame.select(crit[, axis])</td>
<td>Return data corresponding to axis labels matching criteria.</td>
</tr>
<tr>
<td>DataFrame.set_index(keys[, drop, append, ...])</td>
<td>Set the DataFrame index (row labels) using one or more existing columns.</td>
</tr>
<tr>
<td>DataFrame.tail([n])</td>
<td>Returns last n rows.</td>
</tr>
<tr>
<td>DataFrame.take(indices[, axis, convert, is_copy])</td>
<td>Analogous to ndarray.take.</td>
</tr>
<tr>
<td>DataFrame.truncate([before, after, axis, copy])</td>
<td>Truncates a sorted NDFrame before and/or after some particular dates.</td>
</tr>
</tbody>
</table>
**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.align**

DataFrame.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)

Align two object on their axes with the specified join method for each axis Index

**Parameters**
- **other**: DataFrame or Series
- **join**: {'outer', 'inner', 'left', 'right'}, default 'outer'
- **axis**: allowed axis of the other object, default None
  - Align on index (0), columns (1), or both (None)
- **level**: int or level name, default None
  - Broadcast across a level, matching Index values on the passed MultiIndex level
- **copy**: 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, 1}, default 0
  - Filling axis, method and limit

**Returns**
- **(left, right)**: (type of input, type of other)
  - Aligned objects

**pandas.DataFrame.drop**

DataFrame.drop(labels, axis=0, level=None, inplace=False, errors='raise')

Return new object with labels in requested axis removed
**Parameters**

- **labels** : single label or list-like
  - **axis** : int or axis name
  - **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.
    - New in version 0.16.1.

**Returns**

- **dropped** : type of caller

---

**pandas.DataFrame.drop_duplicates**

DataFrame.drop_duplicates(*args, **kwargs)

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
- **take_last** : boolean, default False
  - Take the last observed row in a row. Defaults to the first row
- **inplace** : boolean, default False
  - Whether to drop duplicates in place or to return a copy
- **cols** : kwags only argument of subset [deprecated]

**Returns**

- **deduplicated** : DataFrame

---

**pandas.DataFrame.duplicated**

DataFrame.duplicated(*args, **kwargs)

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
- **take_last** : boolean, default False
  - For a set of distinct duplicate rows, flag all but the last row as duplicated. Default is for all but the first row to be flagged
- **cols** : kwargs only argument of subset [deprecated]

**Returns**

- **duplicated** : Series
pandas.DataFrame.equals

**DataFrame.equals (other)**
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

pandas.DataFrame.filter

**DataFrame.filter (items=None, like=None, regex=None, axis=None)**
Restrict the info axis to set of items or wildcard

- **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 None
    The axis to filter on. By default this is the info axis. The “info axis” is the axis that is used when indexing with[]. For example,df = DataFrame({'a': [1, 2, 3, 4]}); df['a']. So, the DataFrame columns are the info axis.

**Notes**

Arguments are mutually exclusive, but this is not checked for

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.last('10D') -> First 10 days

pandas.DataFrame.head

**DataFrame.head (n=5)**
Returns first n rows
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.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

ts.last(‘5M’) -> Last 5 months
**pandas: powerful Python data analysis toolkit, Release 0.16.2**

---

### pandas.DataFrame.reindex

**DataFrame.reindex (index=None, columns=None, **kwargs)**

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**

- **index**, **columns**: array-like, optional (can be specified in order, or as 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:**

- 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 size gap to forward or backward fill

**Returns** **reindexed**: DataFrame

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

---

### 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, 1, ‘index’, ‘columns’}
- **method**: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}, optional

---

1252 Chapter 33. API Reference
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 size gap to forward or backward fill

**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)
return an object with matching indicies to myself

**Parameters**

- **other**: Object
- **method**: string or None
- **copy**: boolean, default True
- **limit**: int, default None
  Maximum size gap to forward or backward fill

**Returns**: reindexed : same as input

**Notes**

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

**pandas.DataFrame.rename**

DataFrame.rename(index=None, columns=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.

**Parameters**

- **index, columns**: dict-like or function, optional
Transformation to apply to that axis values

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.

Returns renamed : DataFrame (new object)

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

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.
New in version 0.16.1.

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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

pandas.DataFrame.select

DataFrame.select (crit, axis=0)

Return data corresponding to axis labels matching criteria

Parameters crit : function

To be called on each index (label). Should return True or False

axis : int

Returns selection : type of caller

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

```python
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])
```

**pandas.DataFrame.tail**

`DataFrame.tail(n=5)`

Returns last n rows

**pandas.DataFrame.take**

`DataFrame.take(indices, axis=0, convert=True, is_copy=True)`

Analogous to `ndarray.take`

- **Parameters**
  - `indices`: list / array of ints
  - `axis`: int, default 0
  - `convert`: translate neg to pos indices (default)
  - `is_copy`: mark the returned frame as a copy

- **Returns**
  - `taken`: type of caller

**pandas.DataFrame.truncate**

`DataFrame.truncate(before=None, after=None, axis=None, copy=True)`

Truncates a sorted NDFrame before and/or after some particular dates.

- **Parameters**
  - `before`: date
  - `after`: date
  - `axis`: the truncation axis, defaults to the stat axis
  - `copy`: boolean, default is True,
  - return a copy of the truncated section

- **Returns**
  - `truncated`: type of caller

### 33.4.9 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’.
**pandas.DataFrame.dropna**

DataFrames provide a `.dropna()` method to drop labels on a given axis where any or all of the data are missing. The method accepts several 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 inplace and return None.

Returns **dropped** : DataFrame

**pandas.DataFrame.fillna**

DataFrames also offer a `.fillna()` method to fill NA/NaN values using the specified method. Parameters include:

- **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, ‘index’, ‘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.
- **downcast**: dict, default is None.
pandas: powerful Python data analysis toolkit, Release 0.16.2

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

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 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 form 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.

33.4.10 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.pivot(index, columns, values)</td>
<td>Reshape data (produce a “pivot” table) based on column values.</td>
</tr>
<tr>
<td>DataFrame.reorder_levels(order[, axis])</td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td>DataFrame.sort(columns, axis, ascending, ...)</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>DataFrame.sort_index(axis, by, ascending, ...)</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>DataFrame.swaplevel(i, j[, axis])</td>
<td>Sort multilevel index by chosen axis and primary level.</td>
</tr>
<tr>
<td>DataFrame.stack([level, dropna])</td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame.</td>
</tr>
<tr>
<td>DataFrame.unstack([level])</td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame.</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.transpose()</td>
<td>Transpose index and columns</td>
</tr>
</tbody>
</table>

33.4. DataFrame
**pandas.DataFrame.pivot**

Dataframe.pivot \(\text{\textit{index=None, columns=None, values=None}}\)

Reshape data (produce a “pivot” table) based on column values. Uses unique values from index / columns to form axes and return either DataFrame or Panel, depending on whether you request a single value column (DataFrame) or all columns (Panel).

**Parameters**

- **index**: string or object
  - Column name to use to make new frame’s 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

**Returns**

- **pivoted**: DataFrame
  - If no values column specified, will have hierarchically indexed columns

**Notes**

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods

**Examples**

```python
>>> df
   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('foo', 'bar', 'baz')
    A  B  C
one 1  2  3
two 4  5  6

>>> df.pivot('foo', 'bar')['baz']
    A  B  C
one 1  2  3
two 4  5  6
```

**pandas.DataFrame.reorder_levels**

Dataframe.reorder_levels \(\text{\textit{order, axis=0}}\)

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.sort

DataFrame.sort(columns=None, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')

Sort DataFrame either by labels (along either axis) or by the values in column(s)

Parameters columns : object

Column name(s) in frame. Accepts a column name or a list for a nested sort. A tuple will be interpreted as the levels of a multi-index.

ascending : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

axis : {0 or ‘index’, 1 or ‘columns’}, default 0

Sort index/rows versus columns

inplace : boolean, default False

Sort the DataFrame without creating a new instance

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

This option is only applied when sorting on a single column or label.

na_position : {'first', 'last'} (optional, default='last')

‘first’ puts NaNs at the beginning ‘last’ puts NaNs at the end

Returns sorted : DataFrame

Examples

>>> result = df.sort([‘A’, ‘B’], ascending=[1, 0])

pandas.DataFrame.sort_index

DataFrame.sort_index(axis=0, by=None, ascending=True, inplace=False, kind='quicksort', na_position='last')

Sort DataFrame either by labels (along either axis) or by the values in a column

Parameters axis : {0 or ‘index’, 1 or ‘columns’}, default 0

Sort index/rows versus columns

by : object

Column name(s) in frame. Accepts a column name or a list for a nested sort. A tuple will be interpreted as the levels of a multi-index.

ascending : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

inplace : boolean, default False
Sort the DataFrame without creating a new instance

```
na_position : {'first', 'last'} (optional, default='last')
    'first' puts NaNs at the beginning 'last' puts NaNs at the end
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    This option is only applied when sorting on a single column or label.
```

Returns sorted : DataFrame

Examples

```
>>> result = df.sort_index(by=[‘A’, ‘B’], ascending=[True, False])
```

**pandas.DataFrame.sortlevel**

```
DataFrame.sortlevel (level=0, axis=0, ascending=True, inplace=False, sort_remaining=True)
```

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

**pandas.DataFrame.swaplevel**

```
DataFrame.swaplevel (i, j, 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)

**pandas.DataFrame.stack**

```
DataFrame.stack (level=-1, dropna=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 1.0 2.0
  two 3.0 4.0

>>> s.stack()
  one a 1
       b 2
  two a 3
       b 4
```

---

**pandas.DataFrame.unstack**

**DataFrame.unstack**(level=-1)

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

**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.unstack(level=0)
    one  two
   a  1  3
   b  2  4

>>> df = s.unstack(level=0)
>>> df.unstack()
    one  a  1.
          b  3.
    two  a  2.
          b  4.

\textbf{pandas.DataFrame.T}

\begin{verbatim}
DataFrame.T
    Transpose index and columns
\end{verbatim}

\textbf{pandas.DataFrame.to_panel}

\begin{verbatim}
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
    \textbf{Returns} panel : Panel
\end{verbatim}

\textbf{pandas.DataFrame.transpose}

\begin{verbatim}
DataFrame.transpose()
    Transpose index and columns
\end{verbatim}

\textbf{33.4.11 Combining / joining / merging}

\begin{verbatim}
DataFrame.append(other[, ignore_index, ...])
    Append rows of \textit{other} to the end of this frame, returning a new object.
    Columns not in this frame are added as new columns.

    \textbf{Parameters} other : DataFrame or Series/dict-like object, or list of these

    \textbf{DataFrame.assign(**kwargs)}
    Assign new columns to a DataFrame, returning a new object (a copy) with all the new columns.

    \textbf{DataFrame.join(other[, on, how, lsuffix, ...])}
    Join columns with other DataFrame either on index or on a key column.

    \textbf{DataFrame.merge(right[, how, on, left_on, ...])}
    Merge DataFrame objects by performing a database-style join operation by columns or indexes.

    \textbf{DataFrame.update(other[, join, overwrite, ...])}
    Modify DataFrame in place using non-NA values from passed DataFrame.
\end{verbatim}
**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.

**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)
```

```plaintext
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

**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.

New in version 0.16.0.

**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. 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

Since `kwargs` is a dictionary, the order of your arguments may not be preserved. The make things predictable, 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>0.00000</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'])
```

```python
>>> 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>0.00000</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.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) to use for joining, otherwise join 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’\}

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

**sort**: boolean, default False

Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

**Returns** joined : DataFrame

**Notes**

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

**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)

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 (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)

- 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

**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

Returns **merged** : DataFrame
The output type will the be same as ‘left’, if it is a subclass of DataFrame.

### 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

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
       lkey  value_x  rkey  value_y
    0   foo    1.0   foo    5.0
    1   foo    4.0   foo    5.0
    2   bar    2.0   bar    6.0
    3   bar    2.0   bar    8.0
    4   baz    3.0  NaN   NaN
    5  NaN  NaN    7.0
```

### 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.

### 33.4.12 Time series-related

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.asfreq(freq[, method, how, normalize])</td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.</td>
</tr>
<tr>
<td>DataFrame.shift([periods, freq, axis])</td>
<td>Shift index by desired number of periods with an optional time freq</td>
</tr>
<tr>
<td>DataFrame.first_valid_index()</td>
<td>Return label for first non-NA/null value</td>
</tr>
<tr>
<td>DataFrame.last_valid_index()</td>
<td>Return label for last non-NA/null value</td>
</tr>
<tr>
<td>DataFrame.resample(rule[, how, axis, ...])</td>
<td>Convenience method for frequency conversion and resampling of regular time series data.</td>
</tr>
<tr>
<td>DataFrame.to_period([freq, axis, copy])</td>
<td>Convert DataFrame from DatetimeIndex to PeriodIndex with desired method</td>
</tr>
<tr>
<td>DataFrame.to_timestamp([freq, how, axis, copy])</td>
<td>Cast to DatetimeIndex of timestamps, at beginning of period</td>
</tr>
<tr>
<td>DataFrame.tz_convert(tz[, axis, level, copy])</td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td>DataFrame.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
</tbody>
</table>

#### pandas.DataFrame.asfreq

DataFrame.asfreq(freq[, method=method, how=how, normalize=normalize])

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**

- **freq**: DateOffset object, or string
- **method**: {'backfill', 'bfill', 'pad', 'ffill', 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 method
- **how**: {'start', 'end'}, default end
  - For PeriodIndex only, see PeriodIndex.asfreq
- **normalize**: bool, default False
  - Whether to reset output index to midnight

**Returns**

- **converted**: type of caller

#### pandas.DataFrame.shift

DataFrame.shift(periods=periods, freq=freq, axis=0, **kwargs)

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 datetools module or time rule (e.g. ‘EOM’). See Notes.

axis : {0, 1, ‘index’, ‘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.first_valid_index

DataFrame.first_valid_index()
Return label for first non-NA/null value

pandas.DataFrame.last_valid_index

DataFrame.last_valid_index()
Return label for last non-NA/null value

pandas.DataFrame.resample

DataFrame.resample(rule=None, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0)
Convenience method for frequency conversion and resampling of regular time-series data.

Parameters rule : string
the offset string or object representing target conversion

how : string
method for down- or re-sampling, default to ‘mean’ for downsampling

axis : int, optional, default 0

fill_method : string, default None
fill_method for upsampling

closed : {‘right’, ‘left’}
Which side of bin interval is closed

label : {‘right’, ‘left’}
Which bin edge label to label bucket with

convention : {‘start’, ‘end’, ‘s’, ‘e’}

kind : “period”/”timestamp”

loffset : timedelta
Adjust the resampled time labels

limit : int, default None
Maximum size gap to when reindexing with fill_method
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

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_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.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 is a MultiIndex, convert a specific level. Otherwise must be None
copy : boolean, default True
Also make a copy of the underlying data
pandas: powerful Python data analysis toolkit, Release 0.16.2

Raises TypeError
If the axis is tz-naive.

pandas.DataFrame.tz_localize

DataFrame.tz_localize(*args, **kwargs)
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 is 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)
  Attempt to infer fall dst-transition hours based on order

Raises TypeError
If the TimeSeries is tz-aware and tz is not None.

33.4.13 Plotting

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.
DataFrame.plot(data[, x, y, kind, ax, ...]) Make plots of DataFrame using matplotlib / pylab.

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
**pandas.DataFrame.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
- **ax** : matplotlib Axes
- **sharex** : boolean, default False
- **sharey** : boolean, default False
- **figsize** : tuple (width, height) in inches
- **layout** : tuple (optional)
  - (rows, columns) for the layout of the plot
- **return_type** : {'axes', 'dict', 'both'}, default 'dict'
  - The kind of object to return. 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot; 'axes' returns the matplotlib axes the boxplot is drawn on; 'both' returns a namedtuple with the axes and dict.
  - When grouping with by, a dict mapping columns to return_type is returned.
- **kwds** : other plotting keyword arguments to be passed to matplotlib boxplot function

**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.
**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**: (optional) a tuple (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.plot**

DataFrame.plot(data=None, 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.

**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
Title to use for the plot

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)

**layout** : tuple (optional)

(rows, columns) for the layout of the plot

**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
• 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`).

33.4.14 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.DataFrame.from_csv(path[, header, sep, ...])</td>
<td>Read delimited file into DataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.from_dict(data[, orient, dtype])</td>
<td>Construct DataFrame from dict of array-like or dicts</td>
</tr>
<tr>
<td>pandas.DataFrame.from_items(items[, columns, orient])</td>
<td>Convert (key, value) pairs to DataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.from_records(data[, index, ...])</td>
<td>Convert structured or record ndarray to DataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.info(verbosbuf, max_cols, ...)]</td>
<td>Concise summary of a DataFrame.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_pickle(path)</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>pandas.DataFrame.to_csv([path_or_buf, sep, na_rep, ...])</td>
<td>Write DataFrame to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>pandas.DataFrame.to_hdf(path_or_buf, key, **kwargs)</td>
<td>Activate the HDFStore</td>
</tr>
<tr>
<td>pandas.DataFrame.to_sql(name, con[, flavor, ...])</td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_dict(*args, **kwargs)</td>
<td>Convert DataFrame to dictionary.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_excel(excel_writer[, ...])</td>
<td>Write DataFrame to a excel sheet</td>
</tr>
<tr>
<td>pandas.DataFrame.to_json(orientation, ...)]</td>
<td>Convert the object to a JSON string</td>
</tr>
<tr>
<td>pandas.DataFrame.to_html([buf, columns, col_space, ...])</td>
<td>Render a DataFrame as an HTML table.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_latex([buf, columns, ...])</td>
<td>Render a DataFrame to a tabular environment table.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_stata(fname[, convert_dates, ...])</td>
<td>A class for writing Stata binary dta files from array-like objects</td>
</tr>
<tr>
<td>pandas.DataFrame.to_msgpack([path_or_buf])</td>
<td>Msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td>pandas.DataFrame.to_gbq(destination_table[, ...])</td>
<td>Write a DataFrame to a Google BigQuery table.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_records(index, convert_datetime64)</td>
<td>Convert DataFrame to record array.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_sparse(fill_value, kind)</td>
<td>Convert to SparseDataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.to_dense()</td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td>pandas.DataFrame.to_string([buf, columns, ...])</td>
<td>Render a DataFrame to a console-friendly tabular output.</td>
</tr>
<tr>
<td>pandas.DataFrame.to_clipboard([excel, sep])</td>
<td>Attempt to write text representation of object to the system clipboard.</td>
</tr>
</tbody>
</table>

pandas.DataFrame.from_csv

**classmethod** pandas.DataFrame.from_csv(path[, header=0, sep=', ', index_col=0, parse_dates=True, encoding=None, tupleize_cols=False, infer_datetime_format=False)

Read delimited file into DataFrame

**Parameters**

- **path** : string file path or file handle / StringIO
- **header** : int, default 0
  - Row to use at header (skip prior rows)
- **sep** : string, default ‘,’
  - Field delimiter

33.4. DataFrame 1277
pandas: powerful Python data analysis toolkit, Release 0.16.2

**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

**Notes**
- Preferable to use read_table for most general purposes but from_csv makes for an easy roundtrip to and from file, especially with a DataFrame of time series data

**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_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 to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

Returns df : DataFrame

pandas.DataFrame.info

DataFrame.info (verbose=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, 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. Memory usage is shown in human-readable units (base-2 representation).

null_counts : boolean, default None

33.4. DataFrame

1279
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.to_pickle**

`DataFrame.to_pickle(path)`

Pickle (serialize) object to input file path

**Parameters**

- **path**: string
  
  File path

**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, quoting=None, quotechar='', line_terminator='
', chunksize=None, tupleize_cols=False, date_format=None, doublequote=True, escapechar=None, decimal='.', **kwds)`

Write DataFrame to a comma-separated values (csv) file

**Parameters**

- **path_or_buf**: string or file handle, default None
  
  File path or object, if None is provided the result is returned as a string.

- **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 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, 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

- **nanRep**: None
  
  deprecated, use na_rep

- **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.

line_terminator : string, default ‘\n’

The newline character or character sequence to use in the output file

quoting : optional constant from csv module

defaults to csv.QUOTE_MINIMAL

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

write multi_index columns as a list of tuples (if True) or new (expanded format) 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

New in version 0.16.0.

pandas.DataFrame.to_hdf

DataFrame.to_hdf(path_or_buf, key, \*\*kwargs)
activate the HDFStore

Parameters path_or_buf : the path (string) or buffer to put the store

key : string

identifier for the group in the store

mode : optional, {‘a’, ‘w’, ‘r’, ‘r+’}, default ‘a’

‘r’ Read-only; no data can be modified.

‘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

**complevel**: int, 1-9, default 0

If a complib is specified compression will be applied where possible

**complib**: {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

**fletcher32**: bool, default False

If applying compression use the fletcher32 checksum

### pandas.DataFrame.to_sql

`DataFrame.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'
  The flavor of SQL to use. Ignored when using SQLAlchemy engine. ‘mysql’ is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.

- **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.DataFrame.to_dict

DataFrame.to_dict(*args, **kwargs)

Convert DataFrame to dictionary.

**Parameters**

- **orient**: str {'dict', 'list', 'series', 'split', 'records'}
  
  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}]

  Abbreviations are allowed. s indicates `series` and sp indicates `split`.

**Returns**

- **result**: dict like {column -> {index -> value}}

### 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')

Write DataFrame to a 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 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

**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)

**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 = ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

**pandas.DataFrame.to_json**

DataFrame.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

**date_format** : {'epoch', 'iso'}

Type of date conversion. *epoch* = epoch milliseconds, *iso* = ISO8601, 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 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.

**Returns** same type as input object with filtered info axis

**pandas.DataFrame.to_html**

DataFrame.to_html(bu=None, columns=None, col_space=None, colSpace=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)

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.
Parameters frame : DataFrame
    object to render
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
justify : {'left', 'right'}, default None
    Left or right-justify the column labels. If None uses the option from the print config-
    uration (controlled by set_option), ‘right’ out of the box.
index_names : bool, optional
    Prints the names of the indexes, default True
force_unicode : bool, default False
    Always return a unicode result. Deprecated in v0.10.0 as string formatting is now
    rendered to unicode by default.

Returns formatted : string (or unicode, depending on data and options)

pandas.DataFrame.to_latex

DataFrame.to_latex (buf=None, columns=None, col_space=None, colSpace=None, header=True, in-
dex=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=True, longtable=False, escape=True)
Render a DataFrame to a tabular environment table. You can splice this into a LaTeX document. Requires
usepackage{booktabs}.

**to\_latex**-specific options:

- **bold\_rows** [boolean, default True] Make the row labels bold in the output
- **longtable** [boolean, default False] Use a longtable environment instead of tabular. Requires adding a usepackage\{longtable\} to your LaTeX preamble.
- **escape** [boolean, default True] When set to False prevents from escaping latex special characters in column names.

**Parameters**
- **frame** : DataFrame
  object to render
- **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
- **justify** : \{‘left’, ‘right’\}, default None
  Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set\_option), ‘right’ out of the box.
- **index\_names** : bool, optional
  Prints the names of the indexes, default True
- **force\_unicode** : bool, default False
Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

**Returns formatted**: string (or unicode, depending on data and options)

### pandas.DataFrame.to_stata

**DataFrame.to_stata** *(fname=None, convert_dates=None, write_index=True, encoding='latin-1', byte_order=None, time_stamp=None, data_label=None)*

A class for writing Stata binary dta files from array-like objects

**Parameters**

- **fname**: file path or buffer
  - Where to save the dta file.

- **convert_dates**: dict
  - Dictionary mapping column of datetime types to the stata internal format that you want to use for the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either a number or a name.

- **encoding**: str
  - Default is latin-1. Note that Stata does not support unicode.

- **byteorder**: str
  - Can be “>”, “<”, “little”, or “big”. The default is None which uses `sys.byteorder`

**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_msgpack

**DataFrame.to_msgpack** *(path_or_buf=None, **kwargs)*

msgpack (serialize) object to input file path

**Parameters**

- **path_or_buf**: string, 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_gbq

DataFrame.to_gbq(destination_table, project_id=None, chunksize=10000, verbose=True, reauth=False)

Write a DataFrame to a Google BigQuery table.

THIS IS AN EXPERIMENTAL LIBRARY

If the table exists, the dataframe will be written to the table using the defined table schema and column types. For simplicity, this method uses the Google BigQuery streaming API. The to_gbq method chunks data into a default chunk size of 10,000. Failures return the complete error response which can be quite long depending on the size of the insert. There are several important limitations of the Google streaming API which are detailed at: https://developers.google.com/bigquery/streaming-data-into-bigquery.

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.

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
pandas.DataFrame.to_dense

DataFrame.to_dense()
Return dense representation of NDFrame (as opposed to sparse)

pandas.DataFrame.to_string

DataFrame.to_string(buf=None, columns=None, col_space=None, colSpace=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
frame : DataFrame
    object to render

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

justify : {‘left’, ‘right’}, default None
    Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box.

index_names : bool, optional
    Prints the names of the indexes, default True
**force_unicode** : bool, default False

Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

**Returns** formatter : string (or unicode, depending on data and options)

---

**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

---

### 33.5 Panel

#### 33.5.1 Constructor

Panel([data, items, major_axis, minor_axis, ...]) Represents wide format panel data, stored as 3-dimensional array

---

**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

- **major_axis** : Index or array-like

- **minor_axis** : Index or array-like

- **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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>at</code></td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>index(es) of the NDFrame</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]</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>ix</code></td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td><code>loc</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>tuple of axis dimensions</td>
</tr>
<tr>
<td><code>size</code></td>
<td>number of elements in the NDFrame</td>
</tr>
<tr>
<td><code>values</code></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
index(es) of the NDFrame

### pandas.Panel.blocks

Panel.blocks
Internal property, property synonym for as_blocks()

### pandas.Panel.dtypes

Panel.dtypes
Return the dtypes in this object

### pandas.Panel.empty

Panel.empty
True if NDFrame is entirely empty [no items]
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.Panel.iloc

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.

.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.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.

.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

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 upcase to int32.

is_copy

is_copy
Methods

abs()

Return an object with absolute value taken.

Addition of series and other, element-wise (binary operator add).

add(other[, axis])

Concatenate prefix string with panel items names.

add_prefix(prefix)

Concatenate suffix string with panel items names

add_suffix(suffix)

Align two object on their axes with the

align(other[, join, axis, level, copy, ...])

Return whether all elements are True over requested axis

all([axis, bool_only, skipna, level])

Return whether any element is True over requested axis

any([axis, bool_only, skipna, level])

Applies function along input axis of the Panel

apply(func[, axis])

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous

as_blocks()

Convert all TimeSeries inside to specified frequency using DateOffset objects.

as_matrix()

Cast object to input numpy.dtype

astype(dtype[, copy, raise_on_error])

Select values at particular time of day (e.g.,

at_time(time[, asof])

Select values between particular times of the day (e.g., 9:00-9:30 AM)

between_time(start_time, end_time[, ...])

Synonym for NDFrame.fillna(method='bfill')

bfill([axis, inplace, limit, downcast])

Return the bool of a single element PandasObject

bool()

Trim values at input threshold(s)

clip([lower, upper, out, axis])

Return copy of the input with values below given value(s) truncated

clip_lower(threshold[, axis])

Return copy of input with values above given value(s) truncated

clip_upper(threshold[, axis])

Return the compound percentage of the values for the requested axis

compound([axis, skipna, level])

Conform input DataFrame to align with chosen axis pair.

conform(frame[, axis])

Compute NDFrame with “consolidated” internals (data of each dtype grouped together)

consolidate([inplace])

Attempt to infer better dtype for object columns

convert_objects([convert_dates, ...])

Make a copy of this object

copy([deep])

Return number of observations over requested axis.

count([axis])

Return cumulative max over requested axis.

cummax([axis, dtype, out, skipna])

Return cumulative min over requested axis.

cummin([axis, dtype, out, skipna])

Return cumulative prod over requested axis.

cumprod([axis, dtype, out, skipna])

Return cumulative sum over requested axis.

cumsum([axis, dtype, out, skipna])

Generate various summary statistics, excluding NaN values.

describe([percentile_width, percentiles, ...])

Floating division of series and other, element-wise (binary operator truediv).

div(other[, axis])

Floating division of series and other, element-wise (binary operator truediv).

divide(other[, axis])

Return new object with labels in requested axis removed

drop(labels[, axis, level, inplace, errors])

Drop 2D from panel, holding passed axis constant

dropna([axis, how, inplace])

Wrapper for comparison method eq

eq(other)

Determines if two NDFrame objects contain the same elements.

equals(other)

Synonym for NDFrame.fillna(method='ffill')

ffill([axis, inplace, limit, downcast])

Fill NA/NaN values using the specified method

filter([items, like, regex, axis])

Restrict the info axis to set of items or wildcard

first(offset)

Convenience method for setting up initial periods of time series data

floordiv(other[, axis])

Integer division of series and other, element-wise (binary operator floordiv).

fromDict(data[, intersect, orient, dtype])

Construct Panel from dict of DataFrame objects

from_dict(data[, intersect, orient, dtype])

Construct Panel from dict of DataFrame objects

g(f教研)

Wrapper for comparison method ge

gte(other)

Get item from object for given key (DataFrame column, Panel slice, etc.).

get(key[, default])

Return the counts of dtypes in this object

get_dtypes_counts()

Return the counts of ftypes in this object

get_ftype_counts()

get_value(*args, **kwargs)
Table 33.67 – continued from previous page

get_values()
groupby(function[, axis])
gt(other)
head([n])
interpolate(method, axis, limit, inplace, ...)]
isnull()
iteritems()
iterkv(*args, **kwargs)
join(other[, how, lsuffix, rsuffix])
keys()
kurt([axis, skipna, level, numeric_only])
kurtosis([axis, skipna, level, numeric_only])
last(offset)
le(other)
load(path)
lte(other)
mad([axis, skipna, level])
major_xa(key[, copy])
mask(cond[, other, inplace, axis, level, ...])
max([axis, skipna, level, numeric_only])
mean([axis, skipna, level, numeric_only])
median([axis, skipna, level, numeric_only])
min([axis, skipna, level, numeric_only])
minor_xa(key[, copy])
mod(other[, axis])
mul(other[, axis])
multiply(other[, axis])
ne(other)
notnull()
pct_change(periods, fill_method, limit, freq)
pipe(func, *args, **kwargs)
pop(item)
pow(other[, axis])
prod([axis, skipna, level, numeric_only])
product([axis, skipna, level, numeric_only])
raddothers[, axis])
rdiv(other[, axis])
reindex([items, major_axis, minor_axis])
reindex_axis(labels[, axis, method, level, ...])
rename([items, major_axis, minor_axis])
rename_axis(mapper[, axis, copy, inplace])
replace(to_replace, value, inplace, limit, ...)
resample(rule[, how, axis, fill_method, ...])
rfloordiv(other[, axis])
rmmodother[, axis])
rmul(other[, axis])
rpow(other[, axis])
rsub(other[, axis])
rtruediv(other[, axis])
sample([in, frac, replace, weights, ...])
save(path)

same as values (but handles sparseness conversions)
Group data on given axis, returning GroupBy object
Wrapper for comparison method gt
Interpolate values according to different methods.
Return a boolean same-sized object indicating if the values are null
Iterate over (label, values) on info axis
iteritems alias used to get around 2to3. Deprecated
Join items with other Panel either on major and minor axes column
Get the ‘info axis’ (see Indexing for more)
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Convenience method for subsetting final periods of time series data
Wrapper for comparison method le
Deprecated.
Return the mean absolute deviation of the values for the requested axis
Return slice of panel along major axis
Return an object of same shape as self and whose corresponding entries are from
This method returns the maximum of the values in the object.
Return the mean of the values for the requested axis
Return the median of the values for the requested axis
This method returns the minimum of the values in the object.
Return slice of panel along minor axis
Modulo of series and other, element-wise (binary operator mod).
Multiplication of series and other, element-wise (binary operator mul).
Multiplication of series and other, element-wise (binary operator mul).
Wrapper for comparison method ne
Return a boolean same-sized object indicating if the values are
Percent change over given number of periods.
Apply func(self, *args, **kwargs)
Return item and drop from frame.
Exponential power of series and other, element-wise (binary operator pow).
Return the product of the values for the requested axis
Return the product of the values for the requested axis
Addition of series and other, element-wise (binary operator radd).
Floating division of series and other, element-wise (binary operator rtruediv).
Conform Panel to new index with optional filling logic, placing NA/NaN in locations
Conform input object to new index with optional filling logic, placing NA/NaN in locations
return an object with matching indicies to myself
Alter axes input function or functions.
Alter index and / or columns using input function or functions.
Replace values given in ‘to_replace’ with ‘value’.
Convenience method for frequency conversion and resampling of regular time-series
Integer division of series and other, element-wise (binary operator rflooridiv).
Modulo of series and other, element-wise (binary operator rmod).
Multiplication of series and other, element-wise (binary operator rmul).
Exponential power of series and other, element-wise (binary operator rpow).
Subtraction of series and other, element-wise (binary operator rsub).
Floating division of series and other, element-wise (binary operator rtruediv).
Returns a random sample of items from an axis of object.
Deprecated.
select(crit[, axis])  Return data corresponding to axis labels matching criteria
sem(axis, skipna, level, ddof, numeric_only)  
set_axis(axis, labels)  public version of axis assignment
set_value(*args, **kwargs)  Quickly set single value at (item, major, minor) location
shift(*args, **kwargs)  Shift index by desired number of periods with an optional time freq.
skew((axis, skipna, level, numeric_only))  Return unbiased skew over requested axis
slice_shift((periods, axis))  Equivalent to shift without copying data.
sort_index((axis, ascending))  Sort object by labels (along an axis)
squeeze()  squeeze length 1 dimensions
std((axis, skipna, level, ddof, numeric_only))  Return unbiased standard deviation over requested axis.
subtract(other[, axis])  Subtraction of series and other, element-wise (binary operator sub).
sum((axis, skipna, level, numeric_only))  Subtraction of series and other, element-wise (binary operator sub).
swap_axes(axis1, axis2[, copy])  Return the sum of the values for the requested axis
swap_level(i, j[, axis])  Interchange axes and swap values axes appropriately

tail([n])  
take(indices[, axis, convert, is_copy])  Analogous to ndarray.take
to_long(*args, **kwargs)  Attempt to write text representation of object to the system clipboard This can be

to_clipboard([excel, sep])  Return dense representation of NDFrame (as opposed to sparse)
to_dense()  Write each DataFrame in Panel to a separate excel sheet

to_excel(path[, na_rep, engine])  Transform wide format into long (stacked) format as DataFrame whose columns

to_frame([filter_observations])  activate the HDFStore

to_hdf(path_or_buf, key, **kwargs)  Convert the object to a JSON string.
to_json([path_or_buf, orient, date_format, ...])  
to_msgpack([path_or_buf])  msgpack (serialize) object to input file path

to_pickle(path)  Pickle (serialize) object to input file path

to_sparse([fill_value, kind])  Convert to SparsePanel

to_sql(name, con[, flavor, schema, ...])  Write records stored in a DataFrame to a SQL database.

