### 10 Indexing and Selecting Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Different Choices for Indexing (<code>loc</code>, <code>iloc</code>, and <code>ix</code>)</td>
<td>223</td>
</tr>
<tr>
<td>10.2</td>
<td>Deprecations</td>
<td>224</td>
</tr>
<tr>
<td>10.3</td>
<td>Basics</td>
<td>224</td>
</tr>
<tr>
<td>10.4</td>
<td>Attribute Access</td>
<td>226</td>
</tr>
<tr>
<td>10.5</td>
<td>Slicing ranges</td>
<td>227</td>
</tr>
<tr>
<td>10.6</td>
<td>Selection By Label</td>
<td>229</td>
</tr>
<tr>
<td>10.7</td>
<td>Selection By Position</td>
<td>231</td>
</tr>
<tr>
<td>10.8</td>
<td>Setting With Enlargement</td>
<td>234</td>
</tr>
<tr>
<td>10.9</td>
<td>Fast scalar value getting and setting</td>
<td>235</td>
</tr>
<tr>
<td>10.10</td>
<td>Boolean indexing</td>
<td>236</td>
</tr>
<tr>
<td>10.11</td>
<td>The <code>where()</code> Method and Masking</td>
<td>239</td>
</tr>
<tr>
<td>10.12</td>
<td>The <code>query()</code> Method (Experimental)</td>
<td>242</td>
</tr>
<tr>
<td>10.13</td>
<td>Take Methods</td>
<td>254</td>
</tr>
<tr>
<td>10.14</td>
<td>Duplicate Data</td>
<td>256</td>
</tr>
<tr>
<td>10.15</td>
<td>Dictionary-like <code>get()</code> method</td>
<td>256</td>
</tr>
<tr>
<td>10.16</td>
<td>Advanced Indexing with <code>.ix</code></td>
<td>257</td>
</tr>
<tr>
<td>10.17</td>
<td>The <code>select()</code> Method</td>
<td>260</td>
</tr>
<tr>
<td>10.18</td>
<td>The <code>lookup()</code> Method</td>
<td>260</td>
</tr>
<tr>
<td>10.19</td>
<td>Float64Index</td>
<td>260</td>
</tr>
<tr>
<td>10.20</td>
<td>Returning a view versus a copy</td>
<td>263</td>
</tr>
<tr>
<td>10.21</td>
<td>Fallback indexing</td>
<td>265</td>
</tr>
<tr>
<td>10.22</td>
<td>Index objects</td>
<td>266</td>
</tr>
<tr>
<td>10.23</td>
<td>Hierarchical indexing (MultiIndex)</td>
<td>267</td>
</tr>
<tr>
<td>10.24</td>
<td>Setting index metadata (<code>name(s)</code>, <code>levels</code>, <code>labels</code>)</td>
<td>279</td>
</tr>
<tr>
<td>10.25</td>
<td>Adding an index to an existing DataFrame</td>
<td>280</td>
</tr>
<tr>
<td>10.26</td>
<td>Add an index using DataFrame columns</td>
<td>280</td>
</tr>
<tr>
<td>10.27</td>
<td>Remove/reset the index, <code>reset_index</code></td>
<td>281</td>
</tr>
<tr>
<td>10.28</td>
<td>Adding an ad hoc index</td>
<td>282</td>
</tr>
<tr>
<td>10.29</td>
<td>Indexing internal details</td>
<td>283</td>
</tr>
</tbody>
</table>

### 11 Computational tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Statistical functions</td>
<td>285</td>
</tr>
<tr>
<td>11.2</td>
<td>Moving (rolling) statistics / moments</td>
<td>289</td>
</tr>
<tr>
<td>11.3</td>
<td>Expanding window moment functions</td>
<td>296</td>
</tr>
<tr>
<td>11.4</td>
<td>Exponentially weighted moment functions</td>
<td>298</td>
</tr>
</tbody>
</table>

### 12 Working with missing data

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>Missing data basics</td>
<td>301</td>
</tr>
<tr>
<td>12.2</td>
<td>Datetimes</td>
<td>303</td>
</tr>
<tr>
<td>12.3</td>
<td>Calculations with missing data</td>
<td>303</td>
</tr>
<tr>
<td>12.4</td>
<td>Cleaning / filling missing data</td>
<td>305</td>
</tr>
<tr>
<td>12.5</td>
<td>Missing data casting rules and indexing</td>
<td>319</td>
</tr>
</tbody>
</table>

### 13 Group By: split-apply-combine

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>Splitting an object into groups</td>
<td>322</td>
</tr>
<tr>
<td>13.2</td>
<td>Iterating through groups</td>
<td>326</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>13.3 Aggregation</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>13.4 Transformation</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>13.5 Filtration</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>13.6 Dispatching to instance methods</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>13.7 Flexible apply</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>13.8 Other useful features</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>14 Merge, join, and concatenate</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>14.1 Concatenating objects</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>14.2 Database-style DataFrame joining/merging</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>15 Reshaping and Pivot Tables</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>15.1 Reshaping by pivoting DataFrame objects</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>15.2 Reshaping by stacking and unstacking</td>
<td>363</td>
<td></td>
</tr>
<tr>
<td>15.3 Reshaping by Melt</td>
<td>366</td>
<td></td>
</tr>
<tr>
<td>15.4 Combining with stats and GroupBy</td>
<td>368</td>
<td></td>
</tr>
<tr>
<td>15.5 Pivot tables and cross-tabulations</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>15.6 Tiling</td>
<td>372</td>
<td></td>
</tr>
<tr>
<td>15.7 Computing indicator / dummy variables</td>
<td>373</td>
<td></td>
</tr>
<tr>
<td>15.8 Factorizing values</td>
<td>374</td>
<td></td>
</tr>
<tr>
<td>16 Time Series / Date functionality</td>
<td>377</td>
<td></td>
</tr>
<tr>
<td>16.1 Time Stamps vs. Time Spans</td>
<td>378</td>
<td></td>
</tr>
<tr>
<td>16.2 Converting to Timestamps</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>16.3 Generating Ranges of Timestamps</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>16.4 DatetimeIndex</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>16.5 DateOffset objects</td>
<td>388</td>
<td></td>
</tr>
<tr>
<td>16.6 Time series-related instance methods</td>
<td>393</td>
<td></td>
</tr>
<tr>
<td>16.7 Up- and downsampling</td>
<td>395</td>
<td></td>
</tr>
<tr>
<td>16.8 Time Span Representation</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>16.9 Converting between Representations</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>16.10 Time Zone Handling</td>
<td>401</td>
<td></td>
</tr>
<tr>
<td>16.11 Time Deltas</td>
<td>404</td>
<td></td>
</tr>
<tr>
<td>16.12 Time Deltas &amp; Reductions</td>
<td>408</td>
<td></td>
</tr>
<tr>
<td>16.13 Time Deltas &amp; Conversions</td>
<td>408</td>
<td></td>
</tr>
<tr>
<td>17 Plotting with matplotlib</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>17.1 Basic plotting: plot</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>17.2 Other plotting features</td>
<td>423</td>
<td></td>
</tr>
<tr>
<td>18 Trellis plotting interface</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td>18.1 Examples</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td>18.2 Scales</td>
<td>452</td>
<td></td>
</tr>
<tr>
<td>19 IO Tools (Text, CSV, HDF5, ...)</td>
<td>455</td>
<td></td>
</tr>
<tr>
<td>19.1 CSV &amp; Text files</td>
<td>456</td>
<td></td>
</tr>
<tr>
<td>19.2 JSON</td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>19.3 HTML</td>
<td>486</td>
<td></td>
</tr>
<tr>
<td>19.4 Excel files</td>
<td>494</td>
<td></td>
</tr>
<tr>
<td>19.5 Clipboard</td>
<td>496</td>
<td></td>
</tr>
<tr>
<td>19.6 Pickling</td>
<td>497</td>
<td></td>
</tr>
<tr>
<td>19.7 msgpack (experimental)</td>
<td>498</td>
<td></td>
</tr>
<tr>
<td>19.8 HDF5 (PyTables)</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>19.9 SQL Queries</td>
<td>525</td>
<td></td>
</tr>
<tr>
<td>19.10 Google BigQuery (Experimental)</td>
<td>526</td>
<td></td>
</tr>
</tbody>
</table>
19.11 STATA Format .................................................. 527
20 Remote Data Access .............................................. 529
  20.1 Yahoo! Finance ................................................. 529
  20.2 Google Finance ................................................. 530
  20.3 FRED ............................................................ 530
  20.4 Fama/French .................................................... 531
  20.5 World Bank ..................................................... 531
21 Enhancing Performance ........................................... 535
  21.1 Cython (Writing C extensions for pandas) ................. 535
  21.2 Expression Evaluation via eval() (Experimental) .... 539
22 Sparse data structures ............................................. 547
  22.1 SparseArray .................................................... 549
  22.2 SparseList ..................................................... 549
  22.3 SparseIndex objects ........................................... 550
23 Caveats and Gotchas ............................................... 551
  23.1 Using If/Truth Statements with Pandas .................... 551
  23.2 NaN, Integer NA values and NA type promotions ........ 552
  23.3 Integer indexing ............................................. 554
  23.4 Label-based slicing conventions ......................... 554
  23.5 Miscellaneous indexing gotchas ......................... 555
  23.6 Timestamp limitations ...................................... 557
  23.7 Parsing Dates from Text Files ............................. 557
  23.8 Differences with NumPy .................................... 558
  23.9 Thread-safety ............................................... 558
  23.10 HTML Table Parsing ........................................ 558
  23.11 Byte-Ordering Issues ...................................... 560
24 rpy2 / R interface ................................................ 561
  24.1 Transferring R data sets into Python ..................... 561
  24.2 Converting DataFrames into R objects ................... 562
  24.3 Calling R functions with pandas objects ................. 562
  24.4 High-level interface to R estimators ..................... 562
25 Pandas Ecosystem .................................................. 563
  25.1 Statsmodels .................................................... 563
  25.2 Vincent ........................................................ 563
  25.3 yhat/ggplot .................................................. 563
  25.4 Seaborn ........................................................ 563
  25.5 Geopandas ..................................................... 564
  25.6 sklearn-pandas .............................................. 564
26 Comparison with R / R libraries ................................ 565
  26.1 Base R .......................................................... 565
  26.2 zoo .............................................................. 569
  26.3 xts ............................................................. 569
  26.4 plyr ............................................................. 569
  26.5 reshape / reshape2 ......................................... 570
27 Comparison with SQL ............................................... 575
  27.1 SELECT ......................................................... 575
  27.2 WHERE ........................................................ 576
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 v0.13.1 (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
```

1.1.1 Output Formatting Enhancements

- `df.info()` view now display dtype info per column (GH5682)
- `df.info()` now honors the option `max_info_rows`, to disable null counts for large frames (GH5974)

```
In [7]: max_info_rows = pd.get_option('max_info_rows')

In [8]: df = DataFrame(dict(A = np.random.randn(10),
                         B = np.random.randn(10),
                         C = date_range('20130101',periods=10))

In [9]: df.iloc[3:6,[0,2]] = np.nan

# set to not display the null counts
In [10]: pd.set_option('max_info_rows',0)

In [11]: df.info()
```
• Add show_dimensions display option for the new DataFrame repr to control whether the dimensions print.

• 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 2001-01-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
<td>4491 days, 00:00:00</td>
</tr>
<tr>
<td>1 2004-06-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
<td>3244 days, 00:00:00</td>
</tr>
</tbody>
</table>

Now the output looks like:

```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

[2 rows x 3 columns]

1.1.2 API changes

• Add -NaN and -nan to the default set of NA values (GH5952). See NA Values.

• Added Series.str.get_dummies vectorized string method (GH6021), to extract dummy/indicator variables for separated string columns:

In [23]: s = Series(['a', 'a|b', np.nan, 'a|c'])

In [24]: s.str.get_dummies(sep='|')
Out[24]:
     a  b  c
0    1  0  0
1    1  1  0
2    0  0  0
3    1  0  1

[4 rows x 3 columns]

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

In [25]: df = DataFrame({'col':['foo', 0, np.nan]}).sort()

In [26]: df2 = DataFrame({'col':[np.nan, 0, 'foo']}, index=[2,1,0])

In [27]: df.equals(df)
Out[27]: True

In [28]: import pandas.core.common as com

In [29]: com.array_equivalent(np.array([0, np.nan]), np.array([0, np.nan]))
Out[29]: True

In [30]: np.array_equal(np.array([0, np.nan]), np.array([0, np.nan]))
Out[30]: 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:

In [31]: def applied_func(col):
   ....:     print "Apply function being called with:", col
   ....:     return col.sum()
In [32]: empty = DataFrame(columns=['a', 'b'])

In [33]: empty.apply(applied_func)
Apply function being called with: Series([], dtype: float64)
Out[33]:
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.