transpose(*args, **kwargs)  Permute the dimensions of the Panel

truediv(other[, axis])  Floating division of series and other, element-wise (binary operator truediv).

truncate([before, after, axis, copy])  Truncates a sorted NDFrame before and/or after some particular dates.

tshift([periods, freq, axis])  Convert tz-aware axis to target time zone.

tz_convert(tz[, axis, level, copy])  Localize tz-naive TimeSeries to target time zone

tz_localize(*args, **kwargs)  Modify Panel in place using non-NA values from passed Panel, or object coercible

update(other[, join, overwrite, ...])  Return unbiased variance over requested axis.

var((axis, skipna, level, ddof, numeric_only))  Return an object of same shape as self and whose corresponding entries are from

where(cond[, other, inplace, axis, level, ...])  Return slice of panel along selected axis

xs(key[, axis, copy])  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.align**

Panel.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)

Align two object on their axes with the specified join method for each axis Index

Parameters  **other** : DataFrame or Series

   **join** : {'outer', 'inner', 'left', 'right'}, default 'outer'

   **axis** : allowed axis of the other object, default None
   
   Align on index (0), columns (1), or both (None)

   **level** : int or level name, default None
   
   Broadcast across a level, matching Index values on the passed MultiIndex level

   **copy** : 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.Na
   
   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, 1}, default 0
Filling axis, method and limit

Returns (left, right) : (type of input, type of other)
Aligned objects

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
  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
  bool_only : boolean, default None
  Include only boolean data. If None, will attempt to use everything, then use only
  boolean data

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
  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
  bool_only : boolean, default None
  Include only boolean data. If None, will attempt to use everything, then use only
  boolean data

Returns any : DataFrame or Panel (if level specified)

pandas.Panel.apply

Panel.apply (func, axis='major', **kwargs)
Applies function along input axis of the Panel

Parameters func : function
Function to apply to each combination of ‘other’ axes e.g. if axis = ‘items’, then the combination of major_axis/minor_axis will be passed a Series

axis : {'major', 'minor', 'items'}

Additional keyword arguments will be passed as keywords to the function

Returns result : Pandas Object

Examples

```python
>>> p.apply(numpy.sqrt)  # returns a Panel
>>> p.apply(lambda x: x.sum(), axis=0)  # equiv to p.sum(0)
>>> p.apply(lambda x: x.sum(), axis=1)  # equiv to p.sum(1)
>>> p.apply(lambda x: x.sum(), axis=2)  # equiv to p.sum(2)
```

pandas.Panel.as_blocks

Panel.as_blocks()

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Returns values : a dict of dtype -> Constructor Types

pandas.Panel.as_matrix

Panel.as_matrix()

pandas.Panel.asfreq

Panel.asfreq(freq=, method=None, how=None, normalize=False)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters freq : DateOffset object, or string

method : {'backfill', 'bfill', 'pad', 'ffill', 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 method

how : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

normalize : bool, default False

Whether to reset output index to midnight

Returns converted : type of caller
pandas.Panel.astype

Panel.astype (dtype, copy=True, raise_on_error=True, **kwargs)
Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters
dtype : numpy.dtype or Python type
raise_on_error : raise on invalid input
kwargs : keyword arguments to pass on to the constructor

Returns
casted : type of caller

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 NDFrame.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, out=None, axis=None)
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.

Returns clipped : Series

Examples

>>> 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  -0.3
 1  -0.2
 2  -0.1
 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)

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.

Returns clipped : same type as input

See also:
clip
pandas.Panel.clip_upper

Panel.clip_upper(threshold, axis=None)
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.

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
Excluding 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

numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then
use only numeric data

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)
Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndar-ray). Mainly an internal API function, but available here to the savvy user
**Parameters inplace**: boolean, default False

If False return new object, otherwise modify existing object

**Returns consolidated**: type of caller

---

**pandas.Panel.convert_objects**

`Panel.convert_objects(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)`

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

---

**pandas.Panel.copy**

`Panel.copy(deep=True)`

Make a copy of this object

**Parameters deep**: boolean or string, default True

Make a deep copy, i.e. also copy 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, dtype=None, out=None, skipna=True, **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 max : DataFrame

pandas.Panel.cummin

Panel.cummin (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative min 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 min : DataFrame

pandas.Panel.cumprod

Panel.cumprod (axis=None, dtype=None, out=None, skipna=True, **kwargs)
Return cumulative prod 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 prod : DataFrame

pandas.Panel.cumsum

Panel.cumsum (axis=None, dtype=None, out=None, skipna=True, **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 sum : DataFrame

pandas.Panel.describe

Panel.describe (percentile_width=None, percentiles=None, include=None, exclude=None)
Generate various summary statistics, excluding NaN values.

Parameters percentile_width : float, deprecated

The percentile_width argument will be removed in a future version. Use percentiles instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

percentiles : array-like, optional
The percentiles to include in the output. Should all be in the interval [0, 1]. By default *percentiles* is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

**include, exclude** : list-like, ‘all’, or None (default)

Specify the form of the returned result. Either:

- None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
- A list of dtypes or strings to be included/excluded. To select all numeric types use *numpy* *numpy.number*. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
- If include is the string ‘all’, the output column-set will match the input one.

**Returns** summary: NDFrame of summary statistics

See also:

DataFrame.select_dtypes

**Notes**

The output DataFrame index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the *count* and *most common* pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.

**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}

Returns
Panel

See also:
Panel.rtruediv

pandas.Panel.drop

Panel.drop(labels, axis=0, level=None, inplace=False, errors='raise')
Return new object with labels in requested axis removed

Parameters
labels : single label or list-like
axis : int or axis name
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.
New in version 0.16.1.

Returns
dropped : type of caller

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)
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 NDFrame.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.

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

pandas.Panel.filter

Panel.filter(items=None, like=None, regex=None, axis=None)
Restrict the info axis to set of items or wildcard

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 None
The axis to filter on. By default this is the info axis. The “info axis” is the axis that is used when indexing with []. For example, df = DataFrame({‘a’: [1, 2, 3, 4]}); df[‘a’]. So, the DataFrame columns are the info axis.

Notes

Arguments are mutually exclusive, but this is not checked for

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.last(‘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.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.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)
   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
   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
      Mapping function for chosen access
pandas: powerful Python data analysis toolkit, Release 0.16.2

axis : {'major', 'minor', 'items'}, default 'major'

Returns  grouped : PanelGroupBy

pandas.Panel.gt

Panel.gt(other)
Wrapper for comparison method gt

pandas.Panel.head

Panel.head(n=5)

pandas.Panel.interpolate

Panel.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast=None, **kwargs)
Interpolate values according to different methods.

Parameters method : {'linear', 'time', 'index', 'values', 'nearest', 'zero',
'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline'
'piecewise_polynomial', 'pchip'}

• 'linear': ignore the index and treat the values as equally spaced. 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 with the order given both 'poly-
nomial' and 'spline' require that you also specify and order (int) e.g.
df.interpolate(method='polynomial', order=4)
• 'krogh', ‘piecewise_polynomial’, ‘spline’, and ‘pchip’ are all
wrappers around the scipy interpolation methods of similar
names. See the scipy documentation for more on their behavior:
http://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-

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.

inplace : bool, default False
Update the NDFrame in place if possible.

downcast : optional, ‘infer’ or None, defaults to None
Downcast dtypes if possible.
**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.isnull**

Panel.isnull()

Return a boolean same-sized object indicating if the values are null

**See also:**
notnull  boolean inverse of isnull

**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.iterkv**

Panel.iterkv(*args, **kwargs)

iteritems alias used to get around 2to3. Deprecated

**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

\textit{rsuffix} : string

Suffix to use from right frame’s overlapping columns

**Returns** \textit{joined} : Panel

\texttt{pandas.Panel.keys}

\texttt{Panel.keys()}

Get the ‘info axis’ (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

\texttt{pandas.Panel.kurt}

\texttt{Panel.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)}

Return unbiased kurtosis over requested axis using Fishers definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns** \textit{kurt} : DataFrame or Panel (if level specified)

\texttt{pandas.Panel.kurtosis}

\texttt{Panel.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)}

Return unbiased kurtosis over requested axis using Fishers definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**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, 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 data. If None, will attempt to use everything, then use only numeric data
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)

Wrapper for comparison method le

**pandas.Panel.load**

Panel.load(path)

Deprecated. Use read_pickle instead.

**pandas.Panel.lt**

Panel.lt(other)

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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns `mad` : DataFrame or Panel (if level specified)
pandas.Panel.major_xs

Panel.major_xs (key, copy=None)
Return slice of panel along major axis

Parameters key : object
    Major axis label
    
copy : boolean [deprecated]
    Whether to make a copy of the data

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 it is a superset of major_xs functionality, see MultiIndex Slicers

pandas.Panel.mask

Panel.mask (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)
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 or array

other : scalar or NDFrame

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

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)

Returns wh : same type as caller

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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **max**: DataFrame or Panel (if level specified)

```
pandas.Panel.mean
```

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **mean**: DataFrame or Panel (if level specified)

```
pandas.Panel.median
```

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**
- `min`: DataFrame or Panel (if level specified)

pandas.Panel.minor_xs

`Panel.minor_xs(key, copy=None)`

Return slice of panel along minor axis

**Parameters**
- `key`: object
  - Minor axis label
- `copy`: boolean [deprecated]
  - Whether to make a copy of the data

**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 it 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}

**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)
Wrapper for comparison method ne

**pandas.Panel.null**

Panel.null()
Return a boolean same-sized object indicating if the values are not null

See also:

isnull boolean inverse of notnull
pandas: powerful Python data analysis toolkit, Release 0.16.2

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)
New in version 0.16.2.

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**: positional arguments passed into func.
- **kwargs**: 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 on Series or DataFrames. Instead of writing

```python
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write
>>> (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 \( \text{arg2} \):

>>> (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.

**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(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the product 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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns prod : DataFrame or Panel (if level specified)
pandas.Panel.product

```
Panel.product(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
```

Return the product 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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **prod**: DataFrame or Panel (if level specified)

pandas.Panel.radd

```
Panel.radd(other, axis=0)
```

Addition of series and other, element-wise (binary operator radd). Equivalent to `other + panel`

**Parameters**

- **other**: DataFrame or Panel
  - axis : {items, major_axis, minor_axis}
    - Axis to broadcast over

**Returns**

- **Panel**

See also:

- `Panel.add`

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**

Panel.reindex(items=None, major_axis=None, minor_axis=None, **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 (can be specified in order, or as 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:
    - 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 size gap to forward or backward fill

**Returns**

reindexed : Panel

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

**pandas.Panel.reindex_axis**

Panel.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=np.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 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 size gap to forward or backward fill

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)

return an object with matching indicies to myself

Parameters other : Object

- method : string or None
- copy : boolean, default True
- limit : int, default None

Maximum size gap to forward or backward fill

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.

Parameters items, major_axis, minor_axis : dict-like or function, optional
Transformation 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.

Returns renamed : Panel (new object)

pandas.Panel.rename_axis

Panel.rename_axis(mapper, axis=0, copy=True, inplace=False)
Alter index and / or columns using 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.

Parameters mapper : dict-like or function, optional
axis : int or string, default 0

copy : boolean, default True
Also copy underlying data

inplace : boolean, default False

Returns renamed : type of caller

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 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 form 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.Panel.resample`

```python
Panel.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0)
```

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters**

- `rule`: string
  - the offset string or object representing target conversion

- `how`: string
  - method for down- or re-sampling, default to ‘mean’ for downsampling

- `axis`: int, optional, default 0

- `fill_method`: string, default None
  - fill_method for upsampling

- `closed`: ‘right’, ‘left’
  - Which side of bin interval is closed

- `label`: ‘right’, ‘left’
  - Which bin edge label to label bucket with

- `convention`: ‘start’, ‘end’, ‘s’, ‘e’

- `kind`: “period”/”timestamp”

- `loffset`: timedelta
  - Adjust the resampled time labels

- `limit`: int, default None
  - Maximum size gap to when reindexing with fill_method

- `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
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
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.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}
Axis to broadcast over

**Returns** Panel

**See also:**
Panel.pow

### 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}
  
**Returns** Panel

**See also:**
Panel.sub

### 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}
  
**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.

New in version 0.16.1.

**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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

pandas.Panel.save

Panel.save(path)
Deprecated. Use to_pickle instead

pandas.Panel.select

Panel.select(crit, axis=0)
Return data corresponding to axis labels matching criteria

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

numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data
**pandas**: powerful Python data analysis toolkit, Release 0.16.2

**Returns** `sem` : DataFrame or Panel (if level specified)

**pandas.Panel.set_axis**

`Panel.set_axis(axis, labels)`  
public version of axis assignment

**pandas.Panel.set_value**

`Panel.set_value(*args, **kwargs)`  
Quickly set single value at (item, major, minor) location

**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(*args, **kwargs)`  
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, **kw vals)`  
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, 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 data. If None, will attempt to use everything, then use only numeric data

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, ascending=True)
Sort object by labels (along an axis)

Parameters axis : {0, 1}
Sort index/rows versus columns

ascending : boolean, default True
Sort ascending vs. descending

Returns sorted_obj : type of caller

pandas.Panel.squeeze

Panel.squeeze ()
squeeze length 1 dimensions

pandas.Panel.std

Panel.std (axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)
Return unbiased 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

numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns  std : DataFrame or Panel (if level specified)

pandas.Panel.sub

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}
Axis to broadcast over

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}
Axis to broadcast over

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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**  
sum : DataFrame or Panel (if level specified)

**pandas.Panel.swapaxes**

```python
Panel.swapaxes(axis1, axis2, copy=True)
```
Interchange axes and swap values axes appropriately

**Returns**  
y : same as input

**pandas.Panel.swaplevel**

```python
Panel.swaplevel(i, j, 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)

**pandas.Panel.tail**

```python
Panel.tail(n=5)
```

**pandas.Panel.take**

```python
Panel.take(indices, axis=0, convert=True, is_copy=True)
```
Analogous to ndarray.take

**Parameters**  
indices : list / array of ints
axis : int, default 0
convert : translate neg to pos indices (default)
is_copy : mark the returned frame as a copy

**Returns**  
taken : type of caller

**pandas.Panel.toLong**

```python
Panel.toLong(*args, **kwargs)
```

**pandas.Panel.to_clipboard**

```python
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.

**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.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

**Returns**

- **y** : DataFrame

### pandas.Panel.to_hdf

**Panel.to_hdf**(path_or_buf, key, **kwargs)

activate the HDFStore

**Parameters**

- **path_or_buf** : the path (string) or buffer to put the store
- **key** : string
- **mode** : optional, {‘a’, ‘w’, ‘r’, ‘r+’}, default ‘a’
  - ‘r’ : Read-only; no data can be modified.
  - ‘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

- **complevel** : int, 1-9, default 0
If a complib is specified compression will be applied where possible

```complib```: {'zlib', 'bzip2', 'lz4', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

```fletcher32```: bool, default False

If applying compression use the fletcher32 checksum

---

**pandas.Panel.to_json**

```Panel.to_json```

```Panel.to_json``(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

- **date_format**: {'epoch', 'iso'}
  - Type of date conversion. *epoch* = epoch milliseconds, *iso* = ISO8601, 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 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.

**Returns** same type as input object with filtered info axis

**pandas.Panel.to_long**

Panel.to_long(*args, **kwargs)

**pandas.Panel.to_msgpack**

Panel.to_msgpack(path_or_buf=None, **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)

Pickle (serialize) object to input file path

**Parameters**

path : string
    File path

**pandas.Panel.to_sparse**

Panel.to_sparse(fill_value=None, kind='block')

Convert to SparsePanel

**Parameters**

fill_value : float, default NaN

kind : {'block', 'integer'}

**Returns**

y : SparseDataFrame

**pandas.Panel.to_sql**

Panel.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'

The flavor of SQL to use. Ignored when using SQLAlchemy engine. ‘mysql’ is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.

**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.transpose**

**Panel.transpose**(*args, **kwargs)

Permute the dimensions of the Panel

Parameters

**args** : three positional arguments: each one of

{0, 1, 2, ‘items’, ‘major_axis’, ‘minor_axis’}

**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)
```

```python
>>> 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 `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.truncate**

`Panel.truncate(before=None, after=None, axis=None, copy=True)`

Truncates a sorted NDFrame before and/or after some particular dates.

**Parameters**
- `before`: date
- `after`: date
- `axis`: the truncation axis, defaults to the stat axis
- `copy`: boolean, default is True,

**Returns**
- `truncated`: type of caller

**pandas.Panel.tshift**

`Panel.tshift(periods=1, freq=None, axis='major', **kwds)`

**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 is 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**(*args, **kwargs*)
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 is 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)
  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, join='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
- **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
- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **var**: DataFrame or Panel (if level specified)

### pandas.Panel.where

**Panel.where**(cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)

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 or array
- **other**: scalar or NDFrame
- **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
- **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)

**Returns**

- **wh**: same type as caller
pandas.Panel.xs

Panel.xs(key, axis=1, copy=None)
Return slice of panel along selected axis

Parameters  key : object
    Label
    axis : {'items', 'major', 'minor'}, default 1/'major'
    copy : boolean [deprecated]
        Whether to make a copy of the data

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 it is a superset of xs functionality, see MultiIndex Slicers

33.5.2 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

<table>
<thead>
<tr>
<th>Panel.values</th>
<th>Numpy representation of NDFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.axes</td>
<td>index(es) of the NDFrame</td>
</tr>
<tr>
<td>Panel.ndim</td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td>Panel.size</td>
<td>number of elements in the NDFrame</td>
</tr>
<tr>
<td>Panel.shape</td>
<td>tuple of axis dimensions</td>
</tr>
<tr>
<td>Panel.dtypes</td>
<td>Return the dtypes in this object</td>
</tr>
<tr>
<td>Panel.ftypes</td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<tr>
<td>Panel.get_dtypes_counts()</td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>Panel.get_ftypes_counts()</td>
<td>Return the counts of ftypes in this object.</td>
</tr>
</tbody>
</table>

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 upcase to int32.

pandas.Panel.axes

Panel.axes
index(es) of the NDFrame

pandas.Panel.ndim

Panel.ndim
Number of axes / array dimensions

pandas.Panel.size

Panel.size
number of elements in the NDFrame

pandas.Panel.shape

Panel.shape
tuple of axis dimensions

pandas.Panel.dtypes

Panel.dtypes
Return the dtypes in this object

pandas.Panel.ftypes

Panel.ftypes
Return the ftypes (indication of sparse/dense and dtype) in this 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
33.5.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.astype</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>Panel.copy</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>Panel.isnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>Panel.notnull()</td>
<td>Return a boolean same-sized object indicating if the values are not null</td>
</tr>
</tbody>
</table>

**pandas.Panel.astype**

```python
Panel.astype(dtype[, copy, raise_on_error])
```

Cast object to input numpy.dtype

- **Parameters**
  - `dtype`: numpy.dtype or Python type
  - `copy`: boolean, default True
  - `raise_on_error`: boolean, default True
  - `**kwargs`: keyword arguments to pass on to the constructor

- **Returns**
  - `casted`: type of caller

**pandas.Panel.copy**

```python
Panel.copy([deep])
```

Make a copy of this object

- **Parameters**
  - `deep`: boolean or string, default True
    - Make a deep copy, i.e. also copy data

- **Returns**
  - `copy`: type of caller

**pandas.Panel.isnull**

```python
Panel.isnull()
```

Return a boolean same-sized object indicating if the values are null

**See also:**

- `notnull` boolean inverse of isnull

**pandas.Panel.notnull**

```python
Panel.notnull()
```

Return a boolean same-sized object indicating if the values are not null

**See also:**

- `isnull` boolean inverse of notnull

33.5.4 Getting and setting
pandas: powerful Python data analysis toolkit, Release 0.16.2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.get_value(*args, **kwargs)</td>
<td>Quickly retrieve single value at (item, major, minor) location</td>
</tr>
<tr>
<td>Panel.set_value(*args, **kwargs)</td>
<td>Quickly set single value at (item, major, minor) location</td>
</tr>
</tbody>
</table>

### pandas.Panel.get_value

#### 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.set_value

#### 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

### 33.5.5 Indexing, iteration, slicing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.at</td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td>Panel.iat</td>
<td>Fast integer location scalar accessor.</td>
</tr>
<tr>
<td>Panel.ix</td>
<td>A primarily label-location based indexer, with integer position fallback.</td>
</tr>
<tr>
<td>Panel.loc</td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td>Panel.iloc</td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td>Panel.<strong>iter</strong>()</td>
<td>Iterate over info axis</td>
</tr>
<tr>
<td>Panel.iteritems()</td>
<td>Iterate over (label, values) on info axis</td>
</tr>
<tr>
<td>Panel.pop(item)</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>Panel.xs(key[, axis, copy])</td>
<td>Return slice of panel along selected axis</td>
</tr>
<tr>
<td>Panel.major_xs(key[, copy])</td>
<td>Return slice of panel along major axis</td>
</tr>
<tr>
<td>Panel.minor_xs(key[, copy])</td>
<td>Return slice of panel along minor axis</td>
</tr>
</tbody>
</table>

### pandas.Panel.at

Panel.at
- Fast label-based scalar accessor
Similarly to \texttt{loc}, \texttt{at} provides \textbf{label} based scalar lookups. You can also set using these indexers.

\textbf{pandas.Panel.iat}

\texttt{Panel.iat}

Fast integer location scalar accessor.

Similarly to \texttt{iloc.iat} provides \textbf{integer} based lookups. You can also set using these indexers.

\textbf{pandas.Panel.ix}

\texttt{Panel.ix}

A primarily label-location based indexer, with integer position fallback.

\texttt{.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.

\texttt{.ix} is the most general indexer and will support any of the inputs in \texttt{.loc} and \texttt{.iloc}. \texttt{.ix} also supports floating point label schemes. \texttt{.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 \texttt{.iloc} or \texttt{.loc}.

See more at \textit{Advanced Indexing}.

\textbf{pandas.Panel.loc}

\texttt{Panel.loc}

Purely label-location based indexer for selection by label.

\texttt{.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 \textbf{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, \textbf{both} the start and the stop are included!).
- A boolean array.

\texttt{.loc} will raise a \texttt{KeyError} when the items are not found.

See more at \textit{Selection by Label}.

\textbf{pandas.Panel.iloc}

\texttt{Panel.iloc}

Purely integer-location based indexing for selection by position.

\texttt{.iloc[]} is primarily integer position based (from 0 to \texttt{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.

.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.__iter__

Panel.__iter__() Iterate over info axis

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.pop

Panel.pop(item) Return item and drop from frame. Raise KeyError if not found.

pandas.Panel.xs

Panel.xs(key, axis=1, copy=None) Return slice of panel along selected axis

Parameters key : object

Label

axis : {'items', 'major', 'minor'}, default 1/'major'

copy : boolean [deprecated]

Whether to make a copy of the data

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 it is a superset of xs functionality, see MultiIndex Slicers
pandas.Panel.major_xs

Panel.major_xs(key, copy=None)
Return slice of panel along major axis

Parameters  key : object
    Major axis label
    copy : boolean [deprecated]
        Whether to make a copy of the data

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 it is a superset of major_xs functionality, see MultiIndex Slicers

pandas.Panel.minor_xs

Panel.minor_xs(key, copy=None)
Return slice of panel along minor axis

Parameters  key : object
    Minor axis label
    copy : boolean [deprecated]
        Whether to make a copy of the data

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 it is a superset of minor_xs functionality, see MultiIndex Slicers

For more information on .at, .iat, .ix, .loc, and .iloc, see the indexing documentation.

33.5.6 Binary operator functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.add</td>
<td>Addition of series and other, element-wise (binary operator add).</td>
</tr>
<tr>
<td>Panel.sub</td>
<td>Subtraction of series and other, element-wise (binary operator sub).</td>
</tr>
<tr>
<td>Panel.mul</td>
<td>Multiplication of series and other, element-wise (binary operator mul).</td>
</tr>
<tr>
<td>Panel.div</td>
<td>Floating division of series and other, element-wise (binary operator truediv).</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 33.72 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.truediv(other[, axis])</td>
<td>Floating division of series and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td>Panel.floordiv(other[, axis])</td>
<td>Integer division of series and other, element-wise (binary operator floordiv).</td>
</tr>
<tr>
<td>Panel.mod(other[, axis])</td>
<td>Modulo of series and other, element-wise (binary operator mod).</td>
</tr>
<tr>
<td>Panel.pow(other[, axis])</td>
<td>Exponential power of series and other, element-wise (binary operator pow).</td>
</tr>
<tr>
<td>Panel.radd(other[, axis])</td>
<td>Addition of series and other, element-wise (binary operator radd).</td>
</tr>
<tr>
<td>Panel.rsub(other[, axis])</td>
<td>Subtraction of series and other, element-wise (binary operator rsub).</td>
</tr>
<tr>
<td>Panel.rmul(other[, axis])</td>
<td>Multiplication of series and other, element-wise (binary operator rmul).</td>
</tr>
<tr>
<td>Panel.rdiv(other[, axis])</td>
<td>Floating division of series and other, element-wise (binary operator rtruediv).</td>
</tr>
<tr>
<td>Panel.rfloordiv(other[, axis])</td>
<td>Integer division of series and other, element-wise (binary operator rfloordiv).</td>
</tr>
<tr>
<td>Panel.rmod(other[, axis])</td>
<td>Modulo of series and other, element-wise (binary operator rmod).</td>
</tr>
<tr>
<td>Panel.rpow(other[, axis])</td>
<td>Exponential power of series and other, element-wise (binary operator rpow).</td>
</tr>
<tr>
<td>Panel.lt(other)</td>
<td>Wrapper for comparison method lt</td>
</tr>
<tr>
<td>Panel.gt(other)</td>
<td>Wrapper for comparison method gt</td>
</tr>
<tr>
<td>Panel.le(other)</td>
<td>Wrapper for comparison method le</td>
</tr>
<tr>
<td>Panel.ge(other)</td>
<td>Wrapper for comparison method ge</td>
</tr>
<tr>
<td>Panel.ne(other)</td>
<td>Wrapper for comparison method ne</td>
</tr>
<tr>
<td>Panel.eq(other)</td>
<td>Wrapper for comparison method eq</td>
</tr>
</tbody>
</table>

#### 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.sub

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}

Axis to broadcast over

**Returns**

Panel

**See also:**

Panel.rsub

#### 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.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.truediv

Panel.truediv(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.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.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.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.radd

Panel.radd(other, axis=0)
Addition of series and other, element-wise (binary operator radd). Equivalent to other + panel.

Parameters other : DataFrame or Panel
axis : {items, major_axis, minor_axis}
Axis to broadcast over

Returns Panel

See also: Panel.add

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.sub

**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.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.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.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

**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.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}
  - Axis to broadcast over

**Returns**

Panel

**See also:**

Panel.pow

### pandas.Panel.lt

**Panel.lt**(other)

Wrapper for comparison method `lt`

### pandas.Panel.gt

**Panel.gt**(other)

Wrapper for comparison method `gt`
pandas.Panel

Panel.le(other)
Wrapper for comparison method le

Panel.ge(other)
Wrapper for comparison method ge

Panel.ne(other)
Wrapper for comparison method ne

Panel.eq(other)
Wrapper for comparison method eq

33.5.7 Function application, GroupBy

pandas.Panel.apply

Panel.apply(func[, axis])
Applies function along input axis of the Panel

Parameters

- func : function
  Function to apply to each combination of ‘other’ axes e.g. if axis = ‘items’, then the combination of major_axis/minor_axis will be passed a Series

- axis : {'major', 'minor', 'items'}

Additional keyword arguments will be passed as keywords to the function

Returns

result : Pandas Object

Examples

>>> p.apply(numpy.sqrt) # returns a Panel
>>> p.apply(lambda x: x.sum(), axis=0) # equiv to p.sum(0)
>>> p.apply(lambda x: x.sum(), axis=1) # equiv to p.sum(1)
>>> p.apply(lambda x: x.sum(), axis=2) # equiv to p.sum(2)
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.Panel.groupby

Panel.groupby (function, axis='major')
Group data on given axis, returning GroupBy object

Parameters:
- function: callable
  Mapping function for chosen access
- axis: {'major', 'minor', 'items'}, default 'major'

Returns:
- grouped: PanelGroupBy

33.5.8 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.abs()</td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td>Panel.clip(lower=None, upper=None, out=None, axis=None)</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>Panel.clip_lower(threshold[, axis])</td>
<td>Return copy of the input with values below given value(s) truncated</td>
</tr>
<tr>
<td>Panel.clip_upper(threshold[, axis])</td>
<td>Return copy of input with values above given value(s) truncated</td>
</tr>
<tr>
<td>Panel.count(axis)</td>
<td>Return number of observations over requested axis.</td>
</tr>
<tr>
<td>Panel.cummax(axis, dtype, out, skipna)</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>Panel.cummin(axis, dtype, out, skipna)</td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td>Panel.cumprod(axis, dtype, out, skipna)</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>Panel.cumsum(axis, dtype, out, skipna)</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>Panel.max(axis, skipna, level, numeric_only)</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>Panel.mean(axis, skipna, level, numeric_only)</td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td>Panel.median(axis, skipna, level, numeric_only)</td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td>Panel.min(axis, skipna, level, numeric_only)</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td>Panel.pct_change(periods, fill_method, ...)</td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td>Panel.prod(axis, skipna, level, numeric_only)</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>Panel.sem(axis, skipna, level, ddof, ...)</td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td>Panel.skew(axis, skipna, level, numeric_only)</td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td>Panel.sum(axis, skipna, level, numeric_only)</td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td>Panel.std(axis, skipna, level, ddof, ...)</td>
<td>Return unbiased standard deviation over requested axis.</td>
</tr>
<tr>
<td>Panel.var(axis, skipna, level, ddof, ...)</td>
<td>Return unbiased variance over requested 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.clip

Panel.clip(lower=None, upper=None, out=None, axis=None)

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.
Returns clipped : Series

Examples

```python
gf
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
```n
gf.clip(-1.0, 0.5)
```python
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
```n
t
```python
0 -0.3
1 -0.2
2 -0.1
3 0.0
4 0.1
dtype: float64
```n
gf.clip(t, t + 1, axis=0)
```python
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
```n

pandas.Panel.clip_lower

Panel.clip_lower (threshold, axis=None)
`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.

Returns clipped : same type as input

See also:
clip

pandas.Panel.clip_upper

Panel.clip_upper (threshold, axis=None)
`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.

**Returns** clipped : same type as input

**See also:**
clip

**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, dtype=None, out=None, skipna=True, **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** max : DataFrame

**pandas.Panel.cummin**

*Panel.cummin*(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative min 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** min : DataFrame

**pandas.Panel.cumprod**

*Panel.cumprod*(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative prod 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** prod : DataFrame
`pandas.Panel.cumsum`  
Panel.cumsum(axis=None, dtype=None, out=None, skipna=True, **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**  
sum : DataFrame

`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, 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 data. If None, will attempt to use everything, then use only numeric data  

**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, 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 data. If None, will attempt to use everything, then use only numeric data  

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **min**: DataFrame or Panel (if level specified)

**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.prod

Panel.prod(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return the product 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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns prod : DataFrame or Panel (if level specified)

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

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns sem : DataFrame or Panel (if level specified)
**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, 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 data. If None, will attempt to use everything, then use only numeric data

- **Returns**
  - **skew**: DataFrame or Panel (if level specified)

**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, 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 data. If None, will attempt to use everything, then use only numeric data

- **Returns**
  - **sum**: DataFrame or Panel (if level specified)

**Pandas.Panel.std**

Panel.std (axis=None, skipna=None, level=None, numeric_only=None, ddof=1, **kwargs)

Return unbiased 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

- **Returns**
  - **std**: DataFrame or Panel (if level specified)
numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns std : DataFrame or Panel (if level specified)

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

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns var : DataFrame or Panel (if level specified)

33.5.9 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.add_prefix(prefix)</td>
<td>Concatenate prefix string with panel items names.</td>
</tr>
<tr>
<td>Panel.add_suffix(suffix)</td>
<td>Concatenate suffix string with panel items names.</td>
</tr>
<tr>
<td>Panel.drop(labels[, axis, level, inplace, ...])</td>
<td>Return new object with labels in requested axis removed.</td>
</tr>
<tr>
<td>Panel.equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements.</td>
</tr>
<tr>
<td>Panel.filter([items, like, regex, axis])</td>
<td>Convenience method for subsetting initial periods of time series data.</td>
</tr>
<tr>
<td>Panel.first(offset)</td>
<td>Convenience method for subsetting final periods of time series data.</td>
</tr>
<tr>
<td>Panel.last(offset)</td>
<td>Convenience method for subsetting final periods of time series data.</td>
</tr>
<tr>
<td>Panel.reindex([items, major_axis, minor_axis])</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>Panel.reindex_axis(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>Panel.reindex_like(other[, method, copy, limit])</td>
<td>Return an object with matching indicies to myself.</td>
</tr>
<tr>
<td>Panel.rename([items, major_axis, minor_axis])</td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td>Panel.sample([n, frac, replace, weights, ...])</td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td>Panel.select(crit[, axis])</td>
<td>Return data corresponding to axis labels matching criteria.</td>
</tr>
<tr>
<td>Panel.take(indices[, axis, convert, is_copy])</td>
<td>Analogous to ndarray.take.</td>
</tr>
<tr>
<td>Panel.truncate([before, after, axis, copy])</td>
<td>Truncates a sorted NDFrame before and/or after some particular dates.</td>
</tr>
</tbody>
</table>

pandas.Panel.add_prefix

Panel.add_prefix(prefix)

Concatenate prefix string with panel items names.

Parameters prefix : string
pandas: powerful Python data analysis toolkit, Release 0.16.2

**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.drop**

`Panel.drop(labels, axis=0, level=None, inplace=False, errors='raise')`
Return new object with labels in requested axis removed

**Parameters**
- `labels`: single label or list-like
- `axis`: int or axis name
- `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.
New in version 0.16.1.

**Returns**
- `dropped`: type of caller

**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.filter**

`Panel.filter(items=None, like=None, regex=None, axis=None)`
Restrict the info axis to set of items or wildcard

**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 None
The axis to filter on. By default this is the info axis. The “info axis” is the axis that is used when indexing with []. For example, `df = DataFrame({'a': [1, 2, 3, 4]})`; `df['a']`. So, the DataFrame columns are the info axis.

Notes

Arguments are mutually exclusive, but this is not checked for

**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.last('10D')` -> First 10 days

**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.reindex**

`Panel.reindex(items=None, major_axis=None, minor_axis=None, **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 (can be specified in order, or as 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:**

- 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 size gap to forward or backward fill

**Returns** reindexed : Panel

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=[‘A’, ‘B’, ‘C’])
```

**pandas.Panel.reindex_axis**

Panel.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=np.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 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 size gap to forward or backward fill

**Returns** reindexed : Panel
See also:

reindex, reindex_like

Examples

>>> df.reindex_axis(['A', 'B', 'C'], axis=1)

pandas.Panel.reindex_like

Panel.reindex_like (other, method=None, copy=True, limit=None)

return an object with matching indicies to myself

Parameters  
other : Object  
method : string or None  
copy : boolean, default True  
limit : int, default None  

Maximum size gap to forward or backward fill

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.