In [34]: empty.apply(applied_func, reduce=True)
Out[34]:
a   NaN
b   NaN
dtype: float64

In [35]: empty.apply(applied_func, reduce=False)
Out[35]:
Empty DataFrame
Columns: [a, b]
Index: []
[0 rows x 2 columns]

1.1.3 Prior Version Deprecations/Changes

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

1.1.4 Deprecations

There are no deprecations of prior behavior in 0.13.1

1.1.5 Enhancements

- `pd.read_csv` and `pd.to_datetime` learned a new `infer_datetime_format` keyword which greatly improves parsing perf in many cases. Thanks to @lexical for suggesting and @danbirken for rapidly implementing. (GH5490, GH6021)

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

  # 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 [36]: shades = ['light', 'dark']
  In [37]: colors = ['red', 'green', 'blue']
  In [38]: MultiIndex.from_product([shades, colors], names=['shade', 'color'])
  Out[38]:
  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 [39]: import pandas.util.testing as tm
  In [40]: panel = tm.makePanel(5)
  In [41]: panel
  Out[41]:
  <class 'pandas.core.panel.Panel'>
  Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
  Items axis: ItemA to ItemC
  Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
  Minor_axis: A to D
  ```

  ```python
  In [42]: panel['ItemA']
  Out[42]:
  A    B    C    D
  2000-01-03 -0.700262 -0.159861 -0.178315 1.495435
  2000-01-04 -0.237922  0.286230  0.386127 1.785587
  2000-01-05  0.803216 -0.311358  0.309259 1.135875
  2000-01-06  0.323302  1.144951 -0.328860 0.699592
  2000-01-07 -0.419578 -0.726740  0.056344 0.245373
  [5 rows x 4 columns]
  ```

  Specifying an apply that operates on a Series (to return a single element)

  ```python
  In [43]: panel.apply(lambda x: x.dtype, axis='items')
  Out[43]:
  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 [44]: panel.apply(lambda x: x.sum(), axis='major_axis')
  Out[44]:
     ItemA  ItemB  ItemC
  A -0.231243  1.074220  0.542019
  B  0.233222  0.968872 -4.067618
  C  0.244554  2.925382 -1.702876
  D  5.361861 -0.725465 -2.106863
  ```
This is equivalent to

In [45]: panel.sum('major_axis')

Out[45]:

<table>
<thead>
<tr>
<th></th>
<th>ItemA</th>
<th>ItemB</th>
<th>ItemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-0.231243</td>
<td>1.074220</td>
<td>0.542019</td>
</tr>
<tr>
<td>B</td>
<td>0.233222</td>
<td>0.968872</td>
<td>-4.067618</td>
</tr>
<tr>
<td>C</td>
<td>0.244554</td>
<td>2.925382</td>
<td>-1.702876</td>
</tr>
<tr>
<td>D</td>
<td>5.361861</td>
<td>-0.725465</td>
<td>-2.106863</td>
</tr>
</tbody>
</table>

A transformation operation that returns a Panel, but is computing the z-score across the major_axis

In [46]: result = panel.apply(
    ....:     lambda x: (x-x.mean())/x.std(),
    ....:     axis='major_axis',
    ....:)

In [47]: result

Out[47]:
<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 [48]: result['ItemA']

Out[48]:

<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>-1.081788</td>
<td>-0.289691</td>
<td>-0.741235</td>
<td>0.687524</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.317043</td>
<td>0.336096</td>
<td>1.100033</td>
<td>1.159054</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>1.405080</td>
<td>-0.502213</td>
<td>0.849282</td>
<td>0.103199</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.611265</td>
<td>1.540729</td>
<td>-1.232327</td>
<td>-0.605810</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.617515</td>
<td>-1.084921</td>
<td>0.024246</td>
<td>-1.343967</td>
</tr>
</tbody>
</table>

Panel apply() operating on cross-sectional slabs. (GH1148)

In [49]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T

In [50]: result = panel.apply(f, axis = ['items','major_axis'])

In [51]: result

Out[51]:
<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 [52]: result.loc[:,:,>'ItemA']

Out[52]:

<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>-0.842839</td>
<td>0.453596</td>
<td>-0.199453</td>
<td>0.822702</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.013312</td>
<td>-1.058639</td>
<td>0.769984</td>
<td>0.974988</td>
</tr>
</tbody>
</table>
2000-01-05  1.140828  0.267052 -0.593754  1.121503
2000-01-06  0.630766  1.073118 -0.687542  1.008418
2000-01-07 -0.895065 -0.181779 -0.162569 -0.052844

[5 rows x 4 columns]

This is equivalent to the following

```python
In [53]: result = Panel(dict((ax, f(panel.loc[::,ax]))
     ....:       for ax in panel.minor_axis ))
     ....:

Out[53]:
```

```
In [54]: result
```
```
<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 [55]: result.loc[::,:,'ItemA']
```
```
Out[55]:
```

```
A   B   C   D
2000-01-03 -0.842839  0.453596 -0.199453  0.822702
2000-01-04 -1.013312 -1.058639  0.769984  0.974988
2000-01-05  1.140828  0.267052 -0.593754  1.121503
2000-01-06  0.630766  1.073118 -0.687542  1.008418
2000-01-07 -0.895065 -0.181779 -0.162569 -0.052844
```

[5 rows x 4 columns]

### 1.1.6 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](#))

### 1.1.7 Experimental

There are no experimental changes in 0.13.1
1.1.8 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.

1.2 v0.13.0 (January 3, 2014)

This is a major release from 0.12.0 and includes a number of API changes, several new features and enhancements along with a large number of bug fixes.

Highlights include:

• support for a new index type Float64Index, and other Indexing enhancements
• HDFStore has a new string based syntax for query specification
• support for new methods of interpolation
• updated timedelta operations
• a new string manipulation method extract
• Nanosecond support for Offsets
• isin for DataFrames

Several experimental features are added, including:

• new eval/query methods for expression evaluation
• support for msgpack serialization
• an i/o interface to Google’s BigQuery

There are several new or updated docs sections including:

• Comparison with SQL, which should be useful for those familiar with SQL but still learning pandas.
• Comparison with R, idiom translations from R to pandas.
• Enhancing Performance, ways to enhance pandas performance with eval/query.

Warning: In 0.13.0 Series has internally been refactored to no longer sub-class ndarray but instead subclass NDFrame, similar to the rest of the pandas containers. This should be a transparent change with only very limited API implications. See Internal Refactoring

1.2.1 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):
  # 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

  In [3]: arr = np.array([1, 2, 3, 4])

  In [4]: arr2 = np.array([5, 3, 2, 1])

  In [5]: arr / arr2
  Out[5]: array([0, 0, 1, 4])

  In [6]: Series(arr) // Series(arr2)
  Out[6]:
  0 0
  1 0
  2 1
  3 4
  dtype: int64

  True Division

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

• Infer and downcast dtype if downcast='infer' is passed to fillna/ffill/bfill (GH4604)

• __nonzero__ for all NDFrame objects, will now raise a ValueError, this reverts back to (GH1073, GH4633) behavior. See gotchas for a more detailed discussion.

  This prevents doing boolean comparison on entire pandas objects, which is inherently ambiguous. These all will raise a ValueError.
if df:
    ....
df1 and df2
s1 and s2

Added the `.bool()` method to NDFrame objects to facilitate evaluating of single-element boolean Series:

In [1]: Series([True]).bool()
Out[1]: True

In [2]: Series([False]).bool()
Out[2]: False

In [3]: DataFrame([[True]]).bool()
Out[3]: True

In [4]: DataFrame([[False]]).bool()
Out[4]: False

• All non-Index NDFrames (Series, DataFrame, Panel, Panel4D, SparsePanel, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (add, sub, mul, etc.). SparsePanel does not support pow or mod with non-scalars. (GH3765)

• Series and DataFrame now have a `mode()` method to calculate the statistical mode(s) by axis/Series. (GH5367)

• Chained assignment will now by default warn if the user is assigning to a copy. This can be changed with the option `mode.chained_assignment`, allowed options are raise/warn/None. See the docs.

In [5]: dfc = DataFrame({'A':['aaa','bbb','ccc'],'B':[1,2,3]})

In [6]: pd.set_option('chained_assignment','warn')

The following warning / exception will show if this is attempted.

In [7]: dfc.loc[0]['A'] = 1111

Traceback (most recent call last)
...  
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

1.2. v0.13.0 (January 3, 2014)
min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element. (GH6214)

1.2.2 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 than None. This activates truncated display ("...") of long sequences in various places. (GH3391)

1.2.3 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.

1.2.4 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  5  5

[4 rows x 3 columns]

A Panel setting operation on an arbitrary axis aligns the input to the Panel

In [20]: p = pd.Panel(np.arange(16).reshape(2,4,2),
   ....:     items=['Item1','Item2'],
   ....:     major_axis=pd.date_range('2001/1/12',periods=4),
   ....:     minor_axis=['A','B'],dtype='float64')
   ....:

In [21]: p
Out[21]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to B

In [22]: p.loc[:, :,'C'] = Series([30,32],index=p.items)

In [23]: p
Out[23]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to C

In [24]: p.loc[:, :,'C']
Out[24]:

<table>
<thead>
<tr>
<th></th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-12</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>2001-01-13</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>2001-01-14</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>2001-01-15</td>
<td>30</td>
<td>32</td>
</tr>
</tbody>
</table>

[4 rows x 2 columns]

1.2.5 Float64Index API Change

- Added a new index type, Float64Index. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes [],ix,loc for scalar indexing and slicing work exactly the same. See the docs, (GH263)

Construction is by default for floating type values.

In [25]: index = Index([1.5, 2, 3, 4.5, 5])

In [26]: index
Out[26]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='object')

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`

```python
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`**

```python
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

```python
In [37]: s[2.1:4.6]
Out[37]:
3.0  2
4.5  3
dtype: int64

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`.

```python
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.

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

1.2. v0.13.0 (January 3, 2014)
1.2.6 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
2013-01-05 -0.063353 -1.719595
2013-01-06  1.018307 -1.423334
2013-01-07  0.602286  0.935929
2013-01-08  0.329999  0.894066
2013-01-09 -0.933857  0.308986
2013-01-10 -0.012390  0.253387
```

[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  0.066932 -0.929963  0.304346  0.790176
2013-01-02  0.518267  0.530211  0.289180  1.356091
2013-01-03  0.287746  1.371943 -0.284844  0.866407
2013-01-04  0.229041  0.797449  0.153394  1.250650
2013-01-05 -0.063353 -1.719595  1.078142 -0.402282
2013-01-06  1.018307 -1.423334  0.600642  2.202617
2013-01-07  0.602286  0.935929 -0.091967  1.086482
2013-01-08  0.329999  0.894066  0.196023  1.355471
2013-01-09 -0.933857  1.850906 -0.402282  0.862390
2013-01-10 -0.012390  0.253387  0.862390 -0.054772
```

[10 rows x 4 columns]

- the format keyword now replaces the table keyword; allowed values are fixed(f) or table(t) the same defaults as prior < 0.13.0 remain, e.g. put implies fixed format and append implies table format. This default format can be set as an option by setting io.hdf.default_format.

In [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’>
File path: test.h5
/df_fixed frame_table (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])
/df2 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’>
pandas: powerful Python data analysis toolkit, Release 0.13.1

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

1.2.7 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').

1.2.8 Enhancements

• df.to_clipboard() learned a new excel keyword that let’s you paste df data directly into excel (enabled by default). (GH5070).

• read_html now raises a URLError instead of catching and raising a ValueError (GH4303, GH4305)

• Added a test for read_clipboard() and to_clipboard() (GH4282)

• Clipboard functionality now works with PySide (GH4282)
• Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)

• to_dict now takes records as a possible outtype. Returns an array of column-keyed dictionaries. (GH4936)

• NaN handing in get_dummies (GH4446) with dummy_na

    # 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]:
    0 1 0
    1 0 1
    2 0 0

    [3 rows x 2 columns]

    # unless requested
    In [63]: get_dummies([1, 2, np.nan], dummy_na=True)
    Out[63]:
    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]: numpy.timedelta64(108301000030000,'ns')

    In [65]: to_timedelta('15.5us')
    Out[65]: numpy.timedelta64(15500,'ns')

    In [66]: to_timedelta(['1 days 06:05:01.00003','15.5us','nan'])
    Out[66]:
    0 1 days, 06:05:01.000030
    1 0 days, 00:00:00.000016
    2 NaT
dtype: timedelta64[ns]

    In [67]: to_timedelta(np.arange(5),unit='s')
    Out[67]:
    0 00:00:00
    1 00:00:01
    2 00:00:02
    3 00:00:03
    4 00:00:04
dtype: timedelta64[ns]

    In [68]: to_timedelta(np.arange(5),unit='d')
    Out[68]:

1.2. v0.13.0 (January 3, 2014)
A Series of dtype timedelta64[ns] can now be divided by another timedelta64[ns] object, or astyped to yield a float64 dtype Series. This is frequency conversion. See the docs for the docs.

In [69]: from datetime import timedelta

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

In [71]: td[2] += np.timedelta64(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 NaN
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
Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series

```python
In [78]: td * -1
Out[78]:
0   -31 days, 00:00:00
1   -31 days, 00:00:00
2  -31 days, 00:05:03
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`

```python
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

```python
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.

```python
In [84]: td.mean()
Out[84]:
0   31 days, 00:01:41
dtype: timedelta64[ns]

In [85]: td.quantile(.1)
Out[85]: numpy.timedelta64(2678400000000000,'ns')
```
- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to `scipy.stats.gaussian_kde()` (for scipy >= 0.11.0) to set the bandwidth, and to `gkde.evaluate()` to specify the indices at which it is evaluated, respectively. See scipy docs. (GH4298)