Parameters  
items, major_axis, minor_axis : dict-like or function, optional  
Transformation 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.

Returns  renamed : Panel (new object)

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.

New in version 0.16.1.

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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

pandas.Panel.select

Panel.select (crit, axis=0)
    Return data corresponding to axis labels matching criteria

Parameters crit : function
    To be called on each index (label). Should return True or False

axis : int

Returns selection : type of caller

pandas.Panel.take

Panel.take (indices, axis=0, convert=True, is_copy=True)
    Analogous to ndarray.take

Parameters indices : list / array of ints

axis : int, default 0

convert : translate neg to pos indices (default)

is_copy : mark the returned frame as a copy

Returns taken : type of caller
**pandas.Panel.truncate**

Panel.truncate(before=None, after=None, axis=None, copy=True)

Truncates a sorted NDFrame before and/or after some particular dates.

**Parameters**

- **before**: date
  - Truncate before date
- **after**: date
  - Truncate after date
- **axis**: the truncation axis, defaults to the stat axis
- **copy**: boolean, default is True,
  - return a copy of the truncated section

**Returns**

truncated : type of caller

### 33.5.10 Missing data handling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.dropna</td>
<td>Drop 2D from panel, holding passed axis constant</td>
</tr>
<tr>
<td>Panel.fillna</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
</tbody>
</table>

**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.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.
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

33.5.11 Reshaping, sorting, transposing

Panel.sort_index((axis, ascending)) Sort object by labels (along an axis)
Panel.swaplevel(i, j[, axis]) Swap levels i and j in a MultiIndex on a particular axis
Panel.transpose(*args, **kwargs) Permute the dimensions of the Panel
Panel.swapaxes(axis1, axis2[, copy]) Interchange axes and swap values axes appropriately
Panel.conform(frame[, axis]) Conform input DataFrame to align with chosen axis pair.

pandas.Panel.sort_index

Panel.sort_index(axis=0, ascending=True) Sort object by labels (along an axis)

Parameters axis : {0, 1}
Sort index/rows versus columns
ascending : boolean, default True
Sort ascending vs. descending

Returns sorted_obj : type of caller

pandas.Panel.swaplevel

Panel.swaplevel(i, j, 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)

**pandas.Panel.transpose**

`Panel.transpose(*args, **kwargs)`

Permute the dimensions of the Panel

Parameters `args` : three positional arguments: each one of
   `{0, 1, 2, 'items', 'major_axis', 'minor_axis'}`

`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.swapaxes**

`Panel.swapaxes(axis1, axis2, copy=True)`

Interchange axes and swap values axes appropriately

Returns `y` : same as input

**pandas.Panel.conform**

`Panel.conform(frame, axis='items')`

Conform input DataFrame to align with chosen axis pair.

Parameters `frame` : DataFrame
   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

**33.5.12 Combining / joining / merging**

**Panel.join(other[, how, lsuffix, rsuffix])**
Join items with other Panel either on major and minor axes column

**Panel.update(other[, join, overwrite, ...])**
Modify Panel in place using non-NA values from passed Panel, or object coercible to
pandas: powerful Python data analysis toolkit, Release 0.16.2

### 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.update

**Panel.update**(other, join='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.

#### 33.5.13 Time series-related

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.asfreq(freq[, method, how, normalize])</td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset objects.</td>
</tr>
<tr>
<td>Panel.shift(*args, **kwargs)</td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
<tr>
<td>Panel.resample(rule[, how, axis, ...])</td>
<td>Convenience method for frequency conversion and resampling of regular time-series data.</td>
</tr>
<tr>
<td>Panel.tz_convert(tz[, axis, level, copy])</td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td>Panel.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive TimeSeries to target time zone</td>
</tr>
</tbody>
</table>
pandas.Panel.asfreq

`Panel.asfreq(freq, method=None, how=None, normalize=False)`

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**
- `freq` : DateOffset object, or string
- `method` : {'backfill', 'bfill', 'pad', 'ffill', 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 method
- `how` : {'start', 'end'}, default end
  - For PeriodIndex only, see PeriodIndex.asfreq
- `normalize` : bool, default False
  - Whether to reset output index to midnight

**Returns**
- `converted` : type of caller

pandas.Panel.shift

`Panel.shift(*args, **kwargs)`

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.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)`

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters**
- `rule` : string
  - the offset string or object representing target conversion
- `how` : string
  - method for down- or re-sampling, default to ‘mean’ for downsampling
- `axis` : int, optional, default 0
- `fill_method` : string, default None
  - fill_method for upsampling
- `closed` : {'right', 'left'}
  - Which side of bin interval is closed

33.5. Panel 1373
**label** : {'right’, ‘left’}

Which bin edge label to label bucket with

**convention** : {'start’, ‘end’, ‘s’, ‘e’}

**kind** : “period”/”timestamp”

**loffset** : timedelta

Adjust the resampled time labels

**limit** : int, default None

Maximum size gap to when reindexing with fill_method

**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

**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 is 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 *(*args, **kwargs)*

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 is 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)

Attempt to infer fall dst-transition hours based on order

Raises **TypeError**

If the TimeSeries is tz-aware and tz is not None.

### 33.5.14 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.from_dict(data[, intersect, orient, dtype])</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>Panel.to_pickle(path)</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>Panel.to_excel(path[, na_rep, engine])</td>
<td>Write each DataFrame in Panel to a separate excel sheet</td>
</tr>
<tr>
<td>Panel.to_hdf(path_or_buf, key, **kwargs)</td>
<td>Activate the HDFStore</td>
</tr>
<tr>
<td>Panel.to_json([path_or_buf, orient, ...])</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>Panel.to_sparse([fill_value, kind])</td>
<td>Convert to SparsePanel</td>
</tr>
<tr>
<td>Panel.to_frame([filter_observations])</td>
<td>Transform wide format into long (stacked) format as DataFrame whose columns</td>
</tr>
<tr>
<td>Panel.to_clipboard([excel, sep])</td>
<td>Attempt to write text representation of object to the system clipboard This can</td>
</tr>
</tbody>
</table>

**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.to_pickle**

Panel.to_pickle(path)

Pickle (serialize) object to input file path

**Parameters**

- **path** : string
  File path
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_hdf

Panel.to_hdf(path_or_buf, key, **kwargs)
activate the HDFStore

Parameters
- **path_or_buf**: the path (string) or buffer to put the store
- **key**: string
  - identifier for the group in the store
- **mode**: optional, {'a', 'w', 'r', 'r+'}, default ‘a’
  - ’r’ Read-only; no data can be modified.
`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

**complevel**: int, 1-9, default 0

If a complib is specified compression will be applied where possible

**complib**: `{'zlib', 'bzlib2', 'lzo', 'blosc', None}`, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

**fletcher32**: bool, default False

If applying compression use the fletcher32 checksum

### pandas.Panel.to_json

**Panel.to_json**(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

date_format : {'epoch', 'iso'}

Type of date conversion. `epoch` = epoch milliseconds, iso’ = ISO8601, 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 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.

Returns same type as input object with filtered info axis

pandas.Panel.to_sparse

Panel.to_sparse(fill_value=None, kind='block')

Convert to SparsePanel

Parameters fill_value : float, default NaN

kind : {'block', 'integer'}

Returns y : SparseDataFrame

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_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.

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

33.6 Panel4D

33.6.1 Constructor

```python
Panel4D([data, labels, items, major_axis, ...]) Represents a 4 dimensional structured
```

pandas.Panel4D

class pandas.Panel4D (data=None, labels=None, items=None, major_axis=None, minor_axis=None, copy=False, dtype=None)

Represents a 4 dimensional structured

Parameters

data: ndarray (labels x items x major x minor), or dict of Panels

labels: Index or array-like

items: Index or array-like

major_axis: Index or array-like: axis=2

minor_axis: Index or array-like: axis=3

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>Fast label-based scalar accessor</td>
</tr>
<tr>
<td>axes</td>
<td>index(es) of the NDFrame</td>
</tr>
<tr>
<td>blocks</td>
<td>Internal property, property synonym for as_blocks()</td>
</tr>
<tr>
<td>dtypes</td>
<td>Return the dtypes in this object</td>
</tr>
<tr>
<td>empty</td>
<td>True if NDFrame is entirely empty [no items]</td>
</tr>
<tr>
<td>ftypes</td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<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>
</tbody>
</table>

Continued on next page
Table 33.82 – continued from previous page

<table>
<thead>
<tr>
<th>ix</th>
<th>A primarily label-location based indexer, with integer position fallback.</th>
</tr>
</thead>
<tbody>
<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>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.Panel4D.at**

Panel4D.at
Fast label-based scalar accessor

Similarly to loc, at provides label based scalar lookups. You can also set using these indexers.

**pandas.Panel4D.axes**

Panel4D.axes

index(es) of the NDFrame

**pandas.Panel4D.blocks**

Panel4D.blocks

Internal property, property synonym for as_blocks()

**pandas.Panel4D.dtypes**

Panel4D.dtypes

Return the dtypes in this object

**pandas.Panel4D.empty**

Panel4D.empty

True if NDFrame is entirely empty [no items]

**pandas.Panel4D.ftypes**

Panel4D.ftypes

Return the ftypes (indication of sparse/dense and dtype) in this object.

**pandas.Panel4D.iat**

Panel4D.iat

Fast integer location scalar accessor.

Similarly to iloc, iat provides integer based lookups. You can also set using these indexers.
pandas.Panel4D.iloc

Panel4D.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.

.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.Panel4D.ix

Panel4D.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 hierachical 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.Panel4D.loc

Panel4D.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.

.loc will raise a KeyError when the items are not found.

See more at Selection by Label
**pandas.Panel4D.ndim**

Panel4D.ndim  
Number of axes / array dimensions

**pandas.Panel4D.shape**

Panel4D.shape  
tuple of axis dimensions

**pandas.Panel4D.size**

Panel4D.size  
number of elements in the NDFrame

**pandas.Panel4D.values**

Panel4D.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 upcase to int32.

**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Return an object with absolute value taken.</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>align(other[, join, axis, level, copy, ...])</td>
<td>Align two object on their axes with the</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 input axis of the Panel</td>
</tr>
<tr>
<td>as_blocks()</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 all TimeSeries inside to specified frequency using DateOffset objects.</td>
</tr>
<tr>
<td>astype(dtype[, copy, raise_on_error])</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>at_time(time[, asof])</td>
<td>Select values at particular time of day (e.g. 9:00-9:30 AM)</td>
</tr>
<tr>
<td>between_time(start_time, end_time[, ...])</td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM)</td>
</tr>
<tr>
<td>bfill([axis, inplace, limit, downcast()])</td>
<td>Synonym for NDFrame.fillna(method='bfill')</td>
</tr>
</tbody>
</table>
bool()
clip(lower, upper, out, axis))
clip_lower(threshold[, axis])
clip_upper(threshold[, axis])
compound([axis, skipna, level])
conform(frame[, axis])
consolidate(inplace)
convert_objects([convert_dates, ...])
copy([deep])
count([axis])
cummmax([axis, dtype, out, skipna])
cummin([axis, dtype, out, skipna])
cumprod([axis, dtype, out, skipna])
cumsum([axis, dtype, out, skipna])
describe([percentile_width, percentiles, ...])
div(other[, axis])
divide(other[, axis])
drop(labels[, axis, level, inplace, errors])
dropna(*args, **kwargs)
eq(other)
equals(other)
ffill(axis, inplace, limit, downcast)
fillna([value, method, axis, inplace, ...])
filter(*args, **kwargs)
first(offset)
floordiv(other[, axis])
fromDict(data[, intersect, orient, dtype])
from_dict(data[, intersect, orient, dtype])
ge(other)
get(key[, default])
get_dtype_counts()
get_ftype_counts()
get_value(*args, **kwargs)
get_values()
groupby(*args, **kwargs)
gt(other)
head(n)
interpolate([method, axis, limit, inplace, ...])
isnull()
iteritems()
iterkv(*args, **kwargs)
join(*args, **kwargs)
keys()
kurt([axis, skipna, level, numeric_only])
kurtosis([axis, skipna, level, numeric_only])
last(offset)
le(other)
load(path)
lt(other)
mad([axis, skipna, level])
major_xa(key[, copy])
mask(cond[, other, inplace, axis, level, ...])

Table 33.83 – continued from previous page

Return the bool of a single element PandasObject
Trim values at input threshold(s)
Return copy of the input with values below given value(s) truncated
Return copy of input with values above given value(s) truncated
Return the compound percentage of the values for the requested axis
Conform input DataFrame to align with chosen axis pair.
Compute NDFrame with “consolidated” internals (data of each dtype grouped together)
Attempt to infer better dtype for object columns
Make a copy of this object
Return number of observations over requested axis.
Return cumulative max over requested axis.
Return cumulative min over requested axis.
Return return cumulative prod over requested axis.
Return cumulative sum over requested axis.
Generate various summary statistics, excluding NaN values.
Floating division of series and other, element-wise (binary operator truediv).
Floating division of series and other, element-wise (binary operator truediv).
Return new object with labels in requested axis removed

Wrapper for comparison method eq
Determines if two NDFrame objects contain the same elements.
Synonym for NDFrame.fillna(method='ffill')
Fill NA/NaN values using the specified method

Convenience method for subsetting initial periods of time series data
Integer division of series and other, element-wise (binary operator floordiv).
Construct Panel from dict of DataFrame objects
Construct Panel from dict of DataFrame objects
Wrapper for comparison method ge
Get item from object for given key (DataFrame column, Panel slice, etc.).
Return the counts of dtypes in this object
Return the counts of ftypes in this object
Quickly retrieve single value at (item, major, minor) location
same as values (but handles sparseness conversions)

Wrapper for comparison method gt
Interpolate values according to different methods.
Return a boolean same-sized object indicating if the values are null
Iterate over (label, values) on info axis
iteritems alias used to get around 2to3. Deprecated
Get the ‘info axis’ (see Indexing for more)
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis
Convenience method for subsetting final periods of time series data
Wrapper for comparison method le
Deprecated.
Wrapper for comparison method lt
Return the mean absolute deviation of the values for the requested axis
Return slice of panel along major axis
Return an object of same shape as self and whose corresponding entries are from
pandas: powerful Python data analysis toolkit, Release 0.16.2

Table 33.83 – continued from previous page

<table>
<thead>
<tr>
<th>Method Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max()</code></td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td><code>mean()</code></td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td><code>median()</code></td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><code>mod()</code></td>
<td>Return slice of panel along minor axis</td>
</tr>
<tr>
<td><code>mul()</code></td>
<td>Modulo of series and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>multiply()</code></td>
<td>Multiplication of series and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>ndarray.take()</code></td>
<td>Wraper for comparison method <code>ne</code>.</td>
</tr>
<tr>
<td><code>pct_change()</code></td>
<td>Return a boolean same-sized object indicating if the values are</td>
</tr>
<tr>
<td><code>pipe()</code></td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>Apply <code>func(self, *args, **kwargs)</code></td>
</tr>
<tr>
<td><code>pow()</code></td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td><code>prod()</code></td>
<td>Exponential power of series and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>product()</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>radd()</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>rfloordiv()</code></td>
<td>Addition of series and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>rmod()</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>rpow()</code></td>
<td>Conform Panel to new index with optional filling logic, placing NA/NaN in locations</td>
</tr>
<tr>
<td><code>rsub()</code></td>
<td>Conform input object to new index with optional filling logic, placing NA/NaN in locations</td>
</tr>
<tr>
<td><code>rtruediv()</code></td>
<td>return an object with matching indicies to myself</td>
</tr>
<tr>
<td><code>sample()</code></td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><code>save()</code></td>
<td>Alter index and / or columns using input function or functions.</td>
</tr>
<tr>
<td><code>select()</code></td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td><code>sem()</code></td>
<td>Convenience method for frequency conversion and resampling of regular time-series data.</td>
</tr>
<tr>
<td><code>set_axis()</code></td>
<td>Integer division of series and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>set_value()</code></td>
<td>Modulo of series and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>shift()</code></td>
<td>Multiplication of series and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>skew()</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>slice_shift()</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>sort_index()</code></td>
<td>Returns a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><code>squeeze()</code></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><code>std()</code></td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td><code>sub()</code></td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><code>subtract()</code></td>
<td>public version of axis assignment</td>
</tr>
<tr>
<td><code>sum()</code></td>
<td>Quickly set single value at (item, major, minor) location</td>
</tr>
<tr>
<td><code>swapaxes()</code></td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td><code>swaplevel()</code></td>
<td>Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>tail()</code></td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td><code>take()</code></td>
<td>squeeze length 1 dimensions</td>
</tr>
<tr>
<td><code>to_long()</code></td>
<td>Return unbiased standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>to_clipboard()</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>to_clipboard()</code></td>
<td>Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>to_long()</code></td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td><code>to_clipboard()</code></td>
<td>Interchange axes and swap values axes appropriately</td>
</tr>
<tr>
<td><code>to_clipboard()</code></td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
</tbody>
</table>

Continued on next page
Table 33.83 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_dense()</code></td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td><code>to_excel(*args, **kwargs)</code></td>
<td>Activate the HDFStore</td>
</tr>
<tr>
<td><code>to_frame(*args, **kwargs)</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>to_hdf(path_or_buf, key, **kwargs)</code></td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_json([path_or_buf, orient, date_format, ...])</code></td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td><code>to_long(*args, **kwargs)</code></td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td><code>to_msgpack([path_or_buf])</code></td>
<td>Permute the dimensions of the Panel</td>
</tr>
<tr>
<td><code>to_pickle(path)</code></td>
<td>Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>to_sparse(*args, **kwargs)</code></td>
<td>Truncates a sorted NDFrame before and/or after some particular dates.</td>
</tr>
<tr>
<td><code>transmute((before, after, axis, copy))</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(*args, **kwargs)</code></td>
<td>Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.</td>
</tr>
<tr>
<td><code>update(other[, join, overwrite, ...])</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>var([axis, skipna, level, ddof, numeric_only])</code></td>
<td>Return an object of same shape as self and whose corresponding entries are from</td>
</tr>
<tr>
<td><code>where(cond[, other, inplace, axis, level, ...])</code></td>
<td>Return slice of panel along selected axis</td>
</tr>
</tbody>
</table>

**pandas.Panel4D.abs**

`Panel4D.abs()`

Return an object with absolute value taken. Only applicable to objects that are all numeric.

Returns abs: type of caller

**pandas.Panel4D.add**

`Panel4D.add(other, axis=0)`

Addition of series and other, element-wise (binary operator `add`). Equivalent to `panel + other`.

Parameters other : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D

See also:

`Panel4D.radd`

**pandas.Panel4D.add_prefix**

`Panel4D.add_prefix(prefix)`

Concatenate prefix string with panel items names.

Parameters prefix : string

Returns with_prefix: type of caller
**pandas.Panel4D.add_suffix**

Panel4D.add_suffix(suffix)
Concenate suffix string with panel items names

**Parameters**
- suffix : string

**Returns**
- with_suffix : type of caller

**pandas.Panel4D.align**

Panel4D.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)
Align two object 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, 1}, default 0
  
  Filling axis, method and limit

**Returns**
- (left, right) : (type of input, type of other)
  
  Aligned objects

**pandas.Panel4D.all**

Panel4D.all(axis=None, bool_only=None, skipna=None, level=None, **kwargs)
Return whether all elements are True over requested axis

**Parameters**
- axis : {labels (0), items (1), major_axis (2), minor_axis (3)}
- 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 Panel

**bool_only** : boolean, default None

Include only boolean data. If None, will attempt to use everything, then use only boolean data

**Returns** all : Panel or Panel4D (if level specified)

**pandas.Panel4D.any**

Panel4D.**any** *(axis=None, bool_only=None, skipna=None, level=None, **kwargs)*

Return whether any element is True over requested axis

**Parameters** axis : [labels (0), items (1), major_axis (2), minor_axis (3)]

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 Panel

**bool_only** : boolean, default None

Include only boolean data. If None, will attempt to use everything, then use only boolean data

**Returns** any : Panel or Panel4D (if level specified)

**pandas.Panel4D.apply**

Panel4D.**apply** *(func, axis='major', **kwargs)*

Applies function along input axis of the Panel

**Parameters** func : function

Function to apply to each combination of ‘other’ axes e.g. if axis = ‘items’, then the combination of major_axis/minor_axis will be passed a Series

axis : [‘major’, ‘minor’, ‘items’]

**Additional keyword arguments will be passed as keywords to the function**

**Returns** result : Pandas Object

**Examples**

```python
>>> p.apply(numpy.sqrt)  # returns a Panel
>>> p.apply(lambda x: x.sum(), axis=0)  # equiv to p.sum(0)
>>> p.apply(lambda x: x.sum(), axis=1)  # equiv to p.sum(1)
>>> p.apply(lambda x: x.sum(), axis=2)  # equiv to p.sum(2)
```
pandas.Panel4D.as_blocks

Panel4D.as_blocks()
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

**NOTE:** the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Returns values: a dict of dtype -> Constructor Types

pandas.Panel4D.as_matrix

Panel4D.as_matrix()

pandas.Panel4D.asfreq

Panel4D.asfreq(freq=None, method=None, how=None, normalize=False)
Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters freq: DateOffset object, or string
   method: {'backfill', 'bfill', 'pad', 'ffill', 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 method
   how: {'start', 'end'}, default end
      For PeriodIndex only, see PeriodIndex.asfreq
   normalize: bool, default False
      Whether to reset output index to midnight

Returns converted: type of caller

pandas.Panel4D.astype

Panel4D.astype(dtype, copy=True, raise_on_error=True, **kwargs)
Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters dtype: numpy.dtype or Python type
   raise_on_error: raise on invalid input
   kwargs: keyword arguments to pass on to the constructor

Returns casted: type of caller

pandas.Panel4D.at_time

Panel4D.at_time(time, asof=False)
Select values at particular time of day (e.g. 9:30AM)

Parameters time: datetime.time or string
pandas.Panel4D.between_time

Panel4D.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.Panel4D.bfill

Panel4D.bfill(axis=None, inplace=False, limit=None, downcast=None)
Synonym for NDFrame.fillna(method='bfill')

pandas.Panel4D.bool

Panel4D.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.Panel4D.clip

Panel4D.clip(lower=None, upper=None, out=None, axis=None)
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.

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 1
```
>>> t
0 -0.3
1 -0.2
2 -0.1
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.Panel4D.clip_lower**

Panel4D.clip_lower(threshold, axis=None)
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.

Returns
clipped : same type as input

See also:
clip

**pandas.Panel4D.clip_upper**

Panel4D.clip_upper(threshold, axis=None)
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.

Returns
clipped : same type as input

See also:
clip

**pandas.Panel4D.compound**

Panel4D.compound(axis=None, skipna=None, level=None)
Return the compound percentage of the values for the requested axis
**Pandas** is a powerful Python data analysis toolkit, Release 0.16.2

**Parameters**

- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
  - **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 Panel
  - **numeric_only**: boolean, default None
    - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **compounded**: Panel or Panel4D (if level specified)

**pandas.Panel4D.conform**

**Panel4D.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.Panel4D.consolidate**

**Panel4D.consolidate**(inplace=False)
Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray). Mainly an internal API function, but available here to the savvy user

- **Parameters**
  - **inplace**: boolean, default False
    - If False return new object, otherwise modify existing object

**Returns**

consolidated: type of caller

**pandas.Panel4D.convert_objects**

**Panel4D.convert_objects**(convert_dates=True, convert_numeric=False, convert_timedeltas=True, copy=True)
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

**pandas.Panel4D.copy**

Panel4D.**copy**(deep=True)

Make a copy of this object

**Parameters** deep : boolean or string, default True

Make a deep copy, i.e. also copy data

**Returns** copy : type of caller

**pandas.Panel4D.count**

Panel4D.**count**(axis='major')

Return number of observations over requested axis.

**Parameters** axis : {'items', 'major', 'minor'} or {0, 1, 2}

**Returns** count : DataFrame

**pandas.Panel4D.cummax**

Panel4D.**cummax**(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative max over requested axis.

**Parameters** axis : {labels (0), items (1), major_axis (2), minor_axis (3)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**Returns** max : Panel

**pandas.Panel4D.cummin**

Panel4D.**cummin**(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative min over requested axis.

**Parameters** axis : {labels (0), items (1), major_axis (2), minor_axis (3)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**Returns** min : Panel
pandas.Panel4D.cumprod

**Panel4D.cumprod** *(axis=None, dtype=None, out=None, skipna=True, **kwargs)*

Return cumulative prod over requested axis.

**Parameters**

- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **skipna**: boolean, default True
  
  Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

- **prod**: Panel

pandas.Panel4D.cumsum

**Panel4D.cumsum** *(axis=None, dtype=None, out=None, skipna=True, **kwargs)*

Return cumulative sum over requested axis.

**Parameters**

- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **skipna**: boolean, default True
  
  Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

- **sum**: Panel

pandas.Panel4D.describe

**Panel4D.describe** *(percentile_width=None, percentiles=None, include=None, exclude=None)*

Generate various summary statistics, excluding NaN values.

**Parameters**

- **percentile_width**: float, deprecated
  
  The percentile_width argument will be removed in a future version. Use percentiles instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75.

- **percentiles**: array-like, optional
  
  The percentiles to include in the output. Should all be in the interval [0, 1]. By default percentiles is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

- **include, exclude**: list-like, ‘all’, or None (default)
  
  Specify the form of the returned result. Either:
  - None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
  - A list of dtypes or strings to be included/excluded. To select all numeric types use numpy number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
  - If include is the string ‘all’, the output column-set will match the input one.

**Returns**

- **summary**: NDFrame of summary statistics

**See also:**

- DataFrame.select_dtypes
Notes

The output DataFrame index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the count and most common pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.

pandas.Panel4D.div

Panel4D.div(other, axis=0)
Floating division of series and other, element-wise (binary operator truediv). Equivalent to panel / other.

Parameters other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}
Axis to broadcast over

Returns Panel4D
See also:
Panel4D.rtruediv

pandas.Panel4D.divide

Panel4D.divide(other, axis=0)
Floating division of series and other, element-wise (binary operator truediv). Equivalent to panel / other.

Parameters other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}
Axis to broadcast over

Returns Panel4D
See also:
Panel4D.rtruediv

pandas.Panel4D.drop

Panel4D.drop(labels, axis=0, level=None, inplace=False, errors='raise')
Return new object with labels in requested axis removed
Parameters labels : single label or list-like
    axis : int or axis name
    level : int or level name, default None
    inplace : bool, default False
    errors : {'ignore', 'raise'}, default 'raise'
        If ‘ignore’, suppress error and existing labels are dropped.
        New in version 0.16.1.
Returns dropped : type of caller

`pandas.Panel4D.dropna`

Panel4D.dropna(*args, **kwargs)

`pandas.Panel4D.eq`

Panel4D.eq(other)
    Wrapper for comparison method eq

`pandas.Panel4D.equals`

Panel4D.equals(other)
    Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

`pandas.Panel4D.fillna`

Panel4D.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, 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.

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

pandas.Panel4D.filter

Panel4D.filter(*args, **kwargs)

pandas.Panel4D.first

Panel4D.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.last(‘10D’) -> First 10 days

pandas.Panel4D.floordiv

Panel4D.floordiv(other, axis=0)

Integer division of series and other, element-wise (binary operator floordiv). Equivalent to panel // other.

Parameters other : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over
panel4D

pandas.Panel4D.fromDict

classmethod Panel4D.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.Panel4D.from_dict

classmethod Panel4D.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.Panel4D

Panel4D.ge(other)
Wrapper for comparison method ge

pandas.Panel4D.get

Panel4D.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.Panel4D.get_dtype_counts

Panel4D.get_dtype_counts()
Return the counts of dtypes in this object

pandas.Panel4D.get_ftype_counts

Panel4D.get_ftype_counts()
Return the counts of ftypes in this object

pandas.Panel4D.get_value

Panel4D.get_value(*args, **kwargs)
Quickly retrieve single value at (item, major, minor) location

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.Panel4D.get_values

Panel4D.get_values()
same as values (but handles sparseness conversions)

pandas.Panel4D.groupby

Panel4D.groupby(*args, **kwargs)
pandas.Panel4D.gt

Panel4D.gt(other)
Wrapper for comparison method gt

pandas.Panel4D.head

Panel4D.head(n=5)

pandas.Panel4D.interpolate

Panel4D.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast=None, **kwargs)
Interpolate values according to different methods.

**Parameters**

- **method**: {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'pchip'}

  - 'linear': ignore the index and treat the values as equally spaced. 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 with the order given both 'polynomial' and 'spline' require that you also specify and order (int) e.g. df.interpolate(method='polynomial', order=4)
  - 'krogh', 'piecewise_polynomial', 'spline', and 'pchip' are all wrappers around the scipy interpolation methods of similar names. See the scipy documentation for more on their behavior: http://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-interpolation http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html

- **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.

- **inplace**: bool, default False
  Update the NDFrame in place if possible.

- **downcast**: optional, ‘infer’ or None, defaults to None
  Downcast dtypes if possible.

**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.Panel4D.isnull**

Panel4D.
```
isnull()
```
Return a boolean same-sized object indicating if the values are null

See also:

**notnull** boolean inverse of isnull

**pandas.Panel4D.iteritems**

Panel4D.
```iteritems()
```
Iterate over (label, values) on info axis

This is index for Series, columns for DataFrame, major_axis for Panel, and so on.

**pandas.Panel4D.iterkv**

Panel4D.
```iterkv(*args, **kwargs)
```
itertools alias used to get around 2to3. Deprecated

**pandas.Panel4D.join**

Panel4D.
```
join(*args, **kwargs)
```

**pandas.Panel4D.keys**

Panel4D.
```
keys()
```
Get the ‘info axis’ (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

**pandas.Panel4D.kurt**

Panel4D.
```
kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
```
Return unbiased kurtosis over requested axis using Fishers definition of kurtosis (kurtosis of normal ==
0.0). Normalized by N-1
Parameters `axis`: {labels (0), items (1), major_axis (2), minor_axis (3)}

`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 Panel

`numeric_only`: boolean, default None
- Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns `kurt`: Panel or Panel4D (if level specified)

**pandas.Panel4D.kurtosis**

`Panel4D.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`
- Return unbiased kurtosis over requested axis using Fishers definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters `axis`: {labels (0), items (1), major_axis (2), minor_axis (3)}

`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 Panel

`numeric_only`: boolean, default None
- Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns `kurt`: Panel or Panel4D (if level specified)

**pandas.Panel4D.last**

`Panel4D.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.Panel4D.le**

`Panel4D.le(other)`
- Wrapper for comparison method le
**pandas.Panel4D.load**

\[\text{Panel4D.load(path)}\]

Deprecated. Use read_pickle instead.

**pandas.Panel4D.lt**

\[\text{Panel4D.lt(other)}\]

Wrapper for comparison method lt

**pandas.Panel4D.mad**

\[\text{Panel4D.mad(axis=None, skipna=None, level=None)}\]

Return the mean absolute deviation of the values for the requested axis

**Parameters**
- `axis`: {labels (0), items (1), major_axis (2), minor_axis (3)}
- `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 Panel
- `numeric_only`: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**
- `mad`: Panel or Panel4D (if level specified)

**pandas.Panel4D.major_xs**

\[\text{Panel4D.major_xs(key, copy=None)}\]

Return slice of panel along major axis

**Parameters**
- `key`: object
  - Major axis label
- `copy`: boolean [deprecated]
  - Whether to make a copy of the data

**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 it is a superset of major_xs functionality, see MultiIndex Slicers
pandas.Panel4D.mask

Panel4D.mask (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)
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 or array
- **other**: scalar or NDFrame
- **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
- **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)

Returns wh: same type as caller

pandas.Panel4D.max

Panel4D.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**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **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 Panel
- **numeric_only**: boolean, default None
  Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns max: Panel or Panel4D (if level specified)

pandas.Panel4D.mean

Panel4D.mean (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the mean of the values for the requested axis

Parameters
- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **skipna**: boolean, default True
  Exclude NA/null values. If an entire row/column is NA, the result will be NA
pandas: powerful Python data analysis toolkit, Release 0.16.2