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

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

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

  ```python
  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

  ```python
  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.

  ```python
  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   3
  [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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00, ..., 2013-01-01 00:00:00.000000020]
Length: 5, Freq: 5N, Timezone: None
```

or with frequency as offset

```python
In [91]: date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
Out[91]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00, ..., 2013-01-01 00:00:00.000000020]
Length: 5, Freq: 5N, Timezone: 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', tz=None)
```

• 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
[4 rows x 2 columns]
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 True
2  True  False
3  True  False
[4 rows x 2 columns]
```
In [99]: df[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

    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 tofillna’s limit:

    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.

In [104]: np.random.seed(123)
In [105]: df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
    ....: "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 [106]: df["id"] = df.index
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]

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)

### 1.2.9 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,
In [109]: nrows, ncols = 20000, 100

In [110]: df1, df2, df3, df4 = [DataFrame(randn(nrows, ncols))
.....: for _ in xrange(4)]

# eval with NumExpr backend
In [111]: %timeit pd.eval('df1 + df2 + df3 + df4')
10 loops, best of 3: 31.6 ms per loop

# pure Python evaluation
In [112]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 74.8 ms per loop

For more details, see the the docs

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

In [113]: df = DataFrame(randn(10, 2), columns=['a', 'b'])

In [114]: df.eval('a + b')
Out[114]:
   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,

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 df where a < b < c evaluates to True. For more details see the the docs.

• pd.read_msgpack() and pd.to_msgpack() are now a supported method of serialization of arbitrary pandas (and python objects) in a lightweight portable binary format. See the docs

Warning: Since this is an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.
In [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

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.io.gbq provides a simple way to extract from, and load data into, Google’s BigQuery Data Sets by way of pandas DataFrames. BigQuery is a high performance SQL-like database service, useful for performing ad-hoc queries against extremely large datasets. See the docs
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:

<table>
<thead>
<tr>
<th>MONTH</th>
<th>Min Tem</th>
<th>Mean Temp</th>
<th>Max Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-53.336667</td>
<td>39.827892</td>
<td>89.770968</td>
</tr>
<tr>
<td>2</td>
<td>-49.837500</td>
<td>43.685219</td>
<td>93.437932</td>
</tr>
<tr>
<td>3</td>
<td>-77.926087</td>
<td>48.708355</td>
<td>96.099998</td>
</tr>
<tr>
<td>4</td>
<td>-82.892858</td>
<td>55.070087</td>
<td>97.317240</td>
</tr>
<tr>
<td>5</td>
<td>-92.378261</td>
<td>61.428117</td>
<td>102.042856</td>
</tr>
<tr>
<td>6</td>
<td>-77.703334</td>
<td>65.858888</td>
<td>102.900000</td>
</tr>
<tr>
<td>7</td>
<td>-87.821428</td>
<td>68.169663</td>
<td>106.510714</td>
</tr>
<tr>
<td>8</td>
<td>-89.431999</td>
<td>68.614215</td>
<td>105.500000</td>
</tr>
<tr>
<td>9</td>
<td>-86.611112</td>
<td>63.436935</td>
<td>107.142856</td>
</tr>
<tr>
<td>10</td>
<td>-78.209677</td>
<td>56.880838</td>
<td>92.103333</td>
</tr>
<tr>
<td>11</td>
<td>-50.125000</td>
<td>48.861228</td>
<td>94.996428</td>
</tr>
<tr>
<td>12</td>
<td>-50.332258</td>
<td>42.286879</td>
<td>94.396774</td>
</tr>
</tbody>
</table>

Warning: To use this module, you will need a BigQuery account. See
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.

1.2.10 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.
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.

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

Numpy Usage

```
In [126]: np.ones_like(s)
Out[126]: array([1, 1, 1, 1])
```

```
In [127]: np.diff(s)
Out[127]: array([1, 1, 1])
```

```
In [128]: np.where(s>1,s,np.nan)
Out[128]: array([ nan, 2., 3., 4.])
```

Pandonic Usage

```
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)
* __getattribute__,__setattr__
* indexed_same,reindex_like,align,where,mask
* fillna,replace(Series replace is now consistent with DataFrame)
* filter (also added axis argument to selectively filter on a different axis)
* reindex,reindex_axis,take
* truncate (moved to become part of NDFrame)

- These are API changes which make Panel more consistent with DataFrame
  - swapaxes on a Panel with the same axes specified now return a copy
  - support attribute access for setting
  - filter supports the same API as the original DataFrame filter

- Reindex called with no arguments will now return a copy of the input object

- TimeSeries is now an alias for Series. the property is_time_series can be used to distinguish (if desired)

- Refactor of Sparse objects to use BlockManager
  - Created a new block type in internals, SparseBlock, which can hold multi-dtypes and is non-consolidatable. SparseSeries and SparseDataFrame now inherit more methods from there hierarchy (Series/DataFrame), and no longer inherit from SparseArray (which instead is the object of the SparseBlock)
  - Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
  - Operations on sparse structures within DataFrames should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
  - enable setitem on SparseSeries for boolean/integer/slices
  - SparsePanels implementation is unchanged (e.g. not using BlockManager, needs work)

- added ftypes method to Series/DataFrame, similar to dtypes, but indicates if the underlying is sparse/dense (as well as the dtype)

- All NDFrame objects can now use __finalize__() to specify various values to propagate to new objects from an existing one (e.g. name in Series will follow more automatically now)

- Internal type checking is now done via a suite of generated classes, allowing isinstance(value, klass) without having to directly import the klass, courtesy of @jtratner

- Bug in Series update where the parent frame is not updating its cache based on changes (GH4080) or types (GH3217), fillna (GH3386)

- Indexing with dtype conversions fixed (GH4463, GH4204)

- Refactor Series.reindex to core/generic.py (GH4604, GH4618), allow method= in reindexing on a Series to work

- Series.copy no longer accepts the order parameter and is now consistent with NDFrame copy
• Refactor rename methods to core/generic.py; fixes Series.rename 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

1.2.11 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.

1.3 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.

1.3.1 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.

```python
In [1]: p = DataFrame({'first': [4, 5, 8], 'second': [0, 0, 3]})

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

[3 rows x 2 columns]

In [3]: p % p
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      inf
1      1      inf
2      1   1.000000

[3 rows x 2 columns]

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

[3 rows x 2 columns]
```
pandas: powerful Python data analysis toolkit, Release 0.13.1

• Add squeeze keyword to groupby to allow reduction from DataFrame -> Series if groups are unique. This
is a Regression from 0.10.1. We are reverting back to the prior behavior. This means groupby will return the
same shaped objects whether the groups are unique or not. Revert this issue (GH2893) with (GH3596).
In [6]: df2 = DataFrame([{"val1": 1, "val2" : 20}, {"val1":1, "val2": 19},
...:
{"val1":1, "val2": 27}, {"val1":1, "val2": 12}])
...:
In [7]: def func(dataf):
...:
return dataf["val2"]
...:

- dataf["val2"].mean()

# squeezing the result frame to a series (because we have unique groups)
In [8]: df2.groupby("val1", squeeze=True).apply(func)
Out[8]:
0
0.5
1
-0.5
2
7.5
3
-7.5
Name: 1, dtype: float64
# no squeezing (the default, and behavior in 0.10.1)
In [9]: df2.groupby("val1").apply(func)
Out[9]:
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 plently of alternatives. This preserves the iloc API to be purely positional
based.
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]

1.3. v0.12.0 (July 24, 2013)

35


# 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
• `read_html()` now defaults to `None` when reading, and falls back on `bs4 + html5lib` when `lxml` fails to parse. A list of parsers to try until success is also valid.

• The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)

• New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)

### 1.3.2 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

```python
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

• Added module for reading and writing Stata files: `pandas.io.stata` (GH1512) accessible via `read_stata` top-level function for reading, and `to_stata` DataFrame method for writing, See the docs

• Added module for reading and writing json format files: `pandas.io.json` accessible via `read_json` top-level function for reading, and `to_json` DataFrame method for writing, See the docs various issues (GH1226, GH3804, GH3876, GH3867, GH1305)

• MultiIndex column support for reading and writing csv format files
- The header option in `read_csv` now accepts a list of the rows from which to read the index.

- The option, `tupleize_cols` can now be specified in both `to_csv` and `read_csv`, to provide compatibility for the pre 0.12 behavior of writing and reading MultiIndex columns via a list of tuples. The default in 0.12 is to write lists of tuples and not interpret list of tuples as a MultiIndex column.

  Note: The default behavior in 0.12 remains unchanged from prior versions, but starting with 0.13, the default to write and read MultiIndex columns will be in the new format. (GH3571, GH1651, GH3141)

- If an `index_col` is not specified (e.g. you don’t have an index, or wrote it with `df.to_csv(..., index=False)`), then any names on the columns index will be lost.

```
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())
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>C_10_g0</td>
<td>C_10_g1</td>
</tr>
<tr>
<td>C1</td>
<td>C_11_g0</td>
<td>C_11_g1</td>
</tr>
<tr>
<td>C2</td>
<td>C_12_g0</td>
<td>C_12_g1</td>
</tr>
<tr>
<td>C3</td>
<td>C_13_g0</td>
<td>C_13_g1</td>
</tr>
</tbody>
</table>

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

```
Out[24]:
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>C_10_g0</td>
<td>C_10_g1</td>
</tr>
<tr>
<td>C1</td>
<td>C_11_g0</td>
<td>C_11_g1</td>
</tr>
<tr>
<td>C2</td>
<td>C_12_g0</td>
<td>C_12_g1</td>
</tr>
<tr>
<td>C3</td>
<td>C_13_g0</td>
<td>C_13_g1</td>
</tr>
</tbody>
</table>

```
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 |
```

- Support for HDFStore (via PyTables 3.0.0) on Python3
- Iterator support via `read_hdf` that automatically opens and closes the store when iteration is finished. This is only for `tables`

```
In [25]: path = 'store_iterator.h5'
In [26]: DataFrame(randn(10,2)).to_hdf(path,'df',table=True)
In [27]: for df in read_hdf(path,'df', chunksize=3):
    ....:     print(df)
    ....:
    0   1
    0  1.392665 -0.123497
```

---

38 Chapter 1. What’s New
• `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters.

### 1.3.3 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]:
  2  2
  3  3
  4  4
  5  5
  [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]:
  0   NaN
  1   NaN
  2  2
  3  3
  4  4
  [5 rows x 2 columns]
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)

### 1.3.4 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

# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [42]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers’ Day so let’s
# add that for a couple of years
In [43]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]

In [44]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [45]: dt = datetime(2013, 4, 30)

In [46]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00

In [47]: dts = date_range(dt, periods=5, freq=bday_egypt).to_series()

In [48]: 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
dtype: object
```

### 1.3.5 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 yields a Series with either a single character at each index of the original
  Series or NaN. For example,

```python
In [49]: strs = 'go', 'bow', 'joe', 'slow'
In [50]: ds = Series(strs)

In [51]: for s in ds.str:
   ....:     print(s)
   ....:
0  g
1  b
2  j
3  s
dtype: object

In [52]: s
Out[52]:
0   NaN
1   w
2   e
3   o
dtype: object

In [53]: s.dropna().values.item() == 'w'
Out[53]: 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.iteruples() now works with frames with duplicate column names (GH3873)
  – Bug in non-unique indexing via iloc (GH4017); added takeable argument to reindex for location-based taking
    – Allow non-unique indexing in series via .ix/.loc and __getitem__ (GH4246)
    – Fixed non-unique indexing memory allocation issue with .ix/.loc (GH4280)
• DataFrame.from_records did not accept empty recarrays (GH3682)
• read_html now correctly skips tests (GH3741)
• Fixed a bug where DataFrame.replace with a compiled regular expression in the to_replace argument
  wasn’t working (GH3907)
• Improved network test decorator to catch IOError (and therefore URLError as well). Added
  with_connectivity_check decorator to allow explicitly checking a website as a proxy for seeing if there
  is network connectivity. Plus, new optional_args decorator factory for decorators. (GH3910, GH3914)
• Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982,
  GH3985, GH4028, GH4054)
• Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed
  (GH3982, GH3985, GH4028, GH4054)
• Series.hist will now take the figure from the current environment if one is not passed
• Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
• Fixed running of tox under python3 where the pickle import was getting rewritten in an incompatible way
  (GH4062, GH4063)
• Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
• Fixed bug in DataFrame.replace where a nested dict wasn’t being iterated over when regex=False
  (GH4115)
• Fixed bug in the parsing of microseconds when using the format argument in to_datetime (GH4152)
• Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
• Fixed bug in plotting that wasn’t raising on invalid colormap for matplotlib 1.1.1 (GH4215)
• Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
• Fixed bug where Index slices weren’t carrying the name attribute (GH4226)
• Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
• Fixed bug where `html5lib` wasn’t being properly skipped (GH4265)
• Fixed bug where `get_data_famafrench` wasn’t using the correct file edges (GH4281)

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

1.4 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

1.4.1 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`, `Advanced Hierarchical` and `Fallback Indexing`

### 1.4.2 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.

### 1.4.3 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
```
1.044922 -1.685677 0
1.503906 -0.440747 0
-1.328125 -0.115070 1
1.024414 -0.632102 0
0.660156 -0.585977 0
1.236328 -1.444787 0
-2.169922 -0.201135 0