```
level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing
into a Panel

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then
use only numeric data

Returns mean : Panel or Panel4D (if level specified)

pandas.Panel4D.median

Panel4D.median(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return the median of the values for the requested axis

Parameters axis : {labels (0), items (1), major_axis (2), minor_axis (3)}

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 Panel

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then
use only numeric data

Returns median : Panel or Panel4D (if level specified)

pandas.Panel4D.min

Panel4D.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 : {labels (0), items (1), major_axis (2), minor_axis (3)}

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 Panel

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then
use only numeric data

Returns min : Panel or Panel4D (if level specified)
```
pandas.Panel4D.minor_xs

Panel4D.minor_xs(key, copy=None)
Return slice of panel along minor axis

Parameters key : object
    Minor axis label

copy : boolean [deprecated]
    Whether to make a copy of the data

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 it is a superset of minor_xs functionality, see MultiIndex Slicers

pandas.Panel4D.mod

Panel4D.mod(other, axis=0)
Modulo of series and other, element-wise (binary operator mod). Equivalent to panel % other.

Parameters other : Panel or Panel4D
    Axis to broadcast over

Returns Panel4D

See also:
Panel4D.rmod

pandas.Panel4D.mul

Panel4D.mul(other, axis=0)
Multiplication of series and other, element-wise (binary operator mul). Equivalent to panel * other.

Parameters other : Panel or Panel4D
    Axis to broadcast over

Returns Panel4D

See also:
Panel4D.rmul
pandas.Panel4D.multiply

Panel4D.multiply(\text{other, \text{axis}=0})

Multiplication of series and other, element-wise (binary operator \textit{mul}). Equivalent to panel \ast\ other.

Parameters other : Panel or Panel4D
axis : \{labels, items, major_axis, minor_axis\}

Axis to broadcast over

Returns Panel4D

See also:
Panel4D.rmul

pandas.Panel4D.ne

Panel4D.ne(\text{other})

Wrapper for comparison method ne

pandas.Panel4D.notnull

Panel4D.notnull()

Return a boolean same-sized object indicating if the values are not null

See also:

\textit{isnull} boolean inverse of notnull

pandas.Panel4D.pct_change

Panel4D.pct_change(\text{periods}=1, \text{fill_method}='pad', \text{limit}=None, \text{freq}=None, **\text{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 \textit{axis} keyword argument.
pandas.Panel4D.pipe

Panel4D.pipe(func, *args, **kwargs)

Apply func(self, *args, **kwargs)

New in version 0.16.2.

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 : positional arguments passed into func.
- kwargs : 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 on Series or DataFrames. 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.Panel4D.pop

Panel4D.pop(item)

Return item and drop from frame. Raise KeyError if not found.

pandas.Panel4D.pow

Panel4D.pow(other, axis=0)

Exponential power of series and other, element-wise (binary operator pow). Equivalent to panel ** other.
Pandas: powerful Python data analysis toolkit, Release 0.16.2

**Parameters**
- `other`: Panel or Panel4D
  - `axis`: {labels, items, major_axis, minor_axis}
    - Axis to broadcast over

**Returns**
- Panel4D

**See also:**
- Panel4D.rpow
- Panel4D.prod

```python
def Panel4D.prod(axis=None, skipna=None, level=None, numeric_only=None, **kwargs):
    Return the product of the values for the requested axis

Parameters
- `axis`: {labels (0), items (1), major_axis (2), minor_axis (3)}
- `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 Panel
- `numeric_only`: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns
- `prod`: Panel or Panel4D (if level specified)
```

```python
def Panel4D.product(axis=None, skipna=None, level=None, numeric_only=None, **kwargs):
    Return the product of the values for the requested axis

Parameters
- `axis`: {labels (0), items (1), major_axis (2), minor_axis (3)}
- `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 Panel
- `numeric_only`: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns
- `prod`: Panel or Panel4D (if level specified)
```
pandas.Panel4D.radd

Panel4D.radd(other, axis=0)
Addition of series and other, element-wise (binary operator radd). Equivalent to other + panel.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns Panel4D

See also:
Panel4D.add

pandas.Panel4D.rdiv

Panel4D.rdiv(other, axis=0)
Floating division of series and other, element-wise (binary operator rtruediv). Equivalent to other / panel.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns Panel4D

See also:
Panel4D.truediv

pandas.Panel4D.reindex

Panel4D.reindex(items=None, major_axis=None, minor_axis=None, **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 (can be specified in order, or as keywords) New labels / index to conform to. Preferably an Index object to avoid duplicating data
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

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 size gap to forward or backward fill

Returns reindexed : Panel

Examples

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

pandas.Panel4D.reindex_axis

Panel4D.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 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 size gap to forward or backward fill

Returns reindexed : Panel

See also:

reindex, reindex_like
Examples

```python
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

**pandas.Panel4D.reindex_like**

`Panel4D.reindex_like(other, method=None, copy=True, limit=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 size gap to forward or backward fill

- **Returns**
  - `reindexed`: same as input

**Notes**

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

**pandas.Panel4D.rename**

`Panel4D.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.

- **Parameters**
  - `items`, `major_axis`, `minor_axis`: dict-like or function, optional
    - Transformation 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.

- **Returns**
  - `renamed`: Panel (new object)

**pandas.Panel4D.rename_axis**

`Panel4D.rename_axis(mapper, axis=0, copy=True, inplace=False)`

Alter index and / or columns using 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.

- **Parameters**
  - `mapper`: dict-like or function, optional
    - `axis`: int or string, default 0
  - `copy`: boolean, default True
    - Also copy underlying data
pandas: powerful Python data analysis toolkit, Release 0.16.2

 inplace : boolean, default False

 Returns renamed : type of caller

 pandas.Panel4D.replace

Panel4D.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 form 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.Panel4D.resample**

`Panel4D.resample` *(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0)*

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters**

- **rule**: string
  
  the offset string or object representing target conversion

- **how**: string
  
  method for down- or re-sampling, default to ‘mean’ for downsampling

- **axis**: int, optional, default 0

- **fill_method**: string, default None
  
  fill_method for upsampling
**closed** : {'right', 'left'}
Which side of bin interval is closed

**label** : {'right', 'left'}
Which bin edge label to label bucket with

**convention** : {'start', 'end', 's', 'e'}

**kind** : “period”/”timestamp”

**loffset** : timedelta
Adjust the resampled time labels

**limit** : int, default None
Maximum size gap to when reindexing with fill_method

**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

### pandas.Panel4D.rfloordiv

**Panel4D.rfloordiv**(other, axis=0)
Integer division of series and other, element-wise (binary operator rfloordiv). Equivalent to other // panel.

**Parameters**

- **other** : Panel or Panel4D
- **axis** : {labels, items, major_axis, minor_axis}
  
  Axis to broadcast over

**Returns** Panel4D

**See also:**

Panel4D.floordiv

### pandas.Panel4D.rmod

**Panel4D.rmod**(other, axis=0)
Modulo of series and other, element-wise (binary operator rmod). Equivalent to other % panel.

**Parameters**

- **other** : Panel or Panel4D
- **axis** : {labels, items, major_axis, minor_axis}
  
  Axis to broadcast over

**Returns** Panel4D

**See also:**

Panel4D.mod
pandas:` Panel4D.rmul

Panel4D.rmul(other, axis=0)
Multiplication of series and other, element-wise (binary operator `rmul`). Equivalent to `other * panel`.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

See also:
Panel4D.mul

pandas:` Panel4D.rpow

Panel4D.rpow(other, axis=0)
Exponential power of series and other, element-wise (binary operator `rpow`). Equivalent to `other ** panel`.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

See also:
Panel4D.pow

pandas:` Panel4D.rsub

Panel4D.rsub(other, axis=0)
Subtraction of series and other, element-wise (binary operator `rsub`). Equivalent to `other - panel`.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

See also:
Panel4D.sub

pandas:` Panel4D.rtruediv

Panel4D.rtruediv(other, axis=0)
Floating division of series and other, element-wise (binary operator `rtruediv`). Equivalent to `other / panel`.

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}
[72x748]pandas: powerful Python data analysis toolkit, Release 0.16.2

Axis to broadcast over

Returns Panel4D

See also:
Panel4D.truediv

pandas.Panel4D.sample

Panel4D.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.

New in version 0.16.1.

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 called on a DataFrame, will accept the name of a column when axis = 0. 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 Same type as caller.

pandas.Panel4D.save

Panel4D.save(path)

Deprecated. Use to_pickle instead

pandas.Panel4D.select

Panel4D.select(crit, axis=0)

Return data corresponding to axis labels matching criteria

Parameters

crit : function
    To be called on each index (label). Should return True or False
axis : int

Returns selection : type of caller

pandas.Panel4D.sem

Panel4D.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 : {labels (0), items (1), major_axis (2), minor_axis (3)}
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 Panel
numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then
use only numeric data

Returns sem : Panel or Panel4D (if level specified)

pandas.Panel4D.set_axis

Panel4D.set_axis(axis, labels)
public version of axis assignment

pandas.Panel4D.set_value

Panel4D.set_value(*args, **kwargs)
Quickly set single value at (item, major, minor) location

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.Panel4D.shift

Panel4D.shift(*args, **kwargs)
**pandas.Panel4D.skew**

Panel4D.skew(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)

Return unbiased skew over requested axis Normalized by N-1

**Parameters**

- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **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 Panel
- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

- **skew**: Panel or Panel4D (if level specified)

**pandas.Panel4D.slice_shift**

Panel4D.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.Panel4D.sort_index**

Panel4D.sort_index(axis=0, ascending=True)

Sort object by labels (along an axis)

**Parameters**

- **axis**: {0, 1}
  - Sort index/rows versus columns
- **ascending**: boolean, default True
  - Sort ascending vs. descending

**Returns**

- **sorted_obj**: type of caller

**pandas.Panel4D.squeeze**

Panel4D.squeeze()

squeeze length 1 dimensions
pandas: powerful Python data analysis toolkit, Release 0.16.2

**pandas.Panel4D.std**

Panel4D.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

**Parameters**

- **axis**: {labels (0), items (1), major_axis (2), minor_axis (3)}
- **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 Panel
- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

std : Panel or Panel4D (if level specified)

**pandas.Panel4D.sub**

Panel4D.sub(other, axis=0)

Subtraction of series and other, element-wise (binary operator sub). Equivalent to panel - other.

**Parameters**

- **other**: Panel or Panel4D
- **axis**: {labels, items, major_axis, minor_axis}
  - Axis to broadcast over

**Returns**

Panel4D

See also:

Panel4D.rsub

**pandas.Panel4D.subtract**

Panel4D.subtract(other, axis=0)

Subtraction of series and other, element-wise (binary operator sub). Equivalent to panel - other.

**Parameters**

- **other**: Panel or Panel4D
- **axis**: {labels, items, major_axis, minor_axis}
  - Axis to broadcast over

**Returns**

Panel4D

See also:

Panel4D.rsub
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.Panel4D.sum

Panel4D.sum(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return the sum of the values for the requested axis

Parameters
axis : {labels (0), items (1), major_axis (2), minor_axis (3)}

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 Panel

numeric_only : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns
sum : Panel or Panel4D (if level specified)

pandas.Panel4D.swapaxes

Panel4D.swapaxes(axis1, axis2, copy=True)
Interchange axes and swap values axes appropriately

Returns
y : same as input

pandas.Panel4D.swaplevel

Panel4D.swaplevel(i, j, 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)

pandas.Panel4D.tail

Panel4D.tail(n=5)

pandas.Panel4D.take

Panel4D.take(indices, axis=0, convert=True, is_copy=True)
Analogous to ndarray.take

Parameters
indices : list / array of ints
axis : int, default 0
convert : translate neg to pos indices (default)
is_copy : mark the returned frame as a copy
**Returns taken**: type of caller

```python
pandas.Panel4D.toLong
```

```python
Panel4D.toLong(*args, **kwargs)
```

```python
pandas.Panel4D.to_clipboard
```

```python
Panel4D.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

```python
pandas.Panel4D.to_dense
```

```python
Panel4D.to_dense()
```

Return dense representation of NDFrame (as opposed to sparse)

```python
pandas.Panel4D.to_excel
```

```python
Panel4D.to_excel(*args, **kwargs)
```

```python
pandas.Panel4D.to_frame
```

```python
Panel4D.to_frame(*args, **kwargs)
```

```python
pandas.Panel4D.to_hdf
```

```python
Panel4D.to_hdf(path_or_buf, key, **kwargs)
```

activate the HDFStore

- **Parameters path_or_buf**: the path (string) or buffer to put the store
- **key**: string
identifier for the group in the store

**mode**: optional, {'a', 'w', 'r', 'r+'}, default 'a'

- 'r': Read-only; no data can be modified.
- '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'

- 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

**complevel**: int, 1-9, default 0

If a complib is specified compression will be applied where possible

**complib**: {'zlib', 'bz2', 'lzma', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

**fletcher32**: bool, default False

If applying compression use the fletcher32 checksum

---

**pandas.Panel4D.to_json**

Panel4D.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=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 a StringIO of 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

date_format : {‘epoch’, ‘iso’}

Type of date conversion. epoch = epoch milliseconds, iso’ = ISO8601, 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.

Returns same type as input object with filtered info axis

pandas.Panel4D.to_long

Panel4D.to_long(*args, **kwargs)

pandas.Panel4D.to_msgpack

Panel4D.to_msgpack(path_or_buf=None, **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.Panel4D.to_pickle**

`Panel4D.to_pickle(path)`

Pickle (serialize) object to input file path

**Parameters**

- **path**: string
  
  File path

**pandas.Panel4D.to_sparse**

`Panel4D.to_sparse(*args, **kwargs)`

**pandas.Panel4D.to_sql**

`Panel4D.to_sql(name, con, flavor='sqlite', 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', 'mysql'}, default 'sqlite'
  
  The flavor of SQL to use. Ignored when using SQLAlchemy engine. ‘mysql’ is deprecated and will be removed in future versions, but it will be further supported through SQLAlchemy engines.

- **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.Panel4D.transpose**

Panel4D.transpose(*args, **kwargs)

Permute the dimensions of the Panel

**Parameters**

- **args**: three positional arguments: each one of
  - \(0, 1, 2, \text{‘items’}, \text{‘major_axis’}, \text{‘minor_axis’}\)
  - **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.Panel4D.truediv**

Panel4D.truediv(other, axis=0)

Floating division of series and other, element-wise (binary operator truediv). Equivalent to panel / other.

**Parameters**

- **other**: Panel or Panel4D
- **axis**: \{labels, items, major_axis, minor_axis\}
  - Axis to broadcast over

**Returns**

- Panel4D

**See also:**

Panel4D.rtruediv

**pandas.Panel4D.truncate**

Panel4D.truncate(before=None, after=None, axis=None, copy=True)

Truncates a sorted NDFrame before and/or after some particular dates.

**Parameters**

- **before**: date
  - Truncate before date
- **after**: date
  - Truncate after date
- **axis**: the truncation axis, defaults to the stat axis
- **copy**: boolean, default is True,
pandas: powerful Python data analysis toolkit, Release 0.16.2

return a copy of the truncated section

**Returns truncated**: type of caller

**pandas.Panel4D.tshift**

Panel4D.tshift \((periods=1, freq=None, axis='major', **kwds)\)

**pandas.Panel4D.tz_convert**

Panel4D.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 is 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.Panel4D.tz_localize**

Panel4D.tz_localize \(*args, **kwargs\)
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 is 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)

Attempt to infer fall dst-transition hours based on order

**Raises** **TypeError**
If the TimeSeries is tz-aware and tz is not None.

**pandas.Panel4D.update**

Panel4D.update (other, join='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.Panel4D.var**

Panel4D.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**: \{labels (0), items (1), major_axis (2), minor_axis (3)\}
- **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 Panel
- **numeric_only**: boolean, default None
  - Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**
- **var**: Panel or Panel4D (if level specified)

**pandas.Panel4D.where**

Panel4D.where (cond, other=nan, inplace=False, axis=None, level=None, try_cast=False, raise_on_error=True)
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 or array
- **other**: scalar or NDFrame
- **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
- **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)

Returns **wh**: same type as caller

```
pandas.Panel4D.xs
```

Panel4D.xs(key, axis=1, copy=None)
Return slice of panel along selected axis

Parameters

- **key**: object
  Label
- **axis**: {'items', 'major', 'minor'}, default 1/'major'
- **copy**: boolean [deprecated]
  Whether to make a copy of the data

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 it is a superset of xs functionality, see MultiIndex Slicers

### 33.6.2 Attributes and underlying data

**Axes**

- **labels**: axis 1; each label corresponds to a Panel contained inside
- **items**: axis 2; each item corresponds to a DataFrame contained inside
- **major_axis**: axis 3; the index (rows) of each of the DataFrames
- **minor_axis**: axis 4; the columns of each of the DataFrames

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel4D.values</td>
<td>Numpy representation of NDFrame</td>
</tr>
<tr>
<td>Panel4D.axes</td>
<td>index(es) of the NDFrame</td>
</tr>
<tr>
<td>Panel4D.ndim</td>
<td>Number of axes / array dimensions</td>
</tr>
</tbody>
</table>

Continued on next page
Table 33.84 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel4D.size</td>
<td>number of elements in the NDFrame</td>
</tr>
<tr>
<td>Panel4D.shape</td>
<td>tuple of axis dimensions</td>
</tr>
<tr>
<td>Panel4D.dtypes</td>
<td>Return the dtypes in this object</td>
</tr>
<tr>
<td>Panel4D.ftypes</td>
<td>Return the ftypes (indication of sparse/dense and dtype) in this object.</td>
</tr>
<tr>
<td>Panel4D.get_dtype_counts()</td>
<td>Return the counts of dtypes in this object</td>
</tr>
<tr>
<td>Panel4D.get_ftype_counts()</td>
<td>Return the counts of ftypes in this object</td>
</tr>
</tbody>
</table>

**pandas.Panel4D.values**

Panel4D.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 upcase to int32.

**pandas.Panel4D.axes**

Panel4D.axes

index(es) of the NDFrame

**pandas.Panel4D.ndim**

Panel4D.ndim

Number of axes / array dimensions

**pandas.Panel4D.size**

Panel4D.size

number of elements in the NDFrame

**pandas.Panel4D.shape**

Panel4D.shape

tuple of axis dimensions

**pandas.Panel4D.dtypes**

Panel4D.dtypes

Return the dtypes in this object
pandas.Panel4D.ftypes

Panel4D.ftypes
Return the ftypes (indication of sparse/dense and dtype) in this object.

pandas.Panel4D.get_dtype_counts

Panel4D.get_dtype_counts()
Return the counts of dtypes in this object

pandas.Panel4D.get_ftype_counts

Panel4D.get_ftype_counts()
Return the counts of ftypes in this object

33.6.3 Conversion

Panel4D.astype(dtype[, copy, raise_on_error])
Cast object to input numpy.dtype

Parameters
dtype : numpy.dtype or Python type
raise_on_error : raise on invalid input

Returns
casted : type of caller

Panel4D.copy([deep])
Make a copy of this object

Parameters
deep : boolean or string, default True

Make a deep copy, i.e. also copy data

Returns
copy : type of caller

Panel4D.isnull()
Return a boolean same-sized object indicating if the values are null

See also:
notnull  boolean inverse of isnull

pandas.Panel4D.notnull

Panel4D.notnull()
Return a boolean same-sized object indicating if the values are not null

See also:

isnull  boolean inverse of notnull

33.7 Index

Many of these methods or variants thereof are available on the objects that contain an index (Series/Dataframe) and those should most likely be used before calling these methods directly.

| Index | Immutable ndarray implementing an ordered, sliceable set. |

33.7.1 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

Notes

An Index instance can only contain hashable objects

Attributes

| T | return the transpose, which is by definition self |
| base | return the base object if the memory of the underlying data is shared |
| data | return the data pointer of the underlying data |
| flags | has_duplicates | Continued on next page |
Table 33.87 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_monotonic</code></td>
<td>alias for <code>is_monotonic_increasing</code> (deprecated)</td>
</tr>
<tr>
<td><code>is_monotonic_decreasing</code></td>
<td>return if the index is monotonic decreasing (only equal or decreasing) values.</td>
</tr>
<tr>
<td><code>is_monotonic_increasing</code></td>
<td>return if the index is monotonic increasing (only equal or increasing) values.</td>
</tr>
<tr>
<td><code>itemsize</code></td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td><code>names</code></td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td><code>nbytes</code></td>
<td>return the number of dimensions of the underlying data, by definition 1</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td><code>nlevels</code></td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td><code>size</code></td>
<td>return the underlying data as an ndarray</td>
</tr>
</tbody>
</table>

**pandas.Index.T**

Index.T
   return the transpose, which is by definition self

**pandas.Index.base**

Index.base
   return the base object if the memory of the underlying data is shared

**pandas.Index.data**

Index.data
   return the data pointer of the underlying data

**pandas.Index.flags**

Index.flags

**pandas.Index.has_duplicates**

Index.has_duplicates

**pandas.Index.is_monotonic**

Index.is_monotonic
   alias for `is_monotonic_increasing` (deprecated)

**pandas.Index.is_monotonic_decreasing**

Index.is_monotonic_decreasing
   return if the index is monotonic decreasing (only equal or decreasing) values.
pandas.Index.is_monotonic_increasing

Index.is_monotonic_increasing
return if the index is monotonic increasing (only equal or increasing) values.

pandas.Index.itemsize

Index.itemsize
return the size of the dtype of the item of the underlying data

pandas.Index.names

Index.names

pandas.Index.nbytes

Index.nbytes
return the number of bytes in the underlying data

pandas.Index.ndim

Index.ndim
return the number of dimensions of the underlying data, by definition 1

pandas.Index.nlevels

Index.nlevels

pandas.Index.shape

Index.shape
return a tuple of the shape of the underlying data

pandas.Index.size

Index.size
return the number of elements in the underlying data

pandas.Index.strides

Index.strides
return the strides of the underlying data
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)  return an ndarray indexer of the 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)  Make a copy of this object.
copy([names, name, dtype, deep])  Make new Index with passed location(-s) deleted
delete(loc)  Compute sorted set difference of two Index objects
difference(other)  Make new Index with passed list of labels deleted
diff(*args, **kwargs)  Return Index with duplicate values removed
difference(other)  Determines if two Index objects contain the same elements.
astype(dtype)  Encode the object as an enumerated type or categorical variable
copy([names, name, dtype, deep])  Render a string representation of the Index
drop_duplicates(\[take_last\])  Compute indexer and mask for new index given the current index.
duplicated(\[take_last\])  guaranteed return of an indexer even when non-unique
drop(labels[, errors])  return an indexer suitable for taking from a non unique index
diff(*args, **kwargs)  return vector of label values for requested level, equal to the length
drop_duplicates(\[take_last\])  Get integer location for requested label
duplicated(\[take_last\])  Calculate slice bound that corresponds to given label.
diff(*args, **kwargs)  Fast lookup of value from 1-dimensional ndarray.
drop_duplicates(\[take_last\])  return the underlying data as an ndarray
duplicated(\[take_last\])  return index labels by a given array of values.
diff(*args, **kwargs)  return if I have any nans; enables various perf speedups
get_duplicates()  Similar to equals, but check that other comparable attributes are
get_indexer(target[, method, limit])  Make new Index inserting new item at location.
groupby(to_groupby)  Form the intersection of two Index objects.
hasnans()  More flexible, faster check like is but that works through views
identical(other)  holds_integer()  is_boolean()  is_categorical()  is_floating()
### pandas: powerful Python data analysis toolkit, Release 0.16.2

#### Table 33.88 – continued from previous page

- `is_integer()`
- `is_lexsorted_for_tuple(tup)`
- `is_mixed()`
- `is_numeric()`
- `is_object()`
- `is_type_compatible(kind)`
- `isin(values[, level])`
- `item()`
- `join(other[, how, level, return_indexers])`
- `map(mapper)`
- `max()`
- `min()`
- `nunique([dropna])`
- `order([return_indexer, ascending])`
- `putmask(mask, value)`
- `reindex(target[, method, level, limit])`
- `rename(name[, inplace])`
- `repeat(n)`
- `searchsorted(key[, side])`
- `set_names(names[, level, inplace])`
- `set_value(arr, key, value)`
- `shift([periods, freq])`
- `slice_indexer([start, end, step, kind])`
- `slice_locs([start, end, step, kind])`
- `sort(*args, **kwargs)`
- `str`
- `summary([name])`
- `sym_diff(other[, result_name])`
- `take(indexer[, axis])`
- `to_datetime([dayfirst])`
- `to_native_types([slicer])`
- `to_series(**kwargs)`
- `tolist()`
- `transpose()`
- `union(other)`
- `unique()`
- `value_counts([normalize, sort, ascending, ...])`
- `view([cls])`

#### Compute boolean array of whether each index value is found in the passed set of values. Return the first element of the underlying data as a Python scalar. This is an internal non-public method

#### The maximum value of the object

#### Return number of unique elements in the object.

#### Return sorted copy of Index

#### Return a new Index of the values set with the mask

#### Return an ndarray of the flattened values of the underlying data

#### Create index with target’s values (move/add/delete values as necessary)

#### Set new names on index

#### Return a new Index of the values repeated n times

#### np.ndarray searchsorted compat

#### Set new names on index

#### Fast lookup of value from 1-dimensional ndarray.

#### Shift Index containing datetime objects by input number of periods and for an ordered Index, compute the slice indexer for input labels and compute slice locations for input labels.

#### alias of StringMethods

#### Compute the sorted symmetric difference of two Index objects. Return a new Index of the values selected by the indexer

#### For an Index containing strings or datetime.datetime objects, attempt slice and dice then format

#### Create a Series with both index and values equal to the index keys return a list of the Index values

#### return the transpose, which is by definition self

#### Form the union of two Index objects and sorts if possible

#### Return array of unique values in the object.

#### Returns object containing counts of unique values.

**pandas.Index.all**

```python
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.
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.

pandas.Index.append

Index.append(other)
Append a collection of Index options together

Parameters: other: Index or list/tuple of indices
Returns: appended: Index

pandas.Index.argmax

Index.argmax(axis=None)
return a ndarray of the maximum argument indexer

See also:
        numpy.ndarray.argmax

pandas.Index.argmin

Index.argmin(axis=None)
return a ndarray of the minimum argument indexer

See also:
        numpy.ndarray.argmin

pandas.Index.argsort

Index.argsort(*args, **kwargs)
return an ndarray indexer of the underlying data

See also:
        numpy.ndarray.argsort

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'
pandas.Index.asof_locs

Index.asof_locs(where, mask)
where : array of timestamps mask : array of booleans where data is not NA

pandas.Index.astype

Index.astype(dtype)

pandas.Index.copy

Index.copy(names=None, name=None, dtype=None, deep=False)
Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters name : string, optional
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.

pandas.Index.delete

Index.delete(loc)
Make new Index with passed location(-s) deleted

Returns new_index : Index

pandas.Index.diff

Index.diff(*args, **kwargs)

pandas.Index.difference

Index.difference(other)
Compute sorted set difference of two Index objects

Parameters other : Index or array-like

Returns diff : Index

Notes

One can do either of these and achieve the same result

>>> index.difference(index2)
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

pandas.Index.drop_duplicates

Index.drop_duplicates(take_last=False)
Return Index with duplicate values removed

Parameters
take_last : boolean, default False
    Take the last observed index in a group. Default first

Returns
deduplicated : Index

pandas.Index.duplicated

Index.duplicated(take_last=False)
Return boolean np.array denoting duplicate values

Parameters
take_last : boolean, default False
    Take the last observed index in a group. Default first

Returns
duplicated : np.array

pandas.Index.equals

Index.equals(other)
Determines if two Index objects contain the same elements.

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
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)
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’}
  - 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
  Maximum number of consecutive labels in target to match for inexact matches.