[8 rows x 3 columns]

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

# here you get some upcasting
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
In [8]: df3
Out[8]:
   A          B          C
0 -0.365356 -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

[8 rows x 3 columns]

In [9]: df3.dtypes
Out[9]:
A float32
B float64
C float64
dtype: object

1.4.4 Dtype Conversion

This is lower-common-denominator upcasting, meaning you get the dtype which can accomodate 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]: s = Series([datetime(2001,1,1,0,0), 'foo', 1.0, 1,
........:   Timestamp('20010104'), '20010105'],dtype='O')
........:

In [19]: s.convert_objects(convert_dates='coerce')
Out[19]:
         0  2001-01-01
         1   NaT
         2   NaT
         3   NaT
         4  2001-01-04
         5  2001-01-05
dtype: datetime64[ns]

1.4.5 Dtype Gotchas

Platform Gotchas

Starting in 0.11.0, construction of DataFrame/Series will use default dtypes of int 64 and float 64, 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 int 64 dtypes

In [20]: DataFrame([[1,2],columns=['a']]).dtypes
Out[20]:
a   int64
dtype: object

1.4. v0.11.0 (April 22, 2013)
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 [23]: dfi = df3.astype('int32')
In [24]: dfi['D'] = dfi['D'].astype('int64')
In [25]: dfi
Out[25]:
   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 [26]: dfi.dtypes
Out[26]:
A    int32
B    int32
C    int32
D    int64
E    int32
dtype: object
```

```
In [27]: casted = dfi[dfi>0]
In [28]: casted
Out[28]:
   A  B  C   D  E
0  NaN NaN  255  1  1
1  1  NaN  NaN  1  1
2  2  NaN  NaN  1  1
3  NaN NaN  NaN  1  1
4  1  NaN  NaN  1  1
5  2  NaN  NaN  1  1
6  2  NaN  NaN  1  1
7  NaN NaN  NaN  1  1
[8 rows x 5 columns]
```
In [29]: casted.dtypes
Out[29]:
A  float64
B  float64
C  float64
D  int64
E  int32
dtype: object

While float dtypes are unchanged.

In [30]: df4 = df3.copy()

In [31]: df4[‘A’] = df4[‘A’].astype(‘float32’)

In [32]: df4.dtypes
Out[32]:
A  float32
B  float64
C  float64
D  float16
E  int32
dtype: object

In [33]: casted = df4[df4>0]

In [34]: casted
Out[34]:
   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 [35]: casted.dtypes
Out[35]:
A  float32
B  float64
C  float64
D  float16
E  int32
dtype: object

1.4.6 Datetimes Conversion

Datetime64[ns] columns in a DataFrame (or a Series) allow the use of np.nan to indicate a nan value, in addition to the traditional NaT, or not-a-time. This allows convenient nan setting in a generic way. Furthermore datetime64[ns] columns are created by default, when passed datetimelike objects (this change was introduced in 0.10.1) (GH2809, GH2810)
In [36]: df = DataFrame(randn(6, 2), date_range('20010102', periods=6), columns=['A', 'B'])

In [37]: df['timestamp'] = Timestamp('20010103')

In [38]: df
Out[38]:
   A   B     timestamp
0 -1.448835  0.153437 2001-01-03
1 -1.123570 -0.791498 2001-01-03
2  0.105400  1.262401 2001-01-03
3 -0.721844 -0.647645 2001-01-03
4 -0.830631  0.761823 2001-01-03
5  0.597819  1.045558 2001-01-03

[6 rows x 3 columns]

# datetime64[ns] out of the box
In [39]: df.get_dtype_counts()
Out[39]:
datetime64[ns]    1
float64           2
dtype: int64

# use the traditional nan, which is mapped to NaT internally
In [40]: df.ix[2:4, ['A', 'timestamp']] = np.nan

In [41]: df
Out[41]:
   A   B     timestamp
0 -1.448835  0.153437 2001-01-03
1 -1.123570 -0.791498 2001-01-03
2     NaN      1.262401   NaT
3     NaN   -0.647645   NaT
4 -0.830631  0.761823 2001-01-03
5  0.597819  1.045558 2001-01-03

[6 rows x 3 columns]

Astype conversion on datetime64[ns] to object, implicitly converts NaT to np.nan

In [42]: import datetime

In [43]: s = Series([datetime.datetime(2001, 1, 2, 0, 0) for i in range(3)])

In [44]: s.dtype
Out[44]: dtype('<M8[ns]')

In [45]: s[1] = np.nan

In [46]: s
Out[46]:
0  2001-01-02
1   NaT
2  2001-01-02
dtype: datetime64[ns]

In [47]: s.dtype
Out[47]: dtype('<M8[ns]')
1.4.7 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

1.4.8 Enhancements

- Improved performance of `df.to_csv()` by up to 10x in some cases. (GH3059)
- Numexpr is now a Recommended Dependencies, to accelerate certain types of numerical and boolean operations
- Bottleneck is now a Recommended Dependencies, to accelerate certain types of `nan` operations
- HDFStore
  - support `read_hdf/to_hdf` API similar to `read_csv/to_csv`
    
    ```python
    In [51]: df = DataFrame(dict(A=lrange(5), B=lrange(5)))
    In [52]: df.to_hdf('store.h5','table',append=True)
    In [53]: read_hdf('store.h5', 'table', where = ['index>2'])
    Out[53]:
    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 [54]: idx = date_range("2001-10-1", periods=5, freq='M')

In [55]: ts = Series(np.random.rand(len(idx)),index=idx)

In [56]: ts['2001']
Out[56]:
2001-10-31 0.483450
2001-11-30 0.407530
2001-12-31 0.965096
Freq: M, dtype: float64

In [57]: df = DataFrame(dict(A = ts))

In [58]: df['2001']
Out[58]:
    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 [59]: p = Panel(randn(3,4,4),items=['ItemA','ItemB','ItemC'],
   ....:     major_axis=date_range('20010102',periods=4),
   ....:     minor_axis=['A','B','C','D'])

In [60]: p
Out[60]:
<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 [61]: p.reindex(items=['ItemA']).squeeze()
Out[61]:
   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 [62]: p.reindex(items=['ItemA'],minor=['B']).squeeze()
Out[62]:
   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.

### 1.5 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.

#### 1.5.1 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)
1.5.2 New features

- MySQL support for database (contribution from Dan Allan)

1.5.3 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
0  2000-01-01 -1.601262 -0.256718       foo         cool
1  2000-01-02  0.174122 -1.131794       foo         cool
2  2000-01-03  0.980347 -0.674429       foo         cool
3  2000-01-04 -0.761218  1.768215       foo         cool
4  2000-01-05 -0.862613 -0.210968  0.152288       foo         cool
5  2000-01-06  1.498195  0.462413  0.647604       foo         cool
6  2000-01-07  1.511487 -0.727189 -0.342928       foo         cool
7  2000-01-08 -0.007364  1.427674  0.104020       bar         cool
[8 rows x 5 columns]

# on-disk operations
In [8]: store.append('df', df, data_columns = ['B','C','string','string2'])

In [9]: store.select('df', ['B > 0', 'string == foo'])
Out[9]:
     A     B     C     string  string2
0  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
In [10]: df[(df.B > 0) & (df.string == 'foo')]
Out[10]:
     A     B     C     string  string2
0  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.
In [11]: store.unique('df','index')

Out[11]:
array(['2000-01-01T02:00:00.000000000+0200',
      '2000-01-02T02:00:00.000000000+0200',
      '2000-01-03T02:00:00.000000000+0200',
      '2000-01-04T02:00:00.000000000+0200',
      '2000-01-05T02:00:00.000000000+0200',
      '2000-01-06T02:00:00.000000000+0200',
      '2000-01-07T02:00:00.000000000+0200',
      '2000-01-08T02:00:00.000000000+0200'], dtype='datetime64[ns]')

In [12]: store.unique('df','string')

Out[12]: array(['foo', nan, 'bar'], dtype=object)

You can now store datetime64 in data columns

In [13]: df_mixed = df.copy()

In [14]: df_mixed['datetime64'] = Timestamp('20010102')

In [15]: df_mixed.ix[3:4, ['A','B']] = np.nan

In [16]: store.append('df_mixed', df_mixed)

In [17]: df_mixed1 = store.select('df_mixed')

In [18]: df_mixed1

Out[18]:
A       B       C  string  string2  datetime64
2000-01-01 -1.601262 -0.256718 0.239369   foo    cool  2001-01-02
2000-01-02  0.174122 -1.131794 -1.948006   foo    cool  2001-01-02
2000-01-03  0.980347 -0.674429 -0.361633   foo    cool  2001-01-02
2000-01-04   NaN     NaN  0.152288   NaN    NaN  2001-01-02
2000-01-05 -0.862613 -0.210968 -0.859278   NaN    NaN  2001-01-02
2000-01-06  1.498195  0.462413 -0.647604   NaN    NaN  2001-01-02
2000-01-07  1.511487 -0.727189 -0.342928   foo    cool  2001-01-02
2000-01-08 -0.007364  1.427674  0.104020   bar    cool  2001-01-02

[8 rows x 6 columns]

In [19]: df_mixed1.get_dtype_counts()

Out[19]:
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)

In [20]: store.select('df',columns = ['A','B'])

Out[20]:
A       B
2000-01-01 -1.601262 -0.256718
2000-01-02  0.174122 -1.131794
2000-01-03  0.980347 -0.674429
2000-01-04 -0.761218  1.768215
2000-01-05 -0.862613 -0.210968
2000-01-06  1.498195  0.462413
HDFStore now serializes multi-index dataframes when appending tables.