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)
```

pandas.Index.get_indexer_for

Index.get_indexer_for(target, **kwargs)
Guaranteed return of an indexer even when non-unique

pandas.Index.get_indexer_non_unique

Index.get_indexer_non_unique(target)
Return an indexer suitable for taking from a non-unique index return the labels in the same order as the target, and return a missing indexer into the target (missing are marked as -1 in the indexer); target must be an iterable
pandas.Index.get_level_values

Index.get_level_values(level)
Return vector of label values for requested level, equal to the length of the index

Parameters level : int
Returns values : ndarray

pandas.Index.get_loc

Index.get_loc(key, method=)None)
Get integer location for requested label

Parameters key : label
    method : {None, ‘pad’/’ffill’, ‘backfill’/’bfill’, ‘nearest’}
        • 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.

Returns loc : int if unique index, possibly slice or mask if not

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==’right‘) position of given label.

Parameters label : object
    side : {‘left’, ‘right’}
    kind : string / None, the type of indexer

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

pandas.Index.get_values

Index.get_values()
return the underlying data as an ndarray
**pandas.Index.groupby**

Index.groupby(to_groupby)

Group the index labels by a given array of values.

**Parameters** to_groupby : array

Values used to determine the groups.

**Returns** groups : dict

{group name -> group labels}

**pandas.Index.hasnans**

Index.hasnans()

return if I have any nans; enables various perf speedups

**pandas.Index.holds_integer**

Index.holds_integer()

**pandas.Index.identical**

Index.identical(other)

Similar to equals, but check that other comparable attributes are also equal

**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

**pandas.Index.intersection**

Index.intersection(other)

Form the intersection of two Index objects. Sortedness of the result is not guaranteed

**Parameters** other : Index or array-like

**Returns** intersection : Index

**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

pandas.Index.is_boolean

Index.is_boolean()

pandas.Index.is_categorical

Index.is_categorical()

pandas.Index.is_floating

Index.is_floating()

pandas.Index.is_integer

Index.is_integer()

pandas.Index.is_lexsorted_for_tuple

Index.is_lexsorted_for_tuple(tup)

pandas.Index.is_mixed

Index.is_mixed()

pandas.Index.is_numeric

Index.is_numeric()

pandas.Index.is_object

Index.is_object()

pandas.Index.is_type_compatible

Index.is_type_compatible(kind)
**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 sequence of values
    - Sought values.
  - 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.

**pandas.Index.item**

Index.item()

return the first element of the underlying data as a python scalar

**pandas.Index.join**

Index.join(other, how='left', level=None, return_indexers=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

- **Returns**
  - join_index, (left_indexer, right_indexer)

**pandas.Index.map**

Index.map(mapper)

**pandas.Index.max**

Index.max()

The maximum value of the object
pandas.Index.min

Index.min()
The minimum value of the object

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

pandas.Index.order

Index.order(return_indexer=False, ascending=True)
Return sorted copy of Index

pandas.Index.putmask

Index.putmask(mask, value)
return a new Index of the values set with the mask

See also:
numpy.ndarray.putmask

pandas.Index.ravel

Index.ravel(order='C')
return an ndarray of the flattened values of the underlying data

See also:
numpy.ndarray.ravel

pandas.Index.reindex

Index.reindex(target, method=None, level=None, limit=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
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]

pandas.Index.repeat

Index.repeat(n)
return a new Index of the values repeated n times

See also:
numpy.ndarray.repeat

pandas.Index.searchsorted

Index.searchsorted(key, side='left')
np.ndarray searchsorted compat

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 or 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

>>> 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'])
```python
>>> 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'])
```

**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.

**pandas.Index.shift**

**Index.shift** *(periods=1, freq=None)*

Shift Index containing datetime objects by input number of periods and DateOffset

*Returns shifted*: Index

**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

**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
pandas: powerful Python data analysis toolkit, Release 0.16.2

step : int, defaults None
If None, defaults to 1
kind : string, defaults None

Returns start, end : int

pandas.Index.sort

Index.sort(*args, **kwargs)

pandas.Index.summary

Index.summary(name=None)

pandas.Index.sym_diff

Index.sym_diff(other, result_name=None)
Compute the sorted symmetric difference of two Index objects.

Parameters other : Index or array-like
result_name : str

Returns sym_diff : Index

Notes
sym_diff contains elements that appear in either idx1 or idx2 but not both. Equivalent to the Index created by (idx1 - idx2) + (idx2 - idx1) with duplicates dropped.
The sorting of a result containing NaN values is not guaranteed across Python versions. See GitHub issue #6444.

Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.sym_diff(idx2)
Int64Index([1, 5], dtype='int64')

You can also use the ^ operator:
```npy
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

pandas.Index.take

Index.take(indexer, axis=0)
return a new Index of the values selected by the indexer

See also:
```python
numpy.ndarray.take
```
pandas.Index.to_datetime

Index.to_datetime(dayfirst=False)
For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex

pandas.Index.to_native_types

Index.to_native_types(slicer=None, **kwargs)
slice and dice then format

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.

pandas.Index.tolist

Index.tolist()
return a list of the Index values

pandas.Index.transpose

Index.transpose()
return the transpose, which is by definition self

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

pandas.Index.unique

Index.unique()
Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

Returns uniques: ndarray

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

### pandas.Index.view

Index.*view*(cls=None)

#### 33.7.2 Attributes

<table>
<thead>
<tr>
<th>Attribute</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>
<tr>
<td>Index.inferred_type</td>
<td></td>
</tr>
<tr>
<td>Index.is_all_dates</td>
<td></td>
</tr>
<tr>
<td>Index.shape</td>
<td>return a tuple of the shape of the underlying data</td>
</tr>
<tr>
<td>Index.nbytes</td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td>Index.ndim</td>
<td>return the number of dimensions of the underlying data, by definition 1</td>
</tr>
<tr>
<td>Index.size</td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td>Index.strides</td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td>Index.itemsize</td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td>Index.base</td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td>Index.T</td>
<td>return the transpose, which is by definition self</td>
</tr>
</tbody>
</table>

### pandas.Index.values

Index.*values*

return the underlying data as an ndarray
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.Index.is_monotonic

Index.is_monotonic
    alias for is_monotonic_increasing (deprecated)

pandas.Index.is_monotonic_increasing

Index.is_monotonic_increasing
    return if the index is monotonic increasing (only equal or increasing) values.

pandas.Index.is_monotonic_decreasing

Index.is_monotonic_decreasing
    return if the index is monotonic decreasing (only equal or decreasing) values.

pandas.Index.is_unique

Index.is_unique = None

pandas.Index.has_duplicates

Index.has_duplicates

pandas.Index.dtype

Index.dtype = None

pandas.Index.inferred_type

Index.inferred_type = None

pandas.Index.is_all_dates

Index.is_all_dates = None

pandas.Index.shape

Index.shape
    return a tuple of the shape of the underlying data

pandas.Index.nbytes

Index.nbytes
    return the number of bytes in the underlying data
pandas.Index.ndim

Index.ndim
return the number of dimensions of the underlying data, by definition 1

pandas.Index.size

Index.size
return the number of elements in the underlying data

pandas.Index.strides

Index.strides
return the strides of the underlying data

pandas.Index.itemsize

Index.itemsize
return the size of the dtype of the item of the underlying data

pandas.Index.base

Index.base
return the base object if the memory of the underlying data is shared

pandas.Index.T

Index.T
return the transpose, which is by definition self

33.7.3 Modifying and Computations

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.all(*args, **kwargs)</td>
<td>Return whether all elements are True</td>
<td></td>
</tr>
<tr>
<td>Index.any(*args, **kwargs)</td>
<td>Return whether any element is True</td>
<td></td>
</tr>
<tr>
<td>Index.argmin([axis])</td>
<td>Return a ndarray of the minimum argument indexer</td>
<td></td>
</tr>
<tr>
<td>Index.argmax([axis])</td>
<td>Return a ndarray of the maximum argument indexer</td>
<td></td>
</tr>
<tr>
<td>Index.copy([names, name, dtype, deep])</td>
<td>Make a copy of this object.</td>
<td></td>
</tr>
<tr>
<td>Index.delete(loc)</td>
<td>Make new Index with passed location(-s) deleted</td>
<td></td>
</tr>
<tr>
<td>Index.drop([labels[, errors]])</td>
<td>Make new Index with passed list of labels deleted</td>
<td></td>
</tr>
<tr>
<td>Index.drop_duplicates([take_last])</td>
<td>Return Index with duplicate values removed</td>
<td></td>
</tr>
<tr>
<td>Index.duplicated([take_last])</td>
<td>Return boolean np.array denoting duplicate values</td>
<td></td>
</tr>
<tr>
<td>Index.equals(other)</td>
<td>Determines if two Index objects contain the same elements.</td>
<td></td>
</tr>
<tr>
<td>Index.factorize([sort, na_sentinel])</td>
<td>Encode the object as an enumerated type or categorical variable</td>
<td></td>
</tr>
<tr>
<td>Index.identical(other)</td>
<td>Similar to equals, but check that other comparable attributes are</td>
<td></td>
</tr>
<tr>
<td>Index.insert(loc, item)</td>
<td>Make new Index inserting new item at location.</td>
<td></td>
</tr>
<tr>
<td>Index.min()</td>
<td>The minimum value of the object</td>
<td></td>
</tr>
<tr>
<td>Index.max()</td>
<td>The maximum value of the object</td>
<td></td>
</tr>
<tr>
<td>Index.order([return_indexer, ascending])</td>
<td>Return sorted copy of Index</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 33.90 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.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>Index.repeat(n)</code></td>
<td>Return a new Index of the values repeated n times</td>
</tr>
<tr>
<td><code>Index.take(indexer[, axis])</code></td>
<td>Return a new Index of the values selected by the indexer</td>
</tr>
<tr>
<td><code>Index.putmask(mask, value)</code></td>
<td>Return a new Index of the values set with the mask</td>
</tr>
<tr>
<td><code>Index.set_names(names[, level, inplace])</code></td>
<td>Set new names on index.</td>
</tr>
<tr>
<td><code>Index.unique()</code></td>
<td>Return array of unique values in the object.</td>
</tr>
<tr>
<td><code>Index.nunique([dropna])</code></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><code>Index.value_counts([normalize, sort, ...])</code></td>
<td>Returns object containing counts of unique values.</td>
</tr>
</tbody>
</table>

**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.

**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.

**pandas.Index.argmin**

`Index.argmin(axis=None)`

Return a ndarray of the minimum argument indexer

**See also**

- `numpy.ndarray.argmin`

**pandas.Index.argmax**

`Index.argmax(axis=None)`

Return a ndarray of the maximum argument indexer

**See also**

- `numpy.ndarray.argmax`

**pandas.Index.copy**

`Index.copy(names=None, name=None, dtype=None, deep=False)`

Make a copy of this object. Name and dtype sets those attributes on the new object.
**Parameters**

- **name**: string, optional
- **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`.

**pandas.Index.delete**

Index.delete(loc)

Make new Index with passed location(s) deleted

**Returns**

- **new_index**: Index

**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

**pandas.Index.drop_duplicates**

Index.drop_duplicates(take_last=False)

Return Index with duplicate values removed

**Parameters**

- **take_last**: boolean, default False
  
  Take the last observed index in a group. Default first

**Returns**

- **deduplicated**: Index

**pandas.Index.duplicated**

Index.duplicated(take_last=False)

Return boolean np.array denoting duplicate values

**Parameters**

- **take_last**: boolean, default False
  
  Take the last observed index in a group. Default first

**Returns**

- **duplicated**: np.array
pandas: powerful Python data analysis toolkit, Release 0.16.2

**pandas.Index.equals**

Index equals (other)
Determines if two Index objects contain the same elements.

**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

**pandas.Index.identical**

Index identical (other)
Similar to equals, but check that other comparable attributes are also equal

**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

**pandas.Index.min**

Index min()
The minimum value of the object

**pandas.Index.max**

Index max()
The maximum value of the object

**pandas.Index.order**

Index order (return_indexer=False, ascending=True)
Return sorted copy of Index
pandas.Index.reindex

Index.reindex(target, method=None, level=None, limit=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

pandas.Index.repeat

Index.repeat(n)
return a new Index of the values repeated n times

See also:
numpy.ndarray.repeat

pandas.Index.take

Index.take(indexer, axis=0)
return a new Index of the values selected by the indexer

See also:
numpy.ndarray.take

pandas.Index.putmask

Index.putmask(mask, value)
return a new Index of the values set with the mask

See also:
numpy.ndarray.putmask

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 or 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'])
```

pandas.Index.unique

Index.unique()

Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

Returns: uniques : ndarray

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

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

33.7.4 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.astype(dtype)</td>
<td>return a list of the Index values</td>
</tr>
<tr>
<td>Index.tolist()</td>
<td>return a list of the Index values</td>
</tr>
<tr>
<td>Index.to_datetime(dayfirst)</td>
<td>For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex</td>
</tr>
<tr>
<td>Index.to_series(**kwargs)</td>
<td>Create a Series with both index and values equal to the index keys actually</td>
</tr>
</tbody>
</table>

pandas.Index.astype

Index.astype(dtype)

pandas.Index.tolist

Index.tolist()
return a list of the Index values

pandas.Index.to_datetime

Index.to_datetime(dayfirst=False)
For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex

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.

33.7.5 Sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.argsort(*args, **kwargs)</td>
<td>return ndarray indexer of the underlying data</td>
</tr>
<tr>
<td>Index.order([return_indexer, ascending])</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>Index.sort(*args, **kwargs)</td>
<td></td>
</tr>
</tbody>
</table>
pandas.Index.argsort

Index.argsort(*args, **kwargs)
return an ndarray indexer of the underlying data

See also:
numpy.ndarray.argsort

pandas.Index.order

Index.order(return_indexer=False, ascending=True)
Return sorted copy of Index

pandas.Index.sort

Index.sort(*args, **kwargs)

33.7.6 Time-specific operations

Index.shift([periods, freq]) Shift Index containing datetime objects by input number of periods and

pandas.Index.shift

Index.shift(periods=1, freq=None)
Shift Index containing datetime objects by input number of periods and DateOffset
Returns shifted : Index

33.7.7 Combining / joining / set operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.append(other)</td>
<td>Append a collection of Index options together</td>
</tr>
<tr>
<td>Index.join(other[, how, level, return_indexers])</td>
<td>this is an internal non-public method</td>
</tr>
<tr>
<td>Index.intersection(other)</td>
<td>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td>Index.union(other)</td>
<td>Form the union of two Index objects and sorts if possible</td>
</tr>
<tr>
<td>Index.difference(other)</td>
<td>Compute sorted set difference of two Index objects</td>
</tr>
<tr>
<td>Index.sym_diff(other[, result_name])</td>
<td>Compute the sorted symmetric difference of two Index objects.</td>
</tr>
</tbody>
</table>

pandas.Index.append

Index.append(other)
Append a collection of Index options together

Parameters other : Index or list/tuple of indices

Returns appended : Index
pandas.Index.join

Index.join(other, how='left', level=None, return_indexers=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

Returns

join_index, (left_indexer, right_indexer)

pandas.Index.intersection

Index.intersection(other)

Form the intersection of two Index objects. Sortedness of the result is not guaranteed.

Parameters

other : Index or array-like

Returns

intersection

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

pandas.Index.difference

Index.difference(other)

Compute sorted set difference of two Index objects.

Parameters

other : Index or array-like

Returns

diff

Notes

One can do either of these and achieve the same result

>>> index.difference(index2)

pandas.Index.sym_diff

Index.sym_diff(other, result_name=None)

Compute the sorted symmetric difference of two Index objects.
Parameters other : Index or array-like
result_name : str

Returns sym_diff : Index

Notes

sym_diff contains elements that appear in either idx1 or idx2 but not both. Equivalent to the Index created by (idx1 - idx2) + (idx2 - idx1) with duplicates dropped.

The sorting of a result containing NaN values is not guaranteed across Python versions. See GitHub issue #6444.

Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.sym_diff(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the ^ operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

33.7.8 Selecting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Index.get_indexer</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>pandas.Index.get_indexer_non_unique</td>
<td>return an indexer suitable for taking from a non unique index</td>
</tr>
<tr>
<td>pandas.Index.get_level_values</td>
<td>Return vector of label values for requested level, equal to the length</td>
</tr>
<tr>
<td>pandas.Index.get_loc</td>
<td>Get integer location for requested label</td>
</tr>
<tr>
<td>pandas.Index.get_value</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>pandas.Index.isin</td>
<td>Compute boolean array of whether each index value is found in the passed set of values</td>
</tr>
<tr>
<td>pandas.Index.slice_indexer</td>
<td>For an ordered Index, compute the slice indexer for input labels and</td>
</tr>
<tr>
<td>pandas.Index.slice_locs</td>
<td>Compute slice locations for input labels.</td>
</tr>
</tbody>
</table>

pandas.Index.get_indexer

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’}

• 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

Maximum number of consecutive labels in target to match for inexact matches.

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)
```

**pandas.Index.get_indexer_non_unique**

Index.get_indexer_non_unique(target)

Return an indexer suitable for taking from a non unique index return the labels in the same order as the target, and return a missing indexer into the target (missing are marked as -1 in the indexer); target must be an iterable

**pandas.Index.get_level_values**

Index.get_level_values(level)

Return vector of label values for requested level, equal to the length of the index

Parameters level: int

Returns values: ndarray

**pandas.Index.get_loc**

Index.get_loc(key, method=None)

Get integer location for requested label

Parameters key: label

method: {None, ‘pad’/’ffill’, ‘backfill’/’bfill’, ‘nearest’}

• 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.

Returns loc: int if unique index, possibly slice or mask if not

**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
**pandas.Index.isin**

Index.

\[ \text{isin} \left( \text{values}, \text{level}=\text{None} \right) \]

Compute boolean array of whether each index value is found in the passed set of values.

**Parameters**

- **values**: set or sequence of values
  - Sought values.
- **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.

**pandas.Index.slice_indexer**

Index.

\[ \text{slice_indexer} \left( \text{start}=\text{None}, \text{end}=\text{None}, \text{step}=\text{None}, \text{kind}=\text{None} \right) \]

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

**pandas.Index.slice_locs**

Index.

\[ \text{slice_locs} \left( \text{start}=\text{None}, \text{end}=\text{None}, \text{step}=\text{None}, \text{kind}=\text{None} \right) \]

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

**Notes**

This function assumes that the data is sorted, so use at your own peril
If None, defaults to 1

kind : string, defaults None

Returns start, end : int

## 33.8 CategoricalIndex

CategoricalIndex Immutable Index implementing an ordered, sliceable set.

### 33.8.1 pandas.CategoricalIndex

class pandas.CategoricalIndex

Immutable Index implementing an ordered, sliceable set. CategoricalIndex represents a sparsely populated Index with an underlying Categorical.

New in version 0.16.1.

**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

**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>base</td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td>categories</td>
<td>return the data pointer of the underlying data</td>
</tr>
<tr>
<td>codes</td>
<td>alias for is_monotonic_increasing (deprecated)</td>
</tr>
<tr>
<td>data</td>
<td>return if the index is monotonic decreasing (only equal or</td>
</tr>
<tr>
<td>flags</td>
<td>return if the index is monotonic increasing (only equal or</td>
</tr>
<tr>
<td>has_duplicates</td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td>inferred_type</td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td>is_monotonic</td>
<td>return the number of dimensions of the underlying data, by definition 1</td>
</tr>
<tr>
<td>is_monotonic_decreasing</td>
<td></td>
</tr>
<tr>
<td>is_monotonic_increasing</td>
<td></td>
</tr>
<tr>
<td>itemsize</td>
<td></td>
</tr>
<tr>
<td>names</td>
<td></td>
</tr>
<tr>
<td>nbytes</td>
<td></td>
</tr>
<tr>
<td>ndim</td>
<td></td>
</tr>
<tr>
<td>nlevels</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### pandas.CategoricalIndex.T

- **ordered**: return a tuple of the shape of the underlying data
- **shape**: return the number of elements in the underlying data
- **size**: return the strides of the underlying data
- **strides**: return the underlying data, which is a Categorical

### pandas.CategoricalIndex.base

- **ordered**: return the transpose, which is by definition self

### pandas.CategoricalIndex.categories

### pandas.CategoricalIndex.codes

### pandas.CategoricalIndex.data

### pandas.CategoricalIndex.flags

### pandas.CategoricalIndex.has_duplicates

### pandas.CategoricalIndex.inferred_type

### pandas.CategoricalIndex.is_monotonic

- **ordered**: alias for is_monotonic_increasing (deprecated)
pandas.CategoricalIndex.is_monotonic_decreasing

CategoricalIndex.is_monotonic_decreasing
return if the index is monotonic decreasing (only equal or decreasing) values.

pandas.CategoricalIndex.is_monotonic_increasing

CategoricalIndex.is_monotonic_increasing
return if the index is monotonic increasing (only equal or increasing) values.

pandas.CategoricalIndex.itemsize

CategoricalIndex.itemsize
return the size of the dtype of the item of the underlying data

pandas.CategoricalIndex.names

CategoricalIndex.names

pandas.CategoricalIndex.nbytes

CategoricalIndex.nbytes
return the number of bytes in the underlying data

pandas.CategoricalIndex.ndim

CategoricalIndex.ndim
return the number of dimensions of the underlying data, by definition 1

pandas.CategoricalIndex.nlevels

CategoricalIndex.nlevels

pandas.CategoricalIndex.ordered

CategoricalIndex.ordered

pandas.CategoricalIndex.shape

CategoricalIndex.shape
return a tuple of the shape of the underlying data

pandas.CategoricalIndex.size

CategoricalIndex.size
return the number of elements in the underlying data
pandas.CategoricalIndex.strides

CategoricalIndex.strides
return the strides of the underlying data

pandas.CategoricalIndex.values

CategoricalIndex.values
return the underlying data, which is a Categorical

| asi8  
| dtype  
| is_all_dates  
| is_unique  
| name  

Methods

add_categories(*args, **kwargs) Add new categories.
all([other]) Append a collection of CategoricalIndex options together
any([other]) return a ndarray of the maximum argument indexer
append(other) return a ndarray of the minimum argument indexer
argmax([axis]) Sets the Categorical to be ordered
argmin([axis]) Sets the Categorical to be unordered
argsort(*args, **kwargs) For a sorted index, return the most recent label up to and including the passed
as_ordered(*args, **kwargs) where : array of timestamps
as_unordered(*args, **kwargs) Make a copy of this object.
asof(label) Make new Index with passed location(-s) deleted
asof_locs(where, mask) Compute sorted set difference of two Index objects
drop_duplicates([take_last]) Make new Index with passed list of labels deleted
drop(*labels[, errors]) Return Index with duplicate values removed
drop_duplicates([take_last]) Determines if two CategoricalIndex objects contain the same elements.
duplicated([take_last]) Encode the object as an enumerated type or categorical variable
diff(*args, **kwargs) Render a string representation of the Index
duplicate([names, name, dtype, deep]) For a sorted index, return the most recent label up to and including the passed
equals(other) where : array of timestamps
factorize([sort, na_sentinel]) Encode the object as an enumerated type or categorical variable
format([name, formatter]) Render a string representation of the Index
groupby(to_groupby) Render a string representation of the Index
groupby(to_groupby) get_dummies() Compute indexer and mask for new index given the current index.
get_duplicates() guaranteed return of an indexer even when non-unique
get_indexer(target[, method, limit]) this is the same for a CategoricalIndex for get_indexer; the API returns the mask
get_indexer_for(target, **kwargs) Return vector of label values for requested level, equal to the length
get_level_values(level) Get integer location for requested label
get_loc(key[, method]) Calculate slice bound that corresponds to given label.
gles_bound([label, side, kind]) Fast lookup of value from 1-dimensional ndarray.
gles_bound([label, side, kind]) return the underlying data as an ndarray
get_value(series, key) Group the index labels by a given array of values.
gles_bound([label, side, kind]) return if I have any nans; enables various perf speedups
get_values()
Table 33.98 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>holds_integer()</code></td>
<td>Similar to equals, but check that other comparable attributes are identical (other)</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>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td><code>is_(other)</code></td>
<td>More flexible, faster check like <code>is</code> but that works through views</td>
</tr>
<tr>
<td><code>is_boolean()</code></td>
<td>Compute boolean array of whether each index value is found in the passed set.</td>
</tr>
<tr>
<td><code>is_categorical()</code></td>
<td>return the first element of the underlying data as a python scalar</td>
</tr>
<tr>
<td><code>is_floating()</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></td>
</tr>
<tr>
<td><code>item()</code></td>
<td></td>
</tr>
<tr>
<td><code>join(other[, how, level, return_indexers])</code></td>
<td></td>
</tr>
<tr>
<td><code>max(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>min(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>nunique([dropna])</code></td>
<td></td>
</tr>
<tr>
<td><code>order([return_indexer, ascending])</code></td>
<td></td>
</tr>
<tr>
<td><code>putmask(mask, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td></td>
</tr>
<tr>
<td><code>reindex(target[, method, level, limit])</code></td>
<td></td>
</tr>
<tr>
<td><code>remove_categories(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>remove_unused_categories(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>rename(name[, inplace])</code></td>
<td></td>
</tr>
<tr>
<td><code>rename_categories(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>reorder_categories(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>repeat(n)</code></td>
<td></td>
</tr>
<tr>
<td><code>searchsorted(key[, side])</code></td>
<td></td>
</tr>
<tr>
<td><code>set_categories(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_indexer([start, end, step, kind])</code></td>
<td></td>
</tr>
<tr>
<td><code>set_value(arr, key, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>sort(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>str</code></td>
<td>alias of StringMethods</td>
</tr>
<tr>
<td><code>summary([name])</code></td>
<td></td>
</tr>
<tr>
<td><code>sym_diff(other[, result_name])</code></td>
<td></td>
</tr>
<tr>
<td><code>take(indexer[, axis])</code></td>
<td></td>
</tr>
<tr>
<td><code>to_datetime([dayfirst])</code></td>
<td></td>
</tr>
<tr>
<td><code>to_native_types([slicer])</code></td>
<td></td>
</tr>
<tr>
<td><code>to_series(**kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Compute the sorted symmetric difference of two Index objects.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>return a new CategoricalIndex of the values selected by the indexer</td>
</tr>
<tr>
<td><code>union(other)</code></td>
<td>For an Index containing strings or datetime.datetime objects, attempt slice and dice then format</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Create a Series with both index and values equal to the index keys return a list of the Index values</td>
</tr>
<tr>
<td><code>value_counts([normalize, sort, ascending, ...])</code></td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td><code>view([cls])</code></td>
<td>Form the union of two Index objects and sorts if possible</td>
</tr>
</tbody>
</table>

33.8. CategoricalIndex

1467
**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

**pandas.CategoricalIndex.all**

CategoricalIndex.all(other=None)

**pandas.CategoricalIndex.any**

CategoricalIndex.any(other=None)

**pandas.CategoricalIndex.append**

CategoricalIndex.append(other)

Append a collection of CategoricalIndex options together

**Parameters**

other : Index or list/tuple of indices

**Returns**

appended : Index

**Raises**

 ValueError if other is not in the categories

**pandas.CategoricalIndex.argmax**

CategoricalIndex.argmax(axis=None)

return a ndarray of the maximum argument indexer

See also:

numpy.ndarray.argmax
pandas.CategoricalIndex.argmin

CategoricalIndex.argmin(\texttt{axis=None})
return a ndarray of the minimum argument indexer

See also:
numpy.ndarray.argmin

pandas.CategoricalIndex.argsort

CategoricalIndex.argsort(*\texttt{args, **kwargs})

pandas.CategoricalIndex.as_ordered

CategoricalIndex.as_ordered(*\texttt{args, **kwargs})
Sets the Categorical to be ordered

Parameters \texttt{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

pandas.CategoricalIndex.as_unordered

CategoricalIndex.as_unordered(*\texttt{args, **kwargs})
Sets the Categorical to be unordered

Parameters \texttt{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

pandas.CategoricalIndex.asof

CategoricalIndex.asof(\texttt{label})
For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.

See also:

\texttt{get\_loc} asof is a thin wrapper around get\_loc with method='pad'

pandas.CategoricalIndex.asof_locs

CategoricalIndex.asof_locs(\texttt{where, mask})
where : array of timestamps mask : array of booleans where data is not NA

pandas.CategoricalIndex.astype

CategoricalIndex.astype(\texttt{dtype})
**pandas.CategoricalIndex.copy**

CategoricalIndex.copy (names=None, name=None, dtype=None, deep=False)

Make a copy of this object. Name and dtype sets those attributes on the new object.

**Parameters**
- name : string, optional
- 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.

**pandas.CategoricalIndex.delete**

CategoricalIndex.delete(loc)

Make new Index with passed location(-s) deleted

**Returns**
- new_index : Index

**pandas.CategoricalIndex.diff**

CategoricalIndex.diff(*args, **kwargs)

**pandas.CategoricalIndex.difference**

CategoricalIndex.difference(other)

Compute sorted set difference of two Index objects

**Parameters**
- other : Index or array-like

**Returns**
- diff : Index

**Notes**

One can do either of these and achieve the same result

```python
>>> index.difference(index2)
```

**pandas.CategoricalIndex.drop**

CategoricalIndex.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
pandas.CategoricalIndex.drop_duplicates

CategoricalIndex.drop_duplicates(take_last=False)
Return Index with duplicate values removed

Parameters  take_last : boolean, default False
Take the last observed index in a group. Default first

Returns  deduplicated  : Index

pandas.CategoricalIndex.duplicated

CategoricalIndex.duplicated(take_last=False)
Return boolean np.array denoting duplicate values

Parameters  take_last : boolean, default False
Take the last observed index in a group. Default first

Returns  duplicated  : np.array

pandas.CategoricalIndex.equals

CategoricalIndex.equals(other)
Determines if two CategorialIndex objects contain the same elements.

pandas.CategoricalIndex.factorize

CategoricalIndex.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.CategoricalIndex.format

CategoricalIndex.format(name=False, formatter=None, **kwargs)
Render a string representation of the Index

pandas.CategoricalIndex.get_duplicates

CategoricalIndex.get_duplicates()
pandas.CategoricalIndex.get_indexer

CategoricalIndex.get_indexer (target, method=None, limit=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. The mask determines whether labels are found or not in the current index.

Parameters  
- target : MultiIndex or Index (of tuples)  
- method : {'pad', 'ffill', 'backfill', 'bfill'}  
  - pad / ffill: propagate LAST valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

Returns  
- (indexer, mask) : (ndarray, ndarray)

Notes

This is a low-level method and probably should be used at your own risk

Examples

```python
>>> indexer, mask = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
>>> new_values[-mask] = np.nan
```

pandas.CategoricalIndex.get_indexer_for

CategoricalIndex.get_indexer_for (target, **kwargs)  
guaranteed return of an indexer even when non-unique

pandas.CategoricalIndex.get_indexer_non_unique

CategoricalIndex.get_indexer_non_unique (target)  
this is the same for a CategoricalIndex for get_indexer; the API returns the missing values as well

pandas.CategoricalIndex.get_level_values

CategoricalIndex.get_level_values (level)  
Return vector of label values for requested level, equal to the length of the index.

Parameters  
- level : int

Returns  
- values : ndarray

pandas.CategoricalIndex.get_loc

CategoricalIndex.get_loc (key, method=None)  
Get integer location for requested label

Parameters  
- key : label  
- method : {None}
• default: exact matches only.

**Returns**

- **loc**: int if unique index, possibly slice or mask if not

### pandas.CategoricalIndex.get_slice_bound

*CategoricalIndex.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**: string / None, the type of indexer

### pandas.CategoricalIndex.get_value

*CategoricalIndex.get_value*(series, key)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you’re doing

### pandas.CategoricalIndex.get_values

*CategoricalIndex.get_values*( )

Return the underlying data as an ndarray

### pandas.CategoricalIndex.groupby

*CategoricalIndex.groupby*(to_groupby)

Group the index labels by a given array of values.

- **Parameters**
  - **to_groupby**: array
    - Values used to determine the groups.

- **Returns**
  - **groups**: dict
    - {group name -> group labels}

### pandas.CategoricalIndex.hasnans

*CategoricalIndex.hasnans*( )

Return if I have any nans; enables various perf speedups

### pandas.CategoricalIndex.holds_integer

*CategoricalIndex.holds_integer*( )

### pandas.CategoricalIndex.identical

*CategoricalIndex.identical*(other)

Similar to equals, but check that other comparable attributes are also equal
pandas.CategoricalIndex.insert

CategoricalIndex.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

Raises
- ValueError if the item is not in the categories

pandas.CategoricalIndex.intersection

CategoricalIndex.intersection(other)

- Form the intersection of two Index objects. Sortedness of the result is not guaranteed

Parameters
- other : Index or array-like

Returns
- intersection : Index

pandas.CategoricalIndex.is

CategoricalIndex.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

pandas.CategoricalIndex.is_boolean

CategoricalIndex.is_boolean()

pandas.CategoricalIndex.is_categorical

CategoricalIndex.is_categorical()

pandas.CategoricalIndex.is_floating

CategoricalIndex.is_floating()

pandas.CategoricalIndex.is_integer

CategoricalIndex.is_integer()

pandas.CategoricalIndex.is_lexsorted_for_tuple

CategoricalIndex.is_lexsorted_for_tuple(tup)
pandas.CategoricalIndex.is_mixed
CategoricalIndex.is_mixed()
pandas.CategoricalIndex.is_numeric
CategoricalIndex.is_numeric()
pandas.CategoricalIndex.is_object
CategoricalIndex.is_object()
pandas.CategoricalIndex.is_type_compatible
CategoricalIndex.is_type_compatible(kind)
pandas.CategoricalIndex.isin
CategoricalIndex.isin(values, level=None)
Compute boolean array of whether each index value is found in the passed set of values.

Parameters values : set or sequence of values
Sought values.
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.
pandas.CategoricalIndex.item
CategoricalIndex.item()
return the first element of the underlying data as a python scalar
pandas.CategoricalIndex.join
CategoricalIndex.join(other, how='left', level=None, return_indexers=False)
this is an internal non-public method
Compute join_index and indexers to conform data structures to the new index.
pandas: powerful Python data analysis toolkit, Release 0.16.2

Parameters  **other** : Index
  how : {'left', 'right', 'inner', 'outer'}
  level : int or level name, default None
  **return_indexers** : boolean, default False

Returns  join_index, (left_indexer, right_indexer)

```python
pandas.CategoricalIndex.map
```

```python
CategoricalIndex.map(mapper)
```

```python
pandas.CategoricalIndex.max
```

```python
CategoricalIndex.max(*args, **kwargs)
```

The maximum value of the object.

Only ordered *Categoricals* have a maximum!

Returns  max : the maximum of this *Categorical*

Raises  TypeError
  If the *Categorical* is not ordered.

```python
pandas.CategoricalIndex.min
```

```python
CategoricalIndex.min(*args, **kwargs)
```

The minimum value of the object.

Only ordered *Categoricals* have a minimum!

Returns  min : the minimum of this *Categorical*

Raises  TypeError
  If the *Categorical* is not ordered.

```python
pandas.CategoricalIndex.nunique
```

```python
CategoricalIndex.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

```python
pandas.CategoricalIndex.order
```

```python
CategoricalIndex.order(return_indexer=True, ascending=True)
```

Return sorted copy of Index
pandas.CategoricalIndex.putmask

CategoricalIndex.putmask(mask, value)
return a new Index of the values set with the mask

See also:

numpy.ndarray.putmask

pandas.CategoricalIndex.ravel

CategoricalIndex.ravel(order='C')
return an ndarray of the flattened values of the underlying data

See also:

numpy.ndarray.ravel

pandas.CategoricalIndex.reindex

CategoricalIndex.reindex(target, method=None, level=None, limit=None)
Create index with target’s values (move/add/delete values as necessary)

Returns new_index : pd.Index
Resulting index

indexer : np.ndarray or None
Indices of output values in original index

pandas.CategoricalIndex.remove_categories

CategoricalIndex.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
**pandas.CategoricalIndex.remove_unused_categories**

CategoricalIndex.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 catego-
  rical 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

**pandas.CategoricalIndex.rename**

CategoricalIndex.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]

**pandas.CategoricalIndex.rename_categories**

CategoricalIndex.rename_categories(*args, **kwargs)

Renames categories.

The new categories 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.

**Parameters**

- **new_categories** : Index-like
  The renamed categories.

- **inplace** : boolean (default: False)
  Whether or not to rename the categories inplace or return a copy of this catego-
  rical with renamed categories.

**Returns**

- **cat** : Categorical with renamed categories added or None if inplace.

**Raises**

ValueError

If the new categories 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
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

pandas.CategoricalIndex.repeat

CategoricalIndex.repeat(n)
return a new Index of the values repeated n times

See also:
   numpy.ndarray.repeat

pandas.CategoricalIndex.searchsorted

CategoricalIndex.searchsorted(key, side=’left’)
np.ndarray searchsorted compat

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 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

### pandas.CategoricalIndex.set_names

CategoricalIndex.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 or 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')],
```
pandas: powerful Python data analysis toolkit, Release 0.16.2

>>> 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'])

pandas.CategoricalIndex.set_value

CategoricalIndex.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

pandas.CategoricalIndex.shift

CategoricalIndex.shift(periods=1, freq=None)

Shift Index containing datetime objects by input number of periods and DateOffset

Returns shifted : Index

pandas.CategoricalIndex.slice_indexer

CategoricalIndex.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

der : 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

pandas.CategoricalIndex.slice_locs

CategoricalIndex.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

der : label, default None


If None, defaults to the end

**step**: int, defaults None

If None, defaults to 1

**kind**: string, defaults None

Returns `start, end`: int

### pandas.CategoricalIndex.sort

CategoricalIndex.sort(*args, **kwargs)

### pandas.CategoricalIndex.summary

CategoricalIndex.summary(name=None)

### pandas.CategoricalIndex.sym_diff

CategoricalIndex.sym_diff(other, result_name=None)

Compute the sorted symmetric difference of two Index objects.

**Parameters**

- **other**: Index or array-like
- **result_name**: str

Returns **sym_diff**: Index

### Notes

`sym_diff` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `(idx1 - idx2) + (idx2 - idx1)` with duplicates dropped.

The sorting of a result containing NaN values is not guaranteed across Python versions. See GitHub issue #6444.

### Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.sym_diff(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```
pandas.CategoricalIndex.take

CategoricalIndex.take(indexer, axis=0)
    return a new CategoricalIndex of the values selected by the indexer

    See also:
    numpy.ndarray.take

pandas.CategoricalIndex.to_datetime

CategoricalIndex.to_datetime(dayfirst=False)
    For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex

pandas.CategoricalIndex.to_native_types

CategoricalIndex.to_native_types(slicer=None, **kwargs)
    slice and dice then format

pandas.CategoricalIndex.to_series

CategoricalIndex.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.

pandas.CategoricalIndex.tolist

CategoricalIndex.tolist()
    return a list of the Index values

pandas.CategoricalIndex.transpose

CategoricalIndex.transpose()
    return the transpose, which is by definition self

pandas.CategoricalIndex.union

CategoricalIndex.union(other)
    Form the union of two Index objects and sorts if possible

    Parameters other: Index or array-like

    Returns union: Index

pandas.CategoricalIndex.unique

CategoricalIndex.unique()
    Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

    Returns uniques: ndarray
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.CategoricalIndex.value_counts

CategoricalIndex.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.CategoricalIndex.view

CategoricalIndex.view(cls=None)

33.8.2 Categorical Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CategoricalIndex.codes</td>
<td>Renames categories.</td>
</tr>
<tr>
<td>CategoricalIndex.categories</td>
<td></td>
</tr>
<tr>
<td>CategoricalIndex.ordered</td>
<td></td>
</tr>
<tr>
<td>CategoricalIndex.rename_categories(*args, ...)</td>
<td>Reorders categories as specified in new_categories.</td>
</tr>
<tr>
<td>CategoricalIndex.reorder_categories(*args, ...)</td>
<td>Add new categories.</td>
</tr>
<tr>
<td>CategoricalIndex.add_categories(*args, **kwargs)</td>
<td>Removes the specified categories.</td>
</tr>
<tr>
<td>CategoricalIndex.remove_unused_categories(...)</td>
<td>Removes categories which are not used.</td>
</tr>
<tr>
<td>CategoricalIndex.set_categories(*args, **kwargs)</td>
<td>Sets the categories to the specified new_categories.</td>
</tr>
<tr>
<td>CategoricalIndex.as_ordered(*args, **kwargs)</td>
<td>Sets the Categorical to be ordered</td>
</tr>
<tr>
<td>CategoricalIndex.as_unordered(*args, **kwargs)</td>
<td>Sets the Categorical to be unordered</td>
</tr>
</tbody>
</table>

pandas.CategoricalIndex.codes

CategoricalIndex.codes
pandas.CategoricalIndex.categories

CategoricalIndex.categories

pandas.CategoricalIndex.ordered

CategoricalIndex.ordered

pandas.CategoricalIndex.rename_categories

CategoricalIndex.rename_categories(*args, **kwargs)

   Renames categories.
   The new categories 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.

   Parameters
   new_categories : Index-like
       The renamed categories.
   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 with renamed categories added or None if inplace.

   Raises
ValueError

   If the new categories 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

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

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

pandas.CategoricalIndex.remove_categories

CategoricalIndex.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
pandas.CategoricalIndex.remove_unused_categories

`CategoricalIndex.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

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
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

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

33.9 DatetimeIndex

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.

33.9.1 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)

Attempt to infer fall dst-transition hours based on order

name : object

Name to be stored in the index

Attributes

T
asi8
asobject
base
data
date
day
dayofweek
dayofyear
days_in_month
daysinmonth
dtype
flags
freq
defreqstr
has_duplicates
hour
inferred_type
is_all_dates
is_monotonic
is_monotonic_decreasing
is_monotonic_increasing
is_month_end
is_month_start
is_quarter_end
is_quarter_start
is_year_end
is_year_start
itemsize
microsecond
millisecond

return the transpose, which is by definition self

return the base object if the memory of the underlying data is shared

return the data pointer of the underlying data

Returns numpy array of datetime.date.

The days of the datetime

The day of the week with Monday=0, Sunday=6

The ordinal day of the year

The number of days in the month

The number of days in the month

get/set the frequency of the Index

return the frequency object as a string if its set, otherwise None

The hours of the datetime

alias for is_monotonic_increasing (deprecated)

return if the index is monotonic decreasing (only equal or

return if the index is monotonic increasing (only equal or

Logical indicating if last day of month (defined by frequency)

Logical indicating if first day of month (defined by frequency)

Logical indicating if last day of quarter (defined by frequency)

Logical indicating if first day of quarter (defined by frequency)

Logical indicating if last day of year (defined by frequency)

Logical indicating if first day of year (defined by frequency)

return the size of the dtype of the item of the underlying data

The microseconds of the datetime

The milliseconds of the datetime

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minute</td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td>month</td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td>names</td>
<td></td>
</tr>
<tr>
<td>nanosecond</td>
<td>The nanoseconds of the datetime</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, by definition 1</td>
</tr>
<tr>
<td>levels</td>
<td></td>
</tr>
<tr>
<td>quarter</td>
<td>The quarter of the date</td>
</tr>
<tr>
<td>second</td>
<td>The seconds of the datetime</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>time</td>
<td>Returns numpy array of datetime.time.</td>
</tr>
<tr>
<td>tzinfo</td>
<td>Alias for tz attribute</td>
</tr>
<tr>
<td>values</td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td>week</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>weekday</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>weekofyear</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>year</td>
<td>The year of the datetime</td>
</tr>
</tbody>
</table>

**pandas.DatetimeIndex.T**

```
DatetimeIndex.T
```

return the transpose, which is by definition self

**pandas.DatetimeIndex.asi8**

```
DatetimeIndex.asi8
```

**pandas.DatetimeIndex.asobject**

```
DatetimeIndex.asobject
```

**pandas.DatetimeIndex.base**

```
DatetimeIndex.base
```

return the base object if the memory of the underlying data is shared

**pandas.DatetimeIndex.data**

```
DatetimeIndex.data
```

return the data pointer of the underlying data

**pandas.DatetimeIndex.date**

```
DatetimeIndex.date
```

Returns numpy array of datetime.date. The date part of the Timestamps.
pandas.DatetimeIndex.day

DatetimeIndex.day
The days of the datetime

pandas.DatetimeIndex.dayofweek

DatetimeIndex.dayofweek
The day of the week with Monday=0, Sunday=6

pandas.DatetimeIndex.dayofyear

DatetimeIndex.dayofyear
The ordinal day of the year

pandas.DatetimeIndex.days_in_month

DatetimeIndex.days_in_month
The number of days in the month
New in version 0.16.0.

pandas.DatetimeIndex.daysinmonth

DatetimeIndex.daysinmonth
The number of days in the month
New in version 0.16.0.

pandas.DatetimeIndex.dtype

DatetimeIndex.dtype

pandas.DatetimeIndex.flags

DatetimeIndex.flags

pandas.DatetimeIndex.freq

DatetimeIndex.freq
get/set the frequency of the Index

pandas.DatetimeIndex.freqstr

DatetimeIndex.freqstr
return the frequency object as a string if its set, otherwise None
pandas.DatetimeIndex.has_duplicates

DatetimeIndex.has_duplicates

pandas.DatetimeIndex.hour

DatetimeIndex.hour
The hours of the datetime

pandas.DatetimeIndex.inferred_type

DatetimeIndex.inferred_type

pandas.DatetimeIndex.is_all_dates

DatetimeIndex.is_all_dates

pandas.DatetimeIndex.is_monotonic

DatetimeIndex.is_monotonic
alias for is_monotonic_increasing (deprecated)

pandas.DatetimeIndex.is_monotonic_decreasing

DatetimeIndex.is_monotonic_decreasing
return if the index is monotonic decreasing (only equal or decreasing) values.

pandas.DatetimeIndex.is_monotonic_increasing

DatetimeIndex.is_monotonic_increasing
return if the index is monotonic increasing (only equal or increasing) values.

pandas.DatetimeIndex.is_month_end

DatetimeIndex.is_month_end
Logical indicating if last day of month (defined by frequency)

pandas.DatetimeIndex.is_month_start

DatetimeIndex.is_month_start
Logical indicating if first day of month (defined by frequency)

pandas.DatetimeIndex.is_quarter_end

DatetimeIndex.is_quarter_end
Logical indicating if last day of quarter (defined by frequency)
pandas.DatetimeIndex.is_quarter_start

DatetimeIndex.is_quarter_start
Logical indicating if first day of quarter (defined by frequency)

pandas.DatetimeIndex.is_year_end

DatetimeIndex.is_year_end
Logical indicating if last day of year (defined by frequency)

pandas.DatetimeIndex.is_year_start

DatetimeIndex.is_year_start
Logical indicating if first day of year (defined by frequency)

pandas.DatetimeIndex.itemsize

DatetimeIndex.itemsize
return the size of the dtype of the item of the underlying data

pandas.DatetimeIndex.microsecond

DatetimeIndex.microsecond
The microseconds of the datetime

pandas.DatetimeIndex.millisecond

DatetimeIndex.millisecond
The milliseconds of the datetime

pandas.DatetimeIndex.minute

DatetimeIndex.minute
The minutes of the datetime

pandas.DatetimeIndex.month

DatetimeIndex.month
The month as January=1, December=12

pandas.DatetimeIndex.names

DatetimeIndex.names

pandas.DatetimeIndex.nanosecond

DatetimeIndex.nanosecond
The nanoseconds of the datetime
pandas.DatetimeIndex.nbytes

DatetimeIndex.nbytes
return the number of bytes in the underlying data

pandas.DatetimeIndex.ndim

DatetimeIndex.ndim
return the number of dimensions of the underlying data, by definition 1

pandas.DatetimeIndex.nlevels

DatetimeIndex.nlevels

pandas.DatetimeIndex.quarter

DatetimeIndex.quarter
The quarter of the date

pandas.DatetimeIndex.second

DatetimeIndex.second
The seconds of the datetime

pandas.DatetimeIndex.shape

DatetimeIndex.shape
return a tuple of the shape of the underlying data

pandas.DatetimeIndex.size

DatetimeIndex.size
return the number of elements in the underlying data

pandas.DatetimeIndex.strides

DatetimeIndex.strides
return the strides of the underlying data

pandas.DatetimeIndex.time

DatetimeIndex.time
Returns numpy array of datetime.time. The time part of the Timestamps.

pandas.DatetimeIndex.tzinfo

DatetimeIndex.tzinfo
Alias for tz attribute
pandas.DatetimeIndex.values

DatetimeIndex.values
return the underlying data as an ndarray

pandas.DatetimeIndex.week

DatetimeIndex.week
The week ordinal of the year

pandas.DatetimeIndex.weekday

DatetimeIndex.weekday
The day of the week with Monday=0, Sunday=6

pandas.DatetimeIndex.weekofyear

DatetimeIndex.weekofyear
The week ordinal of the year

pandas.DatetimeIndex.year

DatetimeIndex.year
The year of the datetime

Methods

all([other])
any([other])
append(other)
argmax([axis])
argmin([axis])
argsort(*args, **kwargs)
asof(label)
asof_locs(where, mask)
astype(dtype)
copy([names, name, dtype, deep])
delete(loc)
diff(*args, **kwargs)
difference(other)

Append a collection of Index options together
return a ndarray of the maximum argument indexer
return a ndarray of the minimum argument indexer
return an ndarray indexer of the underlying data
For a sorted index, return the most recent label up to and including the passed
where : array of timestamps
Make a copy of this object.
Make a new DatetimeIndex with passed location(s) deleted.
Compute sorted set difference of two Index objects
Table 33.102 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop(labels[, errors])</td>
<td>Make new Index with passed list of labels deleted</td>
</tr>
<tr>
<td>drop_duplicates([take_last])</td>
<td>Return Index with duplicate values removed</td>
</tr>
<tr>
<td>duplicated([take_last])</td>
<td>Return boolean np.array denoting duplicate values</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two Index objects contain the same elements.</td>
</tr>
<tr>
<td>factorize((sort, na_sentinel))</td>
<td>Encode the object as an enumerated type or categorical variable</td>
</tr>
<tr>
<td>format([name, formatter])</td>
<td>Render a string representation of the Index</td>
</tr>
<tr>
<td>get_duplicates()</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>get_indexer(target[, method, limit])</td>
<td>guaranteed return of an indexer even when non-unique</td>
</tr>
<tr>
<td>get_indexer_for(target, **kwargs)</td>
<td>return an indexer suitable for taking from a non unique index</td>
</tr>
<tr>
<td>get_level_values(level)</td>
<td>Return vector of label values for requested level, equal to the length</td>
</tr>
<tr>
<td>get_loc(key[, method])</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>get_values()</td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td>get_duplicates()</td>
<td>Select values at particular time of day (e.g., 9:00-9:30AM)</td>
</tr>
<tr>
<td>get_indexer(target[, method, limit])</td>
<td>Make new Index inserting new item at location</td>
</tr>
<tr>
<td>get_indexer_for(target, **kwargs)</td>
<td>Specialized intersection for DatetimeIndex objects.</td>
</tr>
<tr>
<td>get_indexer_non_unique(target)</td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td>get_values()</td>
<td>Compute boolean array of whether each index value is found in the</td>
</tr>
<tr>
<td>get_value(series, key)</td>
<td>return the first element of the underlying data as a python scalar</td>
</tr>
<tr>
<td>get_values()</td>
<td>See Index.join</td>
</tr>
<tr>
<td>groupby(f)</td>
<td>return the maximum value of the Index</td>
</tr>
<tr>
<td>holds_integer()</td>
<td>return the minimum value of the Index</td>
</tr>
<tr>
<td>identical(other)</td>
<td>Return DatetimeIndex with times to midnight.</td>
</tr>
<tr>
<td>indexer_at_time(time[, asof])</td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td>indexer_between_time(start_time, end_time[, ...])</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>insert(loc, item)</td>
<td>return a new Index of the values set with the mask</td>
</tr>
<tr>
<td>intersection(other)</td>
<td>return an ndarray of the flattened values of the underlying data</td>
</tr>
<tr>
<td>is_(other)</td>
<td>Create index with target’s values (move/add/delete values as necessary)</td>
</tr>
<tr>
<td>is_boolean()</td>
<td>Set new names on index</td>
</tr>
<tr>
<td>is_categorical()</td>
<td>Analogous to ndarray.repeat</td>
</tr>
<tr>
<td>is_floating()</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>is_integer()</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>is_lexsorted_for_tuple(tup)</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>is_mixed()</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>is_numeric()</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>is_object()</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>is_type_compatible(typ)</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>isin(values)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>item()</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>join(other[, how, level, return_indexers])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>map(f)</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>max([axis])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>min([axis])</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>normalize()</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>unique([dropna])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>order([return_indexer, ascending])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>putmask(mask, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>ravel(order)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>reindex(target[, method, level, limit])</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>rename(name[, inplace])</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>repeat(repeats[, axis])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>searchsorted(key[, side])</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>set_names(names[, level, inplace])</td>
<td>Return indexer for specified label slice.</td>
</tr>
<tr>
<td>set_value(arr, key, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>shift(n[, freq])</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>slice_indexer([start, end, step, kind])</td>
<td>Return indexer for specified label slice.</td>
</tr>
</tbody>
</table>
### Table 33.102 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>Alias of <code>StringMethods</code></td>
</tr>
<tr>
<td><code>summary([name])</code></td>
<td>Return a summarized representation</td>
</tr>
<tr>
<td><code>sym_diff(other[, result_name])</code></td>
<td>Compute the sorted symmetric difference of two Index objects.</td>
</tr>
<tr>
<td><code>take(indices[, axis])</code></td>
<td>Analogous to <code>ndarray.take</code></td>
</tr>
<tr>
<td><code>to_datetime([dayfirst])</code></td>
<td>Convert DatetimeIndex to Float64Index of Julian Dates.</td>
</tr>
<tr>
<td><code>to_julian_date()</code></td>
<td>Slice and dice then format</td>
</tr>
<tr>
<td><code>to_period([freq])</code></td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td><code>to_pydatetime()</code></td>
<td>Create a Series with both index and values equal to the index keys</td>
</tr>
<tr>
<td><code>to_series([keep_tz])</code></td>
<td>Return a list of the underlying data</td>
</tr>
<tr>
<td><code>transpose()</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 pytz)</td>
</tr>
<tr>
<td><code>tz_localize(*args, **kwargs)</code></td>
<td>Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil),</td>
</tr>
<tr>
<td><code>union(other)</code></td>
<td>Specialized union for DatetimeIndex objects.</td>
</tr>
<tr>
<td><code>union_many(others)</code></td>
<td>A bit of a hack to accelerate unioning a collection of indexes</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Index.unique with handling for DatetimeIndex/PeriodIndex metadata</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></td>
</tr>
</tbody>
</table>

#### pandas.DatetimeIndex.all

`DatetimeIndex.all(other=None)`

#### pandas.DatetimeIndex.any

`DatetimeIndex.any(other=None)`

#### pandas.DatetimeIndex.append

`DatetimeIndex.append(other)`

Append a collection of Index options together

- **Parameters**
  - `other` : Index or list/tuple of indices

- **Returns**
  - `appended` : Index

#### pandas.DatetimeIndex.argmax

`DatetimeIndex.argmax(axis=None)`

Return a `ndarray` of the maximum argument indexer

- **See also:**
  - `numpy.ndarray.argmax`
pandas.DatetimeIndex.argmin

DatetimeIndex.argmin(axis=None)
    return a ndarray of the minimum argument indexer

See also:
    numpy.ndarray.argmin

pandas.DatetimeIndex.argsort

DatetimeIndex.argsort(*args, **kwargs)
    return an ndarray indexer of the underlying data

See also:
    numpy.ndarray.argsort

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'

pandas.DatetimeIndex.asof_locs

DatetimeIndex.asof_locs(where, mask)
    where : array of timestamps
    mask : array of booleans where data is not NA

pandas.DatetimeIndex.astype

DatetimeIndex.astype(dtype)

pandas.DatetimeIndex.copy

DatetimeIndex.copy(names=None, name=None, dtype=None, deep=False)
    Make a copy of this object. Name and dtype sets those attributes on the new object.

    Parameters
    name : string, optional
    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.
pandas.DatetimeIndex.delete

DatetimeIndex.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

pandas.DatetimeIndex.diff

DatetimeIndex.diff(*args, **kwargs)

pandas.DatetimeIndex.difference

DatetimeIndex.difference(other)
Compute sorted set difference of two Index objects

Parameters  other : Index or array-like

Returns  diff : Index

Notes

One can do either of these and achieve the same result

```python
>>> index.difference(index2)
```
pandas.DatetimeIndex.duplicated

DatetimeIndex.duplicated (take_last=False)
Return boolean np.array denoting duplicate values

Parameters take_last : boolean, default False
Take the last observed index in a group. Default first

Returns duplicated : np.array

pandas.DatetimeIndex.equals

DatetimeIndex.equals (other)
Determines if two Index objects contain the same elements.

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

pandas.DatetimeIndex.format

DatetimeIndex.format (name=False, formatter=None, **kwargs)
Render a string representation of the Index

pandas.DatetimeIndex.get_duplicates

DatetimeIndex.get_duplicates ()

pandas.DatetimeIndex.get_indexer

DatetimeIndex.get_indexer (target, method=None, limit=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

Maximum number of consecutive labels in `target` to match for inexact matches.

**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)
```

**pandas.DatetimeIndex.get_indexer_for**

`DatetimeIndex.get_indexer_for(target, **kwargs)`

Guaranteed return of an indexer even when non-unique.

**pandas.DatetimeIndex.get_indexer_non_unique**

`DatetimeIndex.get_indexer_non_unique(target)`

Return an indexer suitable for taking from a non unique index return the labels in the same order as the target, and return a missing indexer into the target (missing are marked as -1 in the indexer); target must be an iterable.

**pandas.DatetimeIndex.get_level_values**

`DatetimeIndex.get_level_values(level)`

Return vector of label values for requested level, equal to the length of the index.

**Parameters** **level**: int

**Returns** **values**: ndarray

**pandas.DatetimeIndex.get_loc**

`DatetimeIndex.get_loc(key, method=None)`

Get integer location for requested label.

**Returns** **loc**: int

**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.
Pandas: powerful Python data analysis toolkit, Release 0.16.2

Parameters

- **label**: object
- **side**: {'left', 'right'}
- **kind**: string / None, the type of indexer

**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

**pandas.DatetimeIndex.get_value_maybe_box**

```
DatetimeIndex.get_value_maybe_box(series, key)
```

**pandas.DatetimeIndex.get_values**

```
DatetimeIndex.get_values()
```

return the underlying data as an ndarray

**pandas.DatetimeIndex.groupby**

```
DatetimeIndex.groupby(f)
```

**pandas.DatetimeIndex.holds_integer**

```
DatetimeIndex.holds_integer()
```

**pandas.DatetimeIndex.identical**

```
DatetimeIndex.identical(other)
```

Similar to equals, but check that other comparable attributes are also equal

**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

**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)
Parameters start_time : datetime.time or string
end_time : datetime.time or string
include_start : boolean, default True
include_end : boolean, default True
tz : string or pytz.timezone or dateutil.tz.tzfile, default None

Returns values_between_time : TimeSeries

pandas.DatetimeIndex.insert

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

pandas.DatetimeIndex.intersection

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

pandas.DatetimeIndex.is

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

pandas.DatetimeIndex.is_boolean

DatetimeIndex.is_boolean()

pandas.DatetimeIndex.is_categorical

DatetimeIndex.is_categorical()
pandas.DatetimeIndex.is_floating

DatetimeIndex.is_floating()

pandas.DatetimeIndex.is_integer

DatetimeIndex.is_integer()

pandas.DatetimeIndex.is_lexsorted_for_tuple

DatetimeIndex.is_lexsorted_for_tuple(tup)

pandas.DatetimeIndex.is_mixed

DatetimeIndex.is_mixed()

pandas.DatetimeIndex.is_numeric

DatetimeIndex.is_numeric()

pandas.DatetimeIndex.is_object

DatetimeIndex.is_object()

pandas.DatetimeIndex.is_type_compatible

DatetimeIndex.is_type_compatible(typ)

pandas.DatetimeIndex.isin

DatetimeIndex.isin(values)

Parameters values : set or sequence of values

Returns is_contained : ndarray (boolean dtype)

pandas.DatetimeIndex.item

DatetimeIndex.item()

return the first element of the underlying data as a python scalar

pandas.DatetimeIndex.join

DatetimeIndex.join(other, how='left', level=None, return_indexers=False)

See Index.join
**pandas.DatetimeIndex.map**

```python
DatetimeIndex.map(f)
```

**pandas.DatetimeIndex.max**

```python
DatetimeIndex.max(axis=None)
```

return the maximum value of the Index

**See also:**

```python
numpy.ndarray.max
```

**pandas.DatetimeIndex.min**

```python
DatetimeIndex.min(axis=None)
```

return the minimum value of the Index

**See also:**

```python
numpy.ndarray.min
```

**pandas.DatetimeIndex.normalize**

```python
DatetimeIndex.normalize()
```

Return DatetimeIndex with times to midnight. Length is unaltered

**Returns** normalized : DatetimeIndex

**pandas.DatetimeIndex.nunique**

```python
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

**pandas.DatetimeIndex.order**

```python
DatetimeIndex.order(return_indexer=False, ascending=True)
```

Return sorted copy of Index

**pandas.DatetimeIndex.putmask**

```python
DatetimeIndex.putmask(mask, value)
```

return a new Index of the values set with the mask

**See also:**

```python
numpy.ndarray.putmask
```
**pandas.DatetimeIndex.ravel**

`DatetimeIndex.ravel(order='C')`  
return an ndarray of the flattened values of the underlying data  

See also:  
`numpy.ndarray.ravel`

**pandas.DatetimeIndex.reindex**

`DatetimeIndex.reindex(target, method=None, level=None, limit=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

**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]

**pandas.DatetimeIndex.repeat**

`DatetimeIndex.repeat(repeats, axis=None)`  
Analogous to ndarray.repeat

**pandas.DatetimeIndex.searchsorted**

`DatetimeIndex.searchsorted(key, side='left')`

**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 or 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, 1, 0, 1], [0, 0, 1, 1]],
           names=[u'baz', u'quz'])
>>> idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'baz', u'bar'])
```

**pandas.DatetimeIndex.set_value**

DatetimeIndex.set_value(arr, key, value)
Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

**pandas.DatetimeIndex.shift**

DatetimeIndex.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

**pandas.DatetimeIndex.slice_indexer**

DatetimeIndex.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.
**pandas.DatetimeIndex.slice_locs**

```
datetimeindex.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` : string, defaults None

**Returns**
- `start, end` : int

**pandas.DatetimeIndex.snap**

```
datetimeindex.snap(freq='S')
```

Snap time stamps to nearest occurring frequency

**pandas.DatetimeIndex.sort**

```
datetimeindex.sort(*args, **kwargs)
```

**pandas.DatetimeIndex.summary**

```
datetimeindex.summary(name=None)
```

return a summarized representation

**pandas.DatetimeIndex.sym_diff**

```
datetimeindex.sym_diff(other, result_name=None)
```

Compute the sorted symmetric difference of two Index objects.

**Parameters**
- `other` : Index or array-like
- `result_name` : str

**Returns**
- `sym_diff` : Index

**Notes**

`sym_diff` contains elements that appear in either idx1 or idx2 but not both. Equivalent to the Index created by \((\text{idx1} - \text{idx2}) + (\text{idx2} - \text{idx1})\) with duplicates dropped.

The sorting of a result containing NaN values is not guaranteed across Python versions. See GitHub issue #6444.
Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.sym_diff(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

### pandas.DatetimeIndex.take

```
DatetimeIndex.take(indices, axis=0)
```

Analogous to ndarray.take

### pandas.DatetimeIndex.to_datetime

```
DatetimeIndex.to_datetime(dayfirst=False)
```

### 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

### pandas.DatetimeIndex.to_native_types

```
DatetimeIndex.to_native_types(slicer=None, **kwargs)
```

Slice and dice then format

### pandas.DatetimeIndex.to_period

```
DatetimeIndex.to_period(freq=None)
```

Cast to PeriodIndex at a particular frequency

### pandas.DatetimeIndex.to_pydatetime

```
DatetimeIndex.to_pydatetime()
```

Return DatetimeIndex as object ndarray of datetime.datetime objects

Returns : 

datetimes : ndarray

### 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 object 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

**pandas.DatetimeIndex.tolist**

`DatetimeIndex.tolist()`

return a list of the underlying data

**pandas.DatetimeIndex.transpose**

`DatetimeIndex.transpose()`

return the transpose, which is by definition self

**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.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

**Raises** `TypeError`

If DatetimeIndex is tz-naive.

**pandas.DatetimeIndex.tz_localize**

`DatetimeIndex.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

**infer_dst** : boolean, default False (DEPRECATED)

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.

**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

**pandas.DatetimeIndex.union_many**

`DatetimeIndex.union_many(others)`

A bit of a hack to accelerate unioning a collection of indexes

**pandas.DatetimeIndex.unique**

`DatetimeIndex.unique()`

Index.unique with handling for DatetimeIndex/PeriodIndex metadata

**Returns** result : DatetimeIndex or PeriodIndex

**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

```python
pandas.DatetimeIndex.view
```

`DatetimeIndex.view(cls=None)`

### 33.9.2 Time/Date Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DatetimeIndex.year</code></td>
<td>The year of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.month</code></td>
<td>The month as January=1, December=12</td>
</tr>
<tr>
<td><code>DatetimeIndex.day</code></td>
<td>The days of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.hour</code></td>
<td>The hours of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.minute</code></td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.second</code></td>
<td>The seconds of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.microsecond</code></td>
<td>The microseconds of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.nanosecond</code></td>
<td>The nanoseconds of the datetime</td>
</tr>
<tr>
<td><code>DatetimeIndex.date</code></td>
<td>Returns numpy array of datetime.date.</td>
</tr>
<tr>
<td><code>DatetimeIndex.time</code></td>
<td>Returns numpy array of datetime.time.</td>
</tr>
<tr>
<td><code>DatetimeIndex.dayofyear</code></td>
<td>The ordinal day of the year</td>
</tr>
<tr>
<td><code>DatetimeIndex.weekofyear</code></td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td><code>DatetimeIndex.week</code></td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td><code>DatetimeIndex.dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td><code>DatetimeIndex.weekday</code></td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td><code>DatetimeIndex.quarter</code></td>
<td>The quarter of the date</td>
</tr>
<tr>
<td><code>DatetimeIndex.tz</code></td>
<td></td>
</tr>
<tr>
<td><code>DatetimeIndex.freq</code></td>
<td>get/set the frequency of the Index</td>
</tr>
<tr>
<td><code>DatetimeIndex.freqstr</code></td>
<td>return the frequency object as a string if its set, otherwise None</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_month_start</code></td>
<td>Logical indicating if first day of month (defined by frequency)</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_month_end</code></td>
<td>Logical indicating if last day of month (defined by frequency)</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_quarter_start</code></td>
<td>Logical indicating if first day of quarter (defined by frequency)</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_quarter_end</code></td>
<td>Logical indicating if last day of quarter (defined by frequency)</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_year_start</code></td>
<td>Logical indicating if first day of year (defined by frequency)</td>
</tr>
<tr>
<td><code>DatetimeIndex.is_year_end</code></td>
<td>Logical indicating if last day of year (defined by frequency)</td>
</tr>
</tbody>
</table>

```python
pandas.DatetimeIndex.year
```

`DatetimeIndex.year`  
The year of the datetime
pandas.DatetimeIndex.month

DatetimeIndex.month
The month as January=1, December=12

pandas.DatetimeIndex.day

DatetimeIndex.day
The days of the datetime

pandas.DatetimeIndex.hour

DatetimeIndex.hour
The hours of the datetime

pandas.DatetimeIndex.minute

DatetimeIndex.minute
The minutes of the datetime

pandas.DatetimeIndex.second

DatetimeIndex.second
The seconds of the datetime

pandas.DatetimeIndex.microsecond

DatetimeIndex.microsecond
The microseconds of the datetime

pandas.DatetimeIndex.nanosecond

DatetimeIndex.nanosecond
The nanoseconds of the datetime

pandas.DatetimeIndex.date

DatetimeIndex.date
Returns numpy array of datetime.date. The date part of the Timestamps.

pandas.DatetimeIndex.time

DatetimeIndex.time
Returns numpy array of datetime.time. The time part of the Timestamps.
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.DatetimeIndex.dayofyear

DatetimeIndex.dayofyear
The ordinal day of the year

pandas.DatetimeIndex.weekofyear

DatetimeIndex.weekofyear
The week ordinal of the year

pandas.DatetimeIndex.week

DatetimeIndex.week
The week ordinal of the year

pandas.DatetimeIndex.dayofweek

DatetimeIndex.dayofweek
The day of the week with Monday=0, Sunday=6

pandas.DatetimeIndex.weekday

DatetimeIndex.weekday
The day of the week with Monday=0, Sunday=6

pandas.DatetimeIndex.quarter

DatetimeIndex.quarter
The quarter of the date

pandas.DatetimeIndex.tz

DatetimeIndex.tz = None

pandas.DatetimeIndex.freq

DatetimeIndex.freq
get/set the frequency of the Index

pandas.DatetimeIndex.freqstr

DatetimeIndex.freqstr
return the frequency object as a string if its set, otherwise None

pandas.DatetimeIndex.is_month_start

DatetimeIndex.is_month_start
Logical indicating if first day of month (defined by frequency)
pandas.DatetimeIndex.is_month_end

DatetimeIndex.is_month_end
Logical indicating if last day of month (defined by frequency)

pandas.DatetimeIndex.is_quarter_start

DatetimeIndex.is_quarter_start
Logical indicating if first day of quarter (defined by frequency)

pandas.DatetimeIndex.is_quarter_end

DatetimeIndex.is_quarter_end
Logical indicating if last day of quarter (defined by frequency)

pandas.DatetimeIndex.is_year_start

DatetimeIndex.is_year_start
Logical indicating if first day of year (defined by frequency)

pandas.DatetimeIndex.is_year_end

DatetimeIndex.is_year_end
Logical indicating if last day of year (defined by frequency)

pandas.DatetimeIndex.inferred_freq

DatetimeIndex.inferred_freq = None

33.9.3 Selecting

| DatetimeIndex.indexer_at_time(time[, asof]) | Select values at particular time of day (e.g., 9:30AM) |
| DatetimeIndex.indexer_between_time(...) | Select values between particular times of day (e.g., 9:00-9:30AM) |

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 : datatype.time or string
Returns values_at_time : TimeSeries

pandas.DatetimeIndex.indexer_between_time

DatetimIndex.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)
**Parameters**

- **start_time**: datetime.time or string
- **end_time**: datetime.time or string
- **include_start**: boolean, default True
- **include_end**: boolean, default True
- **tz**: string or pytz.timezone or dateutil.tz.tzfile, default None

**Returns**

- **values_between_time**: TimeSeries

### 33.9.4 Time-specific operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.normalize()</td>
<td>Return DatetimeIndex with times to midnight.</td>
</tr>
<tr>
<td>DatetimeIndex.snap(freq)</td>
<td>Snap time stamps to nearest occurring frequency.</td>
</tr>
<tr>
<td>DatetimeIndex.tz_convert(tz)</td>
<td>Convert tz-aware DatetimeIndex from one time zone to another (using pytz).</td>
</tr>
<tr>
<td>DatetimeIndex.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil).</td>
</tr>
</tbody>
</table>

#### pandas.DatetimeIndex.normalize

```
DatetimeIndex.normalize()
```

Return DatetimeIndex with times to midnight. Length is unaltered

**Returns**

- **normalized**: DatetimeIndex

#### pandas.DatetimeIndex.snap

```
DatetimeIndex.snap(freq='S')
```

Snap time stamps to nearest occurring frequency

#### 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.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

**Raises**

- TypeError
  
  If DatetimeIndex is tz-naive.

#### pandas.DatetimeIndex.tz_localize

```
DatetimeIndex.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

**infer_dst** : boolean, default False (DEPRECATED)
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.

### 33.9.5 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>Return DatetimeIndex as object ndarray of datetime.datetime objects</td>
</tr>
<tr>
<td>DatetimeIndex.to_pydatetime()</td>
<td>Create a Series with both index and values equal to the index keys</td>
</tr>
</tbody>
</table>

**pandas.DatetimeIndex.to_datetime**

 DatetimeIndex.to_datetime *(dayfirst=False)*

**pandas.DatetimeIndex.to_period**

 DatetimeIndex.to_period *(freq=None)*

 Cast to PeriodIndex at a particular frequency

**pandas.DatetimeIndex.to_pydatetime**

 DatetimeIndex.to_pydatetime *

 Return DatetimeIndex as object ndarray of datetime.datetime objects

 **Returns** datetimes : ndarray

**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 object 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

33.10 TimedeltaIndex

TimedeltaIndex Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects

33.10.1 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
### 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>asi8</code></td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td><code>asobject</code></td>
<td></td>
</tr>
<tr>
<td><code>base</code></td>
<td>return the base object if the memory of the underlying data is shared</td>
</tr>
<tr>
<td><code>components</code></td>
<td>Return a dataframe of the components (days, hours, minutes, seconds, 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>return the frequency object as a string if its set, otherwise None</td>
</tr>
<tr>
<td><code>flags</code></td>
<td></td>
</tr>
<tr>
<td><code>freqstr</code></td>
<td>alias for <code>is_monotonic_increasing</code> (deprecated)</td>
</tr>
<tr>
<td><code>has_duplicates</code></td>
<td>return if the index is monotonic decreasing (only equal or</td>
</tr>
<tr>
<td><code>inferred_type</code></td>
<td>return if the index is monotonic increasing (only equal or</td>
</tr>
<tr>
<td><code>is_all_dates</code></td>
<td>return the size of the dtype of the item of the underlying data</td>
</tr>
<tr>
<td><code>is_monotonic</code></td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td><code>is_monotonic_decreasing</code></td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</td>
</tr>
<tr>
<td><code>is_monotonic_increasing</code></td>
<td>return the number of bytes in the underlying data</td>
</tr>
<tr>
<td><code>itemsize</code></td>
<td>return the number of dimensions of the underlying data, by definition 1</td>
</tr>
<tr>
<td><code>microseconds</code></td>
<td>return the number of elements in the underlying data</td>
</tr>
<tr>
<td><code>names</code></td>
<td>return the strides of the underlying data</td>
</tr>
<tr>
<td><code>nanoseconds</code></td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td><code>nbytes</code></td>
<td>return the underlying data as an ndarray</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td></td>
</tr>
<tr>
<td><code>nlevels</code></td>
<td></td>
</tr>
<tr>
<td><code>seconds</code></td>
<td></td>
</tr>
<tr>
<td><code>shape</code></td>
<td></td>
</tr>
<tr>
<td><code>size</code></td>
<td></td>
</tr>
<tr>
<td><code>strides</code></td>
<td></td>
</tr>
<tr>
<td><code>values</code></td>
<td></td>
</tr>
</tbody>
</table>

#### pandas.TimedeltaIndex.T

T

return the transpose, which is by definition self

#### pandas.TimedeltaIndex.asi8

T

return the base object if the memory of the underlying data is shared

#### pandas.TimedeltaIndex.asobject

T

return the base object if the memory of the underlying data is shared
pandas.TimedeltaIndex.components

TimedeltaIndex.components
Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns
a DataFrame

pandas.TimedeltaIndex.data

TimedeltaIndex.data
return the data pointer of the underlying data

pandas.TimedeltaIndex.days

TimedeltaIndex.days
Number of days for each element.

pandas.TimedeltaIndex.dtype

TimedeltaIndex.dtype

pandas.TimedeltaIndex.flags

TimedeltaIndex.flags

pandas.TimedeltaIndex.freqstr

TimedeltaIndex.freqstr
return the frequency object as a string if its set, otherwise None

pandas.TimedeltaIndex.has_duplicates

TimedeltaIndex.has_duplicates

pandas.TimedeltaIndex.inferred_type

TimedeltaIndex.inferred_type

pandas.TimedeltaIndex.is_all_dates

TimedeltaIndex.is_all_dates

pandas.TimedeltaIndex.is_monotonic

TimedeltaIndex.is_monotonic
alias for is_monotonic_increasing (deprecated)
pandas.TimedeltaIndex.is_monotonic_decreasing

TimedeltaIndex.is_monotonic_decreasing
return if the index is monotonic decreasing (only equal or decreasing) values.

pandas.TimedeltaIndex.is_monotonic_increasing

TimedeltaIndex.is_monotonic_increasing
return if the index is monotonic increasing (only equal or increasing) values.

pandas.TimedeltaIndex.itemsize

TimedeltaIndex.itemsize
return the size of the dtype of the item of the underlying data

pandas.TimedeltaIndex.microseconds

TimedeltaIndex.microseconds
Number of microseconds (>= 0 and less than 1 second) for each element.

pandas.TimedeltaIndex.names

TimedeltaIndex.names

pandas.TimedeltaIndex.nanoseconds

TimedeltaIndex.nanoseconds
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

pandas.TimedeltaIndex.nbytes

TimedeltaIndex.nbytes
return the number of bytes in the underlying data

pandas.TimedeltaIndex.ndim

TimedeltaIndex.ndim
return the number of dimensions of the underlying data, by definition 1

pandas.TimedeltaIndex.nlevels

TimedeltaIndex.nlevels

pandas.TimedeltaIndex.seconds

TimedeltaIndex.seconds
Number of seconds (>= 0 and less than 1 day) for each element.
pandas.TimedeltaIndex.shape

TimedeltaIndex.shape
return a tuple of the shape of the underlying data

pandas.TimedeltaIndex.size

TimedeltaIndex.size
return the number of elements in the underlying data

pandas.TimedeltaIndex.strides

TimedeltaIndex.strides
return the strides of the underlying data

pandas.TimedeltaIndex.values

TimedeltaIndex.values
return the underlying data as an ndarray

freq
hasnans
inferred_freq
is_unique
name
resolution

Methods

all([other])
any([other])
append(other)
argmax([axis])
argmin([axis])
argsort(*args, **kwargs)
asof(label)
asof_locs(where, mask)
astype(dtype)
copy([names, name, dtype, deep])
delete(loc)
diff(*args, **kwargs)
difference(other)
drop(labels[, errors])
drop_duplicates([take_last])
duplicated([take_last])
equals(other)
factorize([sort, na_sentinel])
format([name, formatter])
get_duplicates()
get_indexer(target[, method, limit])

Append a collection of Index options together
return a ndarray of the maximum argument indexer
return a ndarray of the minimum argument indexer
return an ndarray indexer of the underlying data
For a sorted index, return the most recent label up to and including the passed label.
where : array of timestamps
Make a copy of this object.
Make a new DatetimeIndex with passed location(s) deleted.
Compute sorted set difference of two Index objects
Make new Index with passed list of labels deleted
Return Index with duplicate values removed
Return boolean np.array denoting duplicate values
Determines if two Index objects contain the same elements.
Encode the object as an enumerated type or categorical variable
Render a string representation of the Index
Compute indexer and mask for new index given the current index.
Table 33.109 – continued from previous page

- `get_indexer_for` (target, **kwargs) guaranteed return of an indexer even when non-unique
- `get_indexer_non_unique` (target) return an indexer suitable for taking from a non-unique index
- `get_level_values` (level) return vector of label values for requested level, equal to the length
- `get_loc` (key[, method]) Get integer location for requested label
- `get_slice_bound` (label, side, kind) Calculate slice bound that corresponds to given label.
- `get_value` (series, key) Fast lookup of value from 1-dimensional ndarray.
- `get_value_maybe_box` (series, key) return the underlying data as an ndarray

- `get_values()` return the underlying data as an ndarray

- `groupby` (f) Similar to equals, but check that other comparable attributes are
- `holds_integer()` Make new Index inserting new item at location
- `identical` (other) Specialized intersection for TimedeltaIndex objects.
- `insert` (loc, item) More flexible, faster check like `is` but that works through views
- `intersection` (other) Compute boolean array of whether each index value is found in the
- `is__` (other) return the first element of the underlying data as a python scalar
- `isboolean()` See Index.join
- `iscategorical()` return the maximum value of the Index
- `isfloat()` return the minimum value of the Index
- `isinteger()` Return number of unique elements in the object.
- `islexsorted_for_tuple` (tup) Return sorted copy of Index
- `ismixed()` return a new Index of the values set with the mask
- `isnumeric()` return an ndarray of the flattened values of the underlying data
- `isobject()` Create index with target’s values (move/add/delete values as necessary)
- `is_type_compatible` (typ) Set new names on index.
- `isin` (values) Analogous to ndarray.repeat
- `item()` return a new Index on the flattened values of the underlying data
- `join` (other[, how, level, return_indexers]) Set new names on index.
- `map` (f) Analogous to ndarray.take
- `max` ([axis]) Fast lookup of value from 1-dimensional ndarray.
- `min` ([axis]) alias of StringMethods
- `nunique` ([dropna]) return a summarized representation
- `order` ([return_indexer, ascending]) Compute the sorted symmetric difference of two Index objects.
- `putmask` (mask, value) Analogous to ndarray.take
- `ravel` ([order]) For an Index containing strings or datetime.datetime objects, attempt
- `reindex` (target[, method, level, limit]) Specialized shift which produces a DatetimeIndex
- `rename` (name[, inplace]) For an ordered Index, compute the slice indexer for input labels and
- `repeat` (repeats[, axis]) Compute slice locations for input labels.
- `searchsorted` (key[, side]) alias of StringMethods
- `set_names` (names[, level, inplace]) return a summarized representation
- `set_value` (arr, key, value) Compute the sorted symmetric difference of two Index objects.
- `shift` (n[, freq]) Analogous to ndarray.take
- `slice_indexer` ([start, end, step, kind]) For an Index containing strings or datetime.datetime objects, attempt
- `slice_locs` ([start, end, step, kind]) slice and dice then format
- `sort` (*args, **kwargs) Return TimedeltaIndex as object ndarray of datetime.timedelta objects
- `str` Create a Series with both index and values equal to the index keys
- `summary` ([name]) return a list of the underlying data
- `sym_diff` (other[, result_name]) 33.10. TimedeltaIndex
- `take` (indices[, axis]) 1523
- `to_datetime` ([dayfirst])
Table 33.109 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transpose()</code></td>
<td>return the transpose, which is by definition self</td>
</tr>
<tr>
<td><code>union(other)</code></td>
<td>Specialized union for TimedeltaIndex objects.</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Index.unique with handling for DatetimeIndex/PeriodIndex metadata</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></td>
</tr>
</tbody>
</table>

```python
pandas.TimedeltaIndex.all

TimedeltaIndex.all(\texttt{other=None})
```

```python
pandas.TimedeltaIndex.any

TimedeltaIndex.any(\texttt{other=None})
```

```python
pandas.TimedeltaIndex.append

TimedeltaIndex.append(\texttt{other})

Append a collection of Index options together

**Parameters**
- \texttt{other}: Index or list/tuple of indices

**Returns**
- \texttt{appended}: Index
```

```python
pandas.TimedeltaIndex.argmax

TimedeltaIndex.argmax(\texttt{axis=None})

return a ndarray of the maximum argument indexer

**See also:**
- \texttt{numpy.ndarray.argmax}
```

```python
pandas.TimedeltaIndex.argmin

TimedeltaIndex.argmin(\texttt{axis=None})

return a ndarray of the minimum argument indexer

**See also:**
- \texttt{numpy.ndarray.argmin}
```

```python
pandas.TimedeltaIndex.argsort

TimedeltaIndex.argsort(*\texttt{args}, **\texttt{kwargs})

return an indexer of the underlying data

**See also:**
- \texttt{numpy.ndarray.argsort}
```
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'

pandas.TimedeltaIndex.asof_locs

TimedeltaIndex.asof_locs(where, mask)
where : array of timestamps mask : array of booleans where data is not NA

pandas.TimedeltaIndex.astype

TimedeltaIndex.astype(dtype)

pandas.TimedeltaIndex.copy

TimedeltaIndex.copy(names=None, name=None, dtype=None, deep=False)
Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters

name : string, optional
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.

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

pandas.TimedeltaIndex.diff

TimedeltaIndex.diff(*args, **kwargs)
**pandas.TimedeltaIndex.difference**

TimedeltaIndex.difference(other)
Compute sorted set difference of two Index objects

**Parameters** other : Index or array-like

**Returns** diff : Index

**Notes**

One can do either of these and achieve the same result

```python
>>> index.difference(index2)
```

**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

**pandas.TimedeltaIndex.drop_duplicates**

TimedeltaIndex.drop_duplicates(take_last=False)
Return Index with duplicate values removed

**Parameters** take_last : boolean, default False

Take the last observed index in a group. Default first

**Returns** deduplicated : Index

**pandas.TimedeltaIndex.duplicated**

TimedeltaIndex.duplicated(take_last=False)
Return boolean np.array denoting duplicate values

**Parameters** take_last : boolean, default False

Take the last observed index in a group. Default first

**Returns** duplicated : np.array

**pandas.TimedeltaIndex.equals**

TimedeltaIndex.equals(other)
Determines if two Index objects contain the same elements.
**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

**pandas.TimedeltaIndex.format**

TimedeltaIndex.format(name=False, formatter=None, **kwargs)
Render a string representation of the Index

**pandas.TimedeltaIndex.get_duplicates**

TimedeltaIndex.get_duplicates()

**pandas.TimedeltaIndex.get_indexer**

TimedeltaIndex.get_indexer(target, method=None, limit=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’}
  - 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
  Maximum number of consecutive labels in target to match for inexact matches.

**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)
```
**pandas.TimedeltaIndex.get_indexer_for**

TimedeltaIndex.get_indexer_for(target, **kwargs)

*guaranteed return of an indexer even when non-unique*

**pandas.TimedeltaIndex.get_indexer_non_unique**

TimedeltaIndex.get_indexer_non_unique(target)

*return an indexer suitable for taking from a non unique index return the labels in the same order as the target, and return a missing indexer into the target (missing are marked as -1 in the indexer); target must be an iterable*

**pandas.TimedeltaIndex.get_level_values**

TimedeltaIndex.get_level_values(level)

*Return vector of label values for requested level, equal to the length of the index*

**Parameters**

- level : int

**Returns**

- values : ndarray

**pandas.TimedeltaIndex.get_loc**

TimedeltaIndex.get_loc(key, method=None)

*Get integer location for requested label*

**Returns**

- loc : int

**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 : string / None, the type of indexer

**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*

**pandas.TimedeltaIndex.get_value_maybe_box**

TimedeltaIndex.get_value_maybe_box(series, key)
pandas.TimedeltaIndex.get_values

TimedeltaIndex.get_values()
return the underlying data as an ndarray

pandas.TimedeltaIndex.groupby

TimedeltaIndex.groupby(f)

pandas.TimedeltaIndex.holds_integer

TimedeltaIndex.holds_integer()

pandas.TimedeltaIndex.identical

TimedeltaIndex.identical(other)
Similar to equals, but check that other comparable attributes are also equal

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

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

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
pandas.TimedeltaIndex.is_boolean
TimedeltaIndex.is_boolean()

pandas.TimedeltaIndex.is_categorical
TimedeltaIndex.is_categorical()

pandas.TimedeltaIndex.is_floating
TimedeltaIndex.is_floating()

pandas.TimedeltaIndex.is_integer
TimedeltaIndex.is_integer()

pandas.TimedeltaIndex.is_lexsorted_for_tuple
timedelta_idx.is_lexsorted_for_tuple(tup)

pandas.TimedeltaIndex.is_mixed
TimedeltaIndex.is_mixed()

pandas.TimedeltaIndex.is_numeric
TimedeltaIndex.is_numeric()

pandas.TimedeltaIndex.is_object
TimedeltaIndex.is_object()

pandas.TimedeltaIndex.is_type_compatible
TimedeltaIndex.is_type_compatible(typ)

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)
pandas.TimedeltaIndex.item

TimedeltaIndex.item()
return the first element of the underlying data as a python scalar

pandas.TimedeltaIndex.join

TimedeltaIndex.join(other, how='left', level=None, return_indexers=False)
See Index.join

pandas.TimedeltaIndex.map

TimedeltaIndex.map(f)

pandas.TimedeltaIndex.max

TimedeltaIndex.max(axis=None)
return the maximum value of the Index
See also:
numpy.ndarray.max

pandas.TimedeltaIndex.min

TimedeltaIndex.min(axis=None)
return the minimum value of the Index
See also:
numpy.ndarray.min

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 unique : int

pandas.TimedeltaIndex.order

TimedeltaIndex.order(return_indexer=False, ascending=True)
Return sorted copy of Index
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.TimedeltaIndex.putmask

TimedeltaIndex.putmask(mask, value)

return a new Index of the values set with the mask

See also:
	numpy.ndarray.putmask

pandas.TimedeltaIndex.ravel

TimedeltaIndex.ravel(order='C')

return an ndarray of the flattened values of the underlying data

See also:
	numpy.ndarray.ravel

pandas.TimedeltaIndex.reindex

TimedeltaIndex.reindex(target, method=None, level=None, limit=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

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]

pandas.TimedeltaIndex.repeat

TimedeltaIndex.repeat(repeats, axis=None)

Analogous to ndarray.repeat

pandas.TimedeltaIndex.searchsorted

TimedeltaIndex.searchsorted(key, side='left')
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 or 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
In [1]: idx = MultiIndex.from_tuples([(1, 'one'), (1, 'two'),
                                (2, 'one'), (2, 'two')],
                               names=['foo', 'bar'])

In [2]: idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['baz', 'quz'])

In [3]: idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['baz', 'bar'])
```

pandas.TimedeltaIndex.set_value

TimedeltaIndex.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

pandas.TimedeltaIndex.shift

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
pandas.TimedeltaIndex.slice_indexer

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.

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, default None
  - If None, defaults to 1
- **kind**: string, default None

Returns:
- **start, end**: int

pandas.TimedeltaIndex.sort

TimedeltaIndex.sort(*args, **kwargs)

pandas.TimedeltaIndex.summary

TimedeltaIndex.summary(name=None)
return a summarized representation
pandas.TimedeltaIndex.sym_diff

TimedeltaIndex.sym_diff(other, result_name=None)
Compute the sorted symmetric difference of two Index objects.

Parameters
other : Index or array-like
result_name : str

Returns
sym_diff : Index

Notes

sym_diff contains elements that appear in either idx1 or idx2 but not both. Equivalent to the Index
created by (idx1 - idx2) + (idx2 - idx1) with duplicates dropped.

The sorting of a result containing NaN values is not guaranteed across Python versions. See GitHub issue
#6444.

Examples

```python
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.sym_diff(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the ^ operator:

```python
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

pandas.TimedeltaIndex.take

TimedeltaIndex.take(indices, axis=0)
Analogous to ndarray.take

pandas.TimedeltaIndex.to_datetime
dayfirst=False

For an Index containing strings or datetime.datetime objects, attempt conversion to DatetimeIndex

pandas.TimedeltaIndex.to_native_types

TimedeltaIndex.to_native_types(slicer=None, **kwargs)
slice and dice then format

pandas.TimedeltaIndex.to_pytimedelta

TimedeltaIndex.to_pytimedelta()
Return TimedeltaIndex as object ndarray of datetime.timedelta objects

Returns
datetimes : ndarray
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.

pandas.TimedeltaIndex.tolist

TimedeltaIndex.tolist()
return a list of the underlying data

pandas.TimedeltaIndex.transpose

TimedeltaIndex.transpose()
return the transpose, which is by definition self

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

pandas.TimedeltaIndex.unique

TimedeltaIndex.unique()
Index.unique with handling for DatetimeIndex/PeriodIndex metadata

Returns result: DatetimeIndex or PeriodIndex

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

```
pandas.TimedeltaIndex.view
```

```
TimedeltaIndex.view(cls=None)
```

### 33.10.2 Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TimedeltaIndex.days</code></td>
<td>Number of days for each element.</td>
</tr>
<tr>
<td><code>TimedeltaIndex.seconds</code></td>
<td>Number of seconds (&gt;= 0 and less than 1 day) for each element.</td>
</tr>
<tr>
<td><code>TimedeltaIndex.microseconds</code></td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td><code>TimedeltaIndex.nanoseconds</code></td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</td>
</tr>
<tr>
<td><code>TimedeltaIndex.components</code></td>
<td>Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.</td>
</tr>
</tbody>
</table>

**pandas.TimedeltaIndex.days**

```
TimedeltaIndex.days
```

Number of days for each element.

**pandas.TimedeltaIndex.seconds**

```
TimedeltaIndex.seconds
```

Number of seconds (>= 0 and less than 1 day) for each element.

**pandas.TimedeltaIndex.microseconds**

```
TimedeltaIndex.microseconds
```

Number of microseconds (>= 0 and less than 1 second) for each element.

**pandas.TimedeltaIndex.nanoseconds**

```
TimedeltaIndex.nanoseconds
```

Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

**pandas.TimedeltaIndex.components**

```
TimedeltaIndex.components
```

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

**Returns** a DataFrame
pandas: powerful Python data analysis toolkit, Release 0.16.2

pandas.TimedeltaIndex.inferred_freq

TimedeltaIndex.inferred_freq = None

33.10.3 Conversion

TimedeltaIndex.to_pytimedelta

Return TimedeltaIndex as object ndarray of datetime.timedelta objects

TimedeltaIndex.to_series(**kwargs)

Create a Series with both index and values equal to the index keys

pandas.TimedeltaIndex.to_pytimedelta

TimedeltaIndex.to_pytimedelta()

Return TimedeltaIndex as object ndarray of datetime.timedelta objects

Returns datetimes : ndarray

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.

33.11 GroupBy

GroupBy objects are returned by groupby calls: pandas.DataFrame.groupby(), pandas.Series.groupby(), etc.

33.11.1 Indexing, iteration

GroupBy.__iter__() Groupby iterator
GroupBy.groups dict {group name -> group labels}
GroupBy.indices dict {group name -> group indices}
GroupBy.get_group(name[, obj]) Constructs NDframe from group with provided name

pandas.core.groupby.GroupBy.__iter__

GroupBy.__iter__()

Groupby iterator

Returns Generator yielding sequence of (name, subsetted object)

for each group
pandas.core.groupby.GroupBy.groups

GroupBy.groups
dict {group name -> group labels}

pandas.core.groupby.GroupBy.indices

GroupBy.indices
dict {group name -> group indices}

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

pandas.Grouper([key, level, freq, axis, sort])
A Grouper allows the user to specify a groupby instruction for a target object

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
    grouper 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

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**

```python
>>> df.groupby(Grouper(key='A')) : syntactic sugar for df.groupby('A')
>>> df.groupby(Grouper(key='date', freq='60s')) : specify a resample on the column 'date'
>>> df.groupby(Grouper(level='date', freq='60s', axis=1)) :
specify a resample on the level 'date' on the columns axis with a frequency of 60s
```

**Attributes**

```python
ax

groups
```

**pandas.Grouper.ax**

Grouper.ax

**pandas.Grouper.groups**

Grouper.groups

### 33.11.2 Function application

- **GroupBy.apply**(func, *args, **kwargs) Apply function and combine results together in an intelligent way.
- **GroupBy.aggregate**(func, *args, **kwargs) Apply function and combine results together in an intelligent way.
- **GroupBy.transform**(func, *args, **kwargs) Apply function and combine results together in an intelligent way.

**pandas.core.groupby.GroupBy.apply**

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

**Returns**  applied : type depending on grouped object and function

See also:

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.

---

**pandas.core.groupby.GroupBy.aggregate**

GroupBy. **aggregate** *(func, *args, **kwargs)*

**pandas.core.groupby.GroupBy.transform**

GroupBy. **transform** *(func, *args, **kwargs)*

### 33.11.3 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupBy.count(<em>axis</em>)</td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
<tr>
<td>GroupBy.cumcount(<em>ascending</em>)</td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
<tr>
<td>GroupBy.first()</td>
<td>Compute first of group values</td>
</tr>
<tr>
<td>GroupBy.head(<em>n</em>)</td>
<td>Returns first n rows of each group.</td>
</tr>
<tr>
<td>GroupBy.last()</td>
<td>Compute last of group values</td>
</tr>
<tr>
<td>GroupBy.max()</td>
<td>Compute max of group values</td>
</tr>
<tr>
<td>GroupBy.mean()</td>
<td>Compute mean of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.median()</td>
<td>Compute median of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.min()</td>
<td>Compute min of group values</td>
</tr>
<tr>
<td>GroupBy.nth(<em>n</em>, dropna)</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>GroupBy.ohlc()</td>
<td>Compute sum of values, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.prod()</td>
<td>Compute prod of group values</td>
</tr>
<tr>
<td>GroupBy.size()</td>
<td>Compute group sizes</td>
</tr>
<tr>
<td>GroupBy.sem(<em>ddof</em>)</td>
<td>Compute standard error of the mean of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.std(<em>ddof</em>)</td>
<td>Compute standard deviation of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.sum()</td>
<td>Compute sum of group values</td>
</tr>
<tr>
<td>GroupBy.var(<em>ddof</em>)</td>
<td>Compute variance of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.tail(<em>n</em>)</td>
<td>Returns last n rows of each group</td>
</tr>
</tbody>
</table>

---

**pandas.core.groupby.GroupBy.count**

GroupBy. **count** *(axis=0)*
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.

**Examples**

```python
>>> df = pd.DataFrame([['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
...                   columns=['A'])
>>> df
   A
0 a
1 a
2 a
3 b
4 b
5 a
>>> df.groupby('A').cumcount()
   0  1  2  3
0  0  1  2
1  1  2  3
2  1  2
3  0  1
4  0  1
5  3
```

```python
dtype: int64
```

```python
>>> df.groupby('A').cumcount(ascending=False)
   0  1  2
0  3  2
1  1  2
2  1  2
3  1  3
4  0  1
5  0
```

dtype: int64

pandas.core.groupby.GroupBy.first

GroupBy.first()
Compute first of group values

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.
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
df.groupby('A').head(1)
A  B
0 1 2
2 5 6
```

**pandas.core.groupby.GroupBy.last**

```
GroupBy.last()
```

Compute last of group values

**pandas.core.groupby.GroupBy.max**

```
GroupBy.max()
```

Compute max of group values

**pandas.core.groupby.GroupBy.mean**

```
GroupBy.mean()
```

Compute mean of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**pandas.core.groupby.GroupBy.median**

```
GroupBy.median()
```

Compute median of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**pandas.core.groupby.GroupBy.min**

```
GroupBy.min()
```

Compute min of group values

**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’

**Examples**

```python
g = df.groupby('A')
g.nth(0)  # first row
A B
0 1 NaN
2 5 6
```

```python
g.nth(1)  # second row
A B
1 1 4
```

```python
g.nth(-1)  # last row
A B
1 2 5
```

```python
g.nth(0, dropna='any')  # first row, dropna='any'
B
A
1 4
5 6
```

```python
g.nth(1, dropna='any')  # second row, dropna='any'
# NaNs denote group exhausted when using dropna
B
A
1 NaN
5 NaN
```

**pandas.core.groupby.GroupBy.ohlc**

GroupBy.**ohlc**()
Compute sum of values, excluding missing values. For multiple groupings, the result index will be a MultiIndex.

**pandas.core.groupby.GroupBy.prod**

GroupBy.**prod**()
Compute prod of group values.

**pandas.core.groupby.GroupBy.size**

GroupBy.**size**()
Compute group sizes.

**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.
pandas.core.groupby.GroupBy.std

GroupBy.\texttt{std}(\texttt{ddof}=1)

Compute standard deviation of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

pandas.core.groupby.GroupBy.sum

GroupBy.\texttt{sum}()

Compute sum of group values.

pandas.core.groupby.GroupBy.var

GroupBy.\texttt{var}(\texttt{ddof}=1)

Compute variance of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

pandas.core.groupby.GroupBy.tail

GroupBy.\texttt{tail}(\texttt{n}=5)

Returns last \texttt{n} rows of each group.

Essentially equivalent to \texttt{apply(lambda x: x.tail(n))}, except ignores \texttt{as_index} flag.

Examples

```python
>>> df = DataFrame([[\'a\', 1], [\'a\', 2], [\'b\', 1], [\'b\', 2]],
columns=[\'A\', \'B\'])
>>> df.groupby(\'A\').tail(1)
   A  B
0  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 \texttt{SeriesGroupBy} and \texttt{DataFrameGroupBy} objects, but may differ slightly, usually in that the \texttt{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.

- \texttt{DataFrameGroupBy.bfill}([\texttt{axis, inplace, ...}]) Synonym for \texttt{NDFrame.fillna(method=\'bfill\')}
- \texttt{DataFrameGroupBy.cummax}([\texttt{axis, dtype, out, ...}]) Return cumulative max over requested axis.
- \texttt{DataFrameGroupBy.cummin}([\texttt{axis, dtype, out, ...}]) Return cumulative min over requested axis.
- \texttt{DataFrameGroupBy.cumprod}([\texttt{axis, dtype, out, ...}]) Return cumulative prod over requested axis.
- \texttt{DataFrameGroupBy.cumsum}([\texttt{axis, dtype, out, ...}]) Return cumulative sum over requested axis.
- \texttt{DataFrameGroupBy.describe}([\texttt{...}]) Generate various summary statistics, excluding NaN values.
- \texttt{DataFrameGroupBy.all}([\texttt{axis, bool_only, ...}]) Return whether all elements are True over requested axis.
- \texttt{DataFrameGroupBy.any}([\texttt{axis, bool_only, ...}]) Return whether any element is True over requested axis.
- \texttt{DataFrameGroupBy.corr}([\texttt{method, min_periods}]) Compute pairwise correlation of columns, excluding NA/null values.
### Table 33.117 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrameGroupBy.cov(min_periods)</code></td>
<td>Compute pairwise covariance of columns, excluding NA/null values</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.diff(periods, axis)</code></td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.ffill(axis, inplace, ...)</code></td>
<td>Synonym for NDFrame.fillna(method='ffill')</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.fillna(value, method, ...)</code></td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.hist(data[, column, by, ...])</code></td>
<td>Draw histogram of the DataFrame’s series using matplotlib / pylab.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.idxmax(axis, skipna)</code></td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.idxmin(axis, skipna)</code></td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.mad(axis, skipna, level)</code></td>
<td>Return the mean absolute deviation of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.pct_change(periods, ...)</code></td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.plot(data[, x, y, kind, ...])</code></td>
<td>Make plots of DataFrame using matplotlib / pylab.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.quantile(q, axis, ...)</code></td>
<td>Return values at the given quantile over requested axis, a la numpy.percentile.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.rank(axis, numeric_only, ...)</code></td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.resample(rule[, how, axis, ...])</code></td>
<td>Convenience method for frequency conversion and resampling of regular time-series data.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.shift(periods, freq, axis)</code></td>
<td>Shift index by desired number of periods with an optional time freq</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.skew(axis, skipna, level, ...)</code></td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.take(indices[, axis, ...])</code></td>
<td>Analogous to ndarray.take</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.tshift(periods, freq, axis)</code></td>
<td>Shift the time index, using the index’s frequency if available</td>
</tr>
</tbody>
</table>

**pandas.core.groupby.DataFrameGroupBy.bfill**

DataFrameGroupBy.bfill (axis=None, inplace=False, limit=None, downcast=None)

Synonym for NDFrame.fillna(method='bfill')

**pandas.core.groupby.DataFrameGroupBy.cummax**

DataFrameGroupBy.cummax (axis=None, dtype=None, out=None, skipna=True, **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**

- **max** : Series

**pandas.core.groupby.DataFrameGroupBy.cummin**

DataFrameGroupBy.cummin (axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative min 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**

- **min** : Series
pandas.core.groupby.DataFrameGroupBy.cumprod

DataFrameGroupBy.cumprod(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative prod over requested axis.

Parameters

axis : {index (0), columns (1)}

dtype : None

skipna : boolean, default True

Out

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns

prod : Series

pandas.core.groupby.DataFrameGroupBy.cumsum

DataFrameGroupBy.cumsum(axis=None, dtype=None, out=None, skipna=True, **kwargs)

Return cumulative sum over requested axis.

Parameters

axis : {index (0), columns (1)}

dtype : None

skipna : boolean, default True

Out

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns

sum : Series

pandas.core.groupby.DataFrameGroupBy.describe

DataFrameGroupBy.describe(percentile_width=None, percentiles=None, include=None, exclude=None)

Generate various summary statistics, excluding NaN values.

Parameters

percentile_width : float, deprecated

The percentile_width argument will be removed in a future version. Use percentiles instead. width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

percentiles : array-like, optional

The percentiles to include in the output. Should all be in the interval [0, 1]. By default percentiles is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

include, exclude : list-like, ‘all’, or None (default)

Specify the form of the returned result. Either:

- None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
- A list of dtypes or strings to be included/excluded. To select all numeric types use numpy numpy.number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
- If include is the string ‘all’, the output column-set will match the input one.

Returns

summary: NDFrame of summary statistics

See also:

DataFrame.select_dtypes
**Notes**

The output DataFrame index depends on the requested dtypes:

For numeric dtypes, it will include: count, mean, std, min, max, and lower, 50, and upper percentiles.

For object dtypes (e.g. timestamps or strings), the index will include the count, unique, most common, and frequency of the most common. Timestamps also include the first and last items.

For mixed dtypes, the index will be the union of the corresponding output types. Non-applicable entries will be filled with NaN. Note that mixed-dtype outputs can only be returned from mixed-dtype inputs and appropriate use of the include/exclude arguments.

If multiple values have the highest count, then the *count* and *most common* pair will be arbitrarily chosen from among those with the highest count.

The include, exclude arguments are ignored for Series.

**pandas.core.groupby.DataFrameGroupBy.all**

```
DataFrameGroupBy.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 data. If None, will attempt to use everything, then use only boolean data

**Returns**

- **all**: Series or DataFrame (if level specified)

**pandas.core.groupby.DataFrameGroupBy.any**

```
DataFrameGroupBy.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 data. If None, will attempt to use everything, then use only boolean data

**Returns**

- **any**: Series or DataFrame (if level specified)
pandas.core.groupby.DataFrameGroupBy.corr

DataFrameGroupBy.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.core.groupby.DataFrameGroupBy.cov

DataFrameGroupBy.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.core.groupby.DataFrameGroupBy.diff

DataFrameGroupBy.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.core.groupby.DataFrameGroupBy.ffill

DataFrameGroupBy.ffill(axis=None, inplace=False, limit=None, downcast=None)
Synonym for NDFrame.fillna(method='ffill')
**pandas: powerful Python data analysis toolkit, Release 0.16.2**

**pandas.core.groupby.DataFrameGroupBy.fillna**

DataFrameGroupBy.\texttt{fillna}(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)

Fill NA/NaN values using the specified method

**Parameters**

\begin{itemize}
  \item \texttt{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.
  \item \texttt{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
  \item \texttt{axis} : \{0, 1, 'index', 'columns'\}
  \item \texttt{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).
  \item \texttt{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.
  \item \texttt{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)
\end{itemize}

Returns \texttt{filled} : DataFrame

See also:

\begin{itemize}
  \item \texttt{reindex, asfreq}
\end{itemize}

**pandas.core.groupby.DataFrameGroupBy.hist**

DataFrameGroupBy.\texttt{hist}(data=None, 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**

\begin{itemize}
  \item \texttt{data} : DataFrame
    
    \begin{itemize}
      \item \texttt{column} : string or sequence
        
        If passed, will be used to limit data to a subset of columns
      \end{itemize}
  \item \texttt{by} : object, optional
    
    If passed, then used to form histograms for separate groups
  \item \texttt{grid} : boolean, default True
    
    Whether to show axis grid lines
\end{itemize}
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: (optional) a tuple (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.core.groupby.DataFrameGroupBy.idxmax

DataFrameGroupBy.idxmax( axis=0, skipna=True)
Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

Parameters axis : {0 or ‘index’, 1 or ‘columns’}, default 0
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.core.groupby.DataFrameGroupBy.idxmin

DataFrameGroupBy.idxmin(axis=0, skipna=True)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Parameters**

- **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0
  - 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.core.groupby.DataFrameGroupBy.irow

DataFrameGroupBy.irow(i, copy=False)

pandas.core.groupby.DataFrameGroupBy.mad

DataFrameGroupBy.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, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns**

mad : Series or DataFrame (if level specified)

pandas.core.groupby.DataFrameGroupBy.pct_change

DataFrameGroupBy.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.core.groupby.DataFrameGroupBy.plot**

DataFrameGroupBy.plot(data=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.

**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
         Title to use for the plot

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)

**layout**: tuple (optional)

(rows, columns) for the layout of the plot

**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.core.groupby.DataFrameGroupBy.quantile

DataFrameGroupBy.quantile(q=0.5, axis=0, numeric_only=True)

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

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

>>> 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.core.groupby.DataFrameGroupBy.rank

DataFrameGroupBy.rank(axis=0, numeric_only=None, method='average', 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

Ranks over columns (0) or rows (1)

numeric_only : boolean, default None

Include only float, int, boolean data

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

**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**: DataFrame

**pandas.core.groupby.DataFrameGroupBy.resample**

DataFrameGroupBy.resample (rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0)
Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** **rule**: string
  the offset string or object representing target conversion

**how**: string
  method for down- or re-sampling, default to ‘mean’ for downsampling

**axis**: int, optional, default 0

**fill_method**: string, default None
  fill_method for upsampling

**closed**: {'right', 'left'}
  Which side of bin interval is closed

**label**: {'right', 'left'}
  Which bin edge label to label bucket with

**convention**: {'start', 'end', ‘s’, ‘e’}

**kind**: “period”/”timestamp”

**loffset**: timedelta
  Adjust the resampled time labels

**limit**: int, default None
  Maximum size gap to when reindexing with fill_method

**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
pandas.core.groupby.DataFrameGroupBy.shift

DataFrameGroupBy.shift(periods=1, freq=None, axis=0, **kwargs)
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 datetools module or time rule (e.g. ‘EOM’). See Notes.
- **axis**: {0, 1, ‘index’, ‘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.core.groupby.DataFrameGroupBy.skew

DataFrameGroupBy.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, 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 data. If None, will attempt to use everything, then use only numeric data

Returns

- **skew**: Series or DataFrame (if level specified)

pandas.core.groupby.DataFrameGroupBy.take

DataFrameGroupBy.take(indices, axis=0, convert=True, is_copy=True)
Analogous to ndarray.take

Parameters

- **indices**: list / array of ints
- **axis**: int, default 0
- **convert**: translate neg to pos indices (default)
- **is_copy**: mark the returned frame as a copy

Returns

- **taken**: type of caller
**pandas.core.groupby.DataFrameGroupBy.tshift**

DataFrameGroupBy.tshift *(periods=1, freq=None, axis=0, **kwargs)*

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 datetools 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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeriesGroupBy.nlargest([n, take_last])</td>
<td>Return the largest n elements.</td>
</tr>
<tr>
<td>SeriesGroupBy.nsmallest([n, take_last])</td>
<td>Return the smallest n elements.</td>
</tr>
<tr>
<td>SeriesGroupBy.nunique([dropna])</td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td>SeriesGroupBy.unique()</td>
<td>Return array of unique values in the object.</td>
</tr>
<tr>
<td>SeriesGroupBy.value_counts([normalize, ...])</td>
<td>Returns object containing counts of unique values.</td>
</tr>
</tbody>
</table>

**pandas.core.groupby.SeriesGroupBy.nlargest**

SeriesGroupBy.nlargest *(n=5, take_last=False)*

Return the largest n elements.

**Parameters**

- **n**: int
  Return this many descending sorted values

- **take_last** [bool]
  Where there are duplicate values, take the last duplicate

**Returns**

- **top_n**: Series
  The n largest values in the Series, in sorted order

**Notes**

Faster than .order(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(1e6))
>>> s.nlargest(10)  # only sorts up to the N requested

pandas.core.groupby.SeriesGroupBy.nsmallest

SeriesGroupBy.nsmallest (n=5, take_last=False)
Return the smallest n elements.

Parameters n : int
Return this many ascending sorted values

take_last [bool] Where there are duplicate values, take the last duplicate

Returns bottom_n : Series
The n smallest values in the Series, in sorted order

Notes
Faster than .order().head(n) for small n relative to the size of the Series object.

Examples

>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(1e6))
>>> s.nsmallest(10)  # only sorts up to the N requested

pandas.core.groupby.SeriesGroupBy.nunique

SeriesGroupBy.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.core.groupby.SeriesGroupBy.unique

SeriesGroupBy.unique()
Return array of unique values in the object. Significantly faster than numpy.unique. Includes NA values.

Returns uniques : ndarray
pandas.core.groupby.SeriesGroupBy.value_counts

SeriesGroupBy.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

The following methods are available only for DataFrameGroupBy objects.

DataFrameGroupBy.corrwith(other[, axis, drop]) Compute pairwise correlation between rows or columns of two DataFrame objects.

DataFrameGroupBy.boxplot(grouped[, ...]) Make box plots from DataFrameGroupBy data.

pandas.core.groupby.DataFrameGroupBy.corrwith

DataFrameGroupBy.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.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

>>> 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)

33.12 General utility functions

33.12.1 Working with options

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>describe_option(pat[, _print_desc])</td>
<td>Prints the description for one or more registered options.</td>
</tr>
<tr>
<td>reset_option(pat)</td>
<td>Reset one or more options to their default value.</td>
</tr>
<tr>
<td>get_option(pat)</td>
<td>Retrieves the value of the specified option.</td>
</tr>
<tr>
<td>set_option(pat, value)</td>
<td>Sets the value of the specified option.</td>
</tr>
<tr>
<td>option_context(*args)</td>
<td>Context manager to temporarily set options in the with statement context.</td>
</tr>
</tbody>
</table>
pandas.describe_option

```
pandas.describe_option (pat, _print_desc=False) = <pandas.core.config.CallableDynamicDoc object at 0xb56d488c>
```

Prints the description for one or more registered options.

Call with not arguments to get a listing for all registered options.

Available options:

- `display.chop_threshold`, `colheader_justify`, `column_space`, `date_dayfirst`, `date_yearfirst`, `encoding`, `expand_frame_repr`, `float_format`, `height`, `large_repr`, `line_width`, `max_categories`, `max_columns`, `max_colwidth`, `max_info_columns`, `max_info_rows`, `max_rows`, `max_seq_items`, `memory_usage`, `mpl_style`, `multi_sparse`, `notebook_repr_html`, `pprint_nest_depth`, `precision`, `show_dimensions`, `width`
- `io.excel.xls.[writer]`
- `io.excel.xlsm.[writer]`
- `io.excel.xlsx.[writer]`
- `io.hdf.[default_format, dropna_table]`
- `mode.[chained_assignment, sim_interactive, 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:

- `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 core.format.EngFormatter for an example. [default: None] [currently: None]

display.height [int] Deprecated. [default: 60] [currently: 15] (Deprecated, use display.max_rows instead.)

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.line_width [int] Deprecated. [default: 80] [currently: 80] (Deprecated, use display.width instead.)

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 then 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 or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. [default: True] [currently: True]

display.mpl_style [bool] Setting this to ‘default’ will modify the rcParams used by matplotlib to give plots a more pleasing visual style by default. Setting this to None/False restores the values to their initial value. [default: None] [currently: None]

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: 7] [currently: 7]

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.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can
be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython
qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width.
[default: 80] [currently: 80]

io.excel.xls.writer [string] The default Excel writer engine for ‘xls’ files. Available options: ‘xlwt’ (the de-
default). [default: xlwt] [currently: xlwt]

(the default). [default: openpyxl] [currently: openpyxl]

default), ‘openpyxl’. [default: xlsxwriter] [currently: xlsxwriter]

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: True] [currently: True]

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_null [boolean] True means treat None, NaN, INF, -INF as null (old way), False means None
and NaN are null, but INF, -INF are not null (new way). [default: False] [currently: False]

pandas.reset_option

pandas.reset_option(pat) = <pandas.core.config.CallableDynamicDoc object at 0xb56d486c>
Reset one or more options to their default value.
Pass “all” as argument to reset all options.
Available options:

•display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding,
expand_frame_repr, float_format, height, large_repr, line_width, max_categories, max_columns,
max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage,
mpl_style, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions, width]

•io.excel.xls.[writer]
•io.excel.xlsm.[writer]
•io.excel.xlsx.[writer]
•io.hdf.[default_format, dropna_table]
•mode.[chained_assignment, sim_interactive, 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:

- **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 core.format.EngFormatter for an example. [default: None] [currently: None]
- **display.height** [int] Deprecated. [default: 60] [currently: 15] (Deprecated, use display.max_rows instead.)
- **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.line_width** [int] Deprecated. [default: 80] [currently: 80] (Deprecated, use display.width instead.)
- **display.max_categories** [int] This sets the maximum number of categories pandas should output when printing out a Categorical or a Series of dtpe “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 then 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 or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. [default: True] [currently: True]

display.mpl_style [bool] Setting this to ‘default’ will modify the rcParams used by matplotlib to give plots a more pleasing visual style by default. Setting this to None/False restores the values to their initial value. [default: None] [currently: None]

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: 7] [currently: 7]

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.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]


io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: True] [currently: True]

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_null  [boolean] True means treat None, NaN, INF, -INF as null (old way), False means None and NaN are null, but INF, -INF are not null (new way). [default: False] [currently: False]

pandas.get_option

pandas.get_option(pat) = <pandas.core.config.CallableDynamicDoc object at 0xb56c4e8c>

Retrieves the value of the specified option.

Available options:

- display.chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height, large_repr, line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, mpl_style, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions, width
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- mode.[chained_assignment, sim_interactive, 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  OptionErrors : if no such option exists

Notes

The available options with their descriptions:

- 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 DataFrameFor-
matter. [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 core.format.EngFormatter for an example. [default: None] [currently: None]

display.height [int] Deprecated. [default: 60] [currently: 15] (Deprecated, use display.max_rows instead.)

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.line_width [int] Deprecated. [default: 80] [currently: 80] (Deprecated, use display.width instead.)

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]

- In case python/IPython is running in a terminal and large_repr equals ‘info’ 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_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 then 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]

- In case python/IPython is running in a terminal and large_repr equals ‘info’ 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 or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. [default: True] [currently: True]

display.mpl_style [bool] Setting this to ‘default’ will modify the rcParams used by matplotlib to give plots a more pleasing visual style by default. Setting this to None/False restores the values to their initial value. [default: None] [currently: None]

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: 7] [currently: 7]

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.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]


io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: True] [currently: True]

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_null [boolean] True means treat None, NaN, INF, -INF as null (old way). False means None and NaN are null, but INF, -INF are not null (new way). [default: False] [currently: False]

pandas.set_option

pandas.set_option(pat, value) = <pandas.core.config.CallableDynamicDoc object at 0xb56d484c>

Sets the value of the specified option.

Available options:

•display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format, height, large_repr, line_width, max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, mpl_style, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions, width]

•io.excel.xls.[writer]

•io.excel.xlsm.[writer]

•io.excel.xlsx.[writer]

•io.hdf.[default_format, dropna_table]

•mode.[chained_assignment, sim_interactive, use_inf_as_null]

Parameters pat : str
Regexp which should match a single option. Note: partial matches are supported for convenience, 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:

- **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 core.format.EngFormatter for an example. [default: None] [currently: None]

- **display.height** [int] Deprecated. [default: 60] [currently: 15] (Deprecated, use display.max_rows instead.)

- **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.line_width** [int] Deprecated. [default: 80] [currently: 80] (Deprecated, use display.width instead.)

- **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]

33.12. General utility functions 1571
**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 then 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 or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called.  [default: True] [currently: True]

**display.mpl_style**  [bool] Setting this to ‘default’ will modify the rcParams used by matplotlib to give plots a more pleasing visual style by default. Setting this to None/False restores the values to their initial value.  [default: None] [currently: None]

**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: 7] [currently: 7]

**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.width**  [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width.  [default: 80] [currently: 80]


io.hdf.dropna_table  [boolean] drop ALL nan rows when appending to a table [default: True] [currently: True]

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_null  [boolean] True means treat None, NaN, INF, -INF as null (old way), False means None
                          and NaN are null, but INF, -INF are not null (new way). [default: False] [currently: False]

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):
    ...
This section will provide a look into some of pandas internals.

### 34.1 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 in64 values)
- **PeriodIndex**: An Index object with Period elements

These are range generates to 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

34.1.1 MultiIndex

Internally, the MultiIndex consists of a few things: the levels, the integer labels, and the level names:

In [1]: index = 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.

34.2 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

Note: You can find a nice example in geopandas project.
34.2.1 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>NotImplementedError</td>
<td>Series</td>
<td>DataFrame</td>
</tr>
<tr>
<td>_constructor_expanddim</td>
<td>DataFrame</td>
<td>Panel</td>
<td>NotImplementedError</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
   A  B  C
0 1  4  7
1 2  5  8
2 3  6  9
```

34.2. Subclassing pandas Data Structures
>>> type(df)
<class '__main__.SubclassedDataFrame'>

>>> sliced1 = df[['A', 'B']]
>>> sliced1
   A  B
0  1  4
1  2  5
2  3  6

>>> type(sliced1)
<class '__main__.SubclassedDataFrame'>

>>> sliced2 = df['A']
>>> sliced2
0  1
1  2
2  3
Name: A, dtype: int64

>>> type(sliced2)
<class '__main__.SubclassedSeries'>

34.2.2 Define Original Properties

To let original data structures have additional properties, you should let pandas knows what properties are added. pandas maps unknown properties to data names overriding __getattribute__. Defining original properties can be done in one of 2 ways:

1. Define _internal_names and _internal_names_set for temporary properties which WILL NOT be passed to manipulation results.
2. Define _metadata for normal properties which will be passed to manipulation results.

Below is an example to define 2 original properties, “internal_cache” as a temporary property and “added_property” as a normal property

class SubclassedDataFrame2(DataFrame):

  # temporary properties
  _internal_names = DataFrame._internal_names + ['internal_cache']
  _internal_names_set = set(_internal_names)

  # normal properties
  _metadata = ['added_property']

@property
  def _constructor(self):
    return SubclassedDataFrame2

>>> df = SubclassedDataFrame2({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```python
>>> 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
```
This is the list of changes to pandas between each release. For full details, see the commit logs at http://github.com/pydata/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/pydata/pandas
- Binary installers on PyPI: http://pypi.python.org/pypi/pandas
- Documentation: http://pandas.pydata.org

35.1 pandas 0.16.2

Release date: (June 12, 2015)

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.

35.1.1 Thanks

- Andrew Rosenfeld
- Artemy Kolchinsky
- Bernard Willers
- Christer van der Meeren
- Christian Hudon
- Constantine Glen Evans
35.2 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.
35.2.1 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
- Phillip Cloud
- Pietro Battiston
- Roy Hyunjin Han
- Sam Zhang
- Scott Sanderson
- Stephan Hoyer
- Tiago Antao
35.3 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.

### 35.3.1 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
35.4 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 Whatstnew overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.2.
35.4.1 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
- Victor Chaves
- Victor Salgado
- Vikram Bhandoh
- WANG Aiyong
- Will Holmgren
- behzad nouri
- broessli
35.5 pandas 0.15.1

Release date: (November 9, 2014)

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 What's new overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.1.

35.5.1 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
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.

35.6.1 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 Stofer
• 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
• jnmclarty
• 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
• sinhhrs
• someben
• stahlous
• stas-sl
• Stephan Hoyer
• thatneat
• tom-alcorn
• TomAugspurger
• Tom Augspurger
• Tony Lorenzo
• unknown
• unutbu
• Wes Turner
• Wilfred Hughes
• Yevgeniy Grechka
• Yoshiki Vázquez Baeza
• zachcp
35.7 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.

35.7.1 Thanks

- Andrew Rosenfeld
- Andy Hayden
- Benjamin Adams
- Benjamin M. Gross
- Brian Quistorff
- Brian Wignall
- bwignall
- clham
- Daniel Waeber
- David Bew
- David Stephens
- DSM
- dsm054
- helger
- immerrr
- Jacob Schae
- jaimefrio
- Jan Schulz
- John David Reaver
- John W. O’Brien
- Joris Van den Bossche
35.8 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.

### 35.8.1 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
35.9 pandas 0.13.1

Release date: (February 3, 2014)

35.9.1 New Features

• Added date_format and datetime_format attribute to ExcelWriter. (GH4133)
35.9.2 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).

35.9.3 Experimental Features

35.9.4 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)

35.9.5 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 (GH5942)
• 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).
- 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)

**35.10 pandas 0.13.0**

**Release date:** January 3, 2014
35.10.1 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)

35.10.2 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)

35.10.3 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` typed 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`'s `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 context managers. (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_curves` 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)

### 35.10.4 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 @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)

• 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 `mode='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 @tavistmorph 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)
  – Aliaised __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
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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)

35.10.5 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
  - 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)
isnull/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/DataFame, 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 klass, 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)
**Chapter 35. Release Notes**

**35.10.6 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’ (GH4440)
• 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.timedelt64 or np.offsets.DateOffset when operating with datetimes (GH4532)

• Fix arithmetic with series/datetimeindex and np.timedelt64 not working the same (GH4134) and buggy timedelta in numpy 1.6 (GH4135)

• Fix bug in pd.read_clipboard on windows with PY3 (GH4561); not decoding properly

• tslib.get_period_field() and tslib.get_period_field_arr() now raise if code argument out of range (GH4519, GH4520)

• Fix boolean indexing on an empty series loses index names (GH4235), infer_dtype works with empty arrays.

• Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue (GH3317)

• Fixed issue where DataFrame.apply was reraising exceptions incorrectly (causing the original stack trace to be truncated).

• Fix selection with ix/loc and non_unique selectors (GH4619)

• Fix assignment with iloc/loc involving a dtype change in an existing column (GH4312, GH5702) have internal setitem_with_indexer in core/indexing to use Block.setitem

• Fixed bug where thousands operator was not handled correctly for floating point numbers in csv_import (GH4322)

• Fix an issue with CacheableOffset not properly being used by many DateOffset; this prevented the DateOffset from being cached (GH4609)

• Fix boolean comparison with a DataFrame on the lhs, and a list/tuple on the rhs (GH4576)

• Fix error/dtype conversion with setitem of None on Series/DataFrame (GH4667)

• Fix decoding based on a passed in non-default encoding in pd.read_stata (GH4626)

• Fix DataFrame.from_records with a plain-vanilla ndarray. (GH4727)

• Fix some inconsistencies with Index.rename and MultiIndex.rename, etc. (GH4718, GH4628)

• Bug in using iloc/loc with a cross-sectional and duplicate indicies (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)
• Fix 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)

35.11 pandas 0.12.0

Release date: 2013-07-24
35.11.1 New Features

- `pd.read_html()` can now parse HTML strings, files or urls and returns a list of DataFrames 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` keyword. (GH3860)

35.11.2 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)
  - 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)

DatetimeIndexes 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 datetime.min and datetime.max (respectively), thanks @SleepingPills

read_html now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

35.11.3 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]`) (GH3423)

• 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 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)

35.11.4 Experimental Features

• Added experimental CustomBusinessDay class to support DateOffsets with custom holiday calendars and custom weekmasks. (GH2301)

35.11.5 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/NaT) 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 `r` delimiter and quoted fields (GH3453)
- 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 were 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 PandaAutoDateLocator where invert_xaxis triggered incorrectly MilliSecondLocator (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)

35.12 pandas 0.11.0

Release date: 2013-04-22

35.12.1 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)

### 35.12.2 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.

```python
In [1]: p = Panel(randn(3,4,4),items=['ItemA', 'ItemB', 'ItemC'],
   ...:     major_axis=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()
Out[3]:
```
pandas: powerful Python data analysis toolkit, Release 0.16.2

```
<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</td>
<td>0.119209</td>
<td>-1.044236</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()
```

```
Out[4]:
2001-01-02 -0.282863
2001-01-03 -0.173215
2001-01-04 -2.104569
2001-01-05 -0.706771
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)

```
In [5]: idx = date_range("2001-10-1", periods=5, freq='M')
In [6]: ts = Series(np.random.rand(len(idx)),index=idx)
In [7]: ts['2001']
```

```
Out[7]:
2001-10-31  0.838796
2001-11-30  0.897333
2001-12-31  0.732592
Freq: M, dtype: float64
```

```
In [8]: df = DataFrame(dict(A = ts))
In [9]: df['2001']
```

```
Out[9]:
   A
2001-10-31  0.838796
2001-11-30  0.897333
2001-12-31  0.732592
```

- **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)

35.12.3 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.

### 35.12.4 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(2001,1,0,0),))`, 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 similiar 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)
• Upcasting/split blocks when needed in a mixed DataFrame when setting 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 given 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)

35.13 pandas 0.10.1

Release date: 2013-01-22

35.13.1 New Features

• Add data interface to World Bank WDI pandas.io.wb (GH2592)

35.13.2 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

### 35.13.3 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 automatic indexing via `index` keyword to append
  - support `expectedrows` keyword in append to inform PyTables about the expected table size
  - 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)

### 35.13.4 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)

35.14 pandas 0.10.0

Release date: 2012-12-17

35.14.1 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)
35.14.2 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

35.14.3 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. (GH2224)
- File parsers will not handle NA sentinel values arising from passed converter functions

35.14.4 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/lstrip/rstrip 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)
• 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)
35.14.5 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 appending 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.icol/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)

35.15 pandas 0.9.1

Release date: 2012-11-14

35.15.1 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)
35.15.2 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)

35.15.3 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)

35.15.4 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)
35.16 pandas 0.9.0

Release date: 10/7/2012

35.16.1 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)

35.16.2 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)
### 35.16.3 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 over-ride default NAs unless `keep_default_na` is set to false explicitly (GH1657)
- Enable `skipfooter` parameter in text parsers as an alias for `skip_footer`

### 35.16.4 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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)
pandas: powerful Python data analysis toolkit, Release 0.16.2

- 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 ._utcoffset 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)

35.17 pandas 0.8.1

Release date: July 22, 2012

35.17.1 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)

35.17.2 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 dtype (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)

35.17.3 Bug Fixes

• Fix NA handling in DataFrame.to_panel (GH1582)
  • Handle TypeError issues inside PyObject_RichCompareBool calls in khash (GH1518)
  • 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)

35.18 pandas 0.8.0

Release date: 6/29/2012

35.18.1 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 tuple (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

**35.18.2 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_dtypes 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)

35.18.3 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)

35.18.4 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 xlrd (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)

35.19 pandas 0.7.3

Release date: April 12, 2012

35.19.1 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)

35.19.2 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)

35.19.3 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)

35.20 pandas 0.7.2

Release date: March 16, 2012

35.20.1 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

35.20.2 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)

35.20.3 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)

35.20.4 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)

35.21 pandas 0.7.1

Release date: February 29, 2012

35.21.1 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

35.21.2 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)
35.21.3 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 clear 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)

35.22 pandas 0.7.0

Release date: 2/9/2012

35.22.1 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
35.22.2 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 \texttt{ix} or \texttt{[]} 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 \texttt{[]} on a Series for both getting and setting (GH86)
- \texttt{[]} operator (\texttt{__getitem__} and \texttt{__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 \texttt{.ix} on DataFrame and friends (GH328)
- Rename \texttt{DataFrame.delevel} to \texttt{DataFrame.reset_index} and add deprecation warning
- \texttt{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 \texttt{LongPanel} class (GH552)
- Deprecated \texttt{Panel.to_long}, renamed to \texttt{to_frame}
- Deprecated \texttt{colSpace} argument in \texttt{DataFrame.to_string}, renamed to \texttt{col_space}
- Rename \texttt{precision} to \texttt{accuracy} in engineering float formatter (GH395)
- The default delimiter for \texttt{read_csv} is comma rather than letting \texttt{csv.Sniffer} infer it
- Rename \texttt{col_or_columns} argument in \texttt{DataFrame.drop_duplicates} (GH734)

35.22.3 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 \texttt{Series.to_string}, add \texttt{length} option (GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of \texttt{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 \texttt{reset_index} on DataFrame with a regular (non-hierarchical) index (GH476)
- Use Cythonized groupers when possible in Series/DataFrame stat ops with \texttt{level} parameter passed (GH545)
- Ported skiplist data structure to C to speed up \texttt{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 (GH GH507, various pull requests)

Unstack called on DataFrame with non-MultiIndex will return Series (GH 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)

### 35.22.4 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)

35.22.5 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
35.23 pandas 0.6.1

Release date: 12/13/2011

35.23.1 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)

35.23.2 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 (GH114)
- Add Series.from_csv function (GH482)

35.23.3 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

### 35.23.4 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)
35.23.5 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

35.24 pandas 0.6.0

Release date: 11/25/2011

35.24.1 API Changes

- Arithmetic methods like \texttt{sum} will attempt to sum dtype=object values by default instead of excluding them (GH382)

35.24.2 New Features

- Add \texttt{melt} function to \texttt{pandas.core.reshape}
- Add \texttt{level} parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Add \texttt{head} and \texttt{tail} methods to Series, analogous to to DataFrame (PR GH296)
- Add \texttt{Series.isin} function which checks if each value is contained in a passed sequence (GH289)
- Add \texttt{float_format} option to \texttt{Series.to_string}
- Add \texttt{skip_footer} (GH291) and \texttt{converters} (GH343) options to \texttt{read_csv} and \texttt{read_table}
- Add proper, tested weighted least squares to standard and panel OLS (GH GH303)
- Add \texttt{drop_duplicates} and \texttt{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)

35.24.3 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

### 35.24.4 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 swaplevel (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)

35.24.5 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
35.25 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.

35.25.1 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`
- `_lastTimeWithValue` use `last_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`
  - `_firstTimeWithValue` use `first_valid_index`
  - `_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`

### 35.25.2 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

### 35.25.3 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

### 35.25.4 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 pyx file dependencies in Cython module build (GH271)
35.25.5 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
35.25.6 Thanks

- Thomas Kluyver
- Daniel Fortunov
- Aman Thakral
- Luca Beltrame
- Wouter Overmeire

35.26 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 d enhanced features. Also, pandas can now
be installed and used on Python 3 hanks Thomas Kluyver!).

35.26.1 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)

35.26.2 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

35.26.3 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

35.26.4 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`

• 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

35.26.5 Thanks

• Thomas Kluyver

• rsamson

35.27 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

35.27.1 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

35.27.2 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

### 35.27.3 API Changes

### 35.27.4 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

### 35.27.5 Thanks

- Uri Laserson
- Scott Sinclair

### 35.28 pandas 0.4.1

**Release date:** 9/25/2011

is is primarily a bug fix release but includes some new features and improvements

### 35.28.1 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
35.28.2 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

35.28.3 API Changes

35.28.4 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

35.28.5 Thanks

- Yaroslav Halchenko
- Jeff Reback
- Skipper Seabold
- Dan Lovell
- Nick Pentreath
35.29 pandas 0.4.0

Release date: 9/12/2011

35.29.1 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

35.29.2 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).

35.29.3 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.to_string) 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
• `fillMethod` 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 dnums` 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

35.29.4 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`
• 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

35.29.5 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

35.30 pandas 0.3.0

Release date: February 20, 2011
35.30.1 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

35.30.2 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.)

35.30.3 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

35.30.4 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
pandas, 1