```
In [21]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
                          ['one', 'two', 'three'],
                          [0, 1, 2, 0, 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'])

In [22]: df = DataFrame(np.random.randn(10, 3), index=index,
                            columns=['A', 'B', 'C'])
```

```
In [23]: df
Out[23]:
      A       B       C
foo one  2.052171 -1.230963 -0.019240
two     -1.713238  0.838912  0.637855
      0.215109 -1.515362  1.586924
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
      1.060780  1.673018  1.249874
three   1.060780  1.673018  1.249874
qux one  1.458210 -0.710542  0.825392
two     1.557329  1.993441 -0.616293
      0.150468  0.132104  0.580923
```

```
In [24]: store.append('mi', df)
```

```
In [25]: store.select('mi')
Out[25]:
      A       B       C
foo one  2.052171 -1.230963 -0.019240
two     -1.713238  0.838912  0.637855
      0.215109 -1.515362  1.586924
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
      1.060780  1.673018  1.249874
three   1.060780  1.673018  1.249874
qux one  1.458210 -0.710542  0.825392
two     1.557329  1.993441 -0.616293
      0.150468  0.132104  0.580923
```

```
# the levels are automatically included as data columns
In [26]: store.select('mi', Term('foo=bar'))
Out[26]:
      A       B       C
foo one  2.052171 -1.230963 -0.019240
two     -1.713238  0.838912  0.637855
      0.215109 -1.515362  1.586924
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
      1.060780  1.673018  1.249874
three   1.060780  1.673018  1.249874
qux one  1.458210 -0.710542  0.825392
two     1.557329  1.993441 -0.616293
      0.150468  0.132104  0.580923
```

56 Chapter 1. What’s New
bar one -0.447974 -1.573998 0.630925   
two -0.071659 -1.277640 -0.102206
   [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 [27]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
                             columns=['A', 'B', 'C', 'D', 'E', 'F'])

In [28]: df_mt['foo'] = 'bar'

# you can also create the tables individually
In [29]: store.append_to_multiple({ 'df1_mt' : ['A','B'], 'df2_mt' : None }, df_mt, selector = 'df1_mt')

In [30]: store
Out[30]:
   <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
   /df                  frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,strings])
   /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 [31]: store.select('df1_mt')
Out[31]:
        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 [32]: store.select('df2_mt')
Out[32]:
         C     D     E     F    foo
2000-01-01 0.1431163 0.036240 0.9904587 -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 0.090653  0.901110 -0.096145  1.717830 bar
2000-01-05 0.134024  0.205969  1.348944 -1.198246 bar
2000-01-06 0.059493  0.461111 -1.565401  0.025706 bar
2000-01-07 0.302752  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 [33]: store.select_as_multiple([ 'df1_mt', 'df2_mt' ], where = [ 'A>0', 'B>0' ], selector = 'df1_mt')

1.5. v0.10.1 (January 22, 2013)
Out[33]:

<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>2000-01-04</td>
<td>1.298390</td>
<td>1.662964</td>
<td>-0.040863</td>
<td>0.290110</td>
<td>-0.096145</td>
<td>1.717830</td>
<td>bar</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.942864</td>
<td>2.502156</td>
<td>-0.302741</td>
<td>0.261551</td>
<td>-0.066342</td>
<td>0.897097</td>
<td>bar</td>
</tr>
</tbody>
</table>

[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 indices on the `indexables` and `data columns` of the table
- You can pass `chunksize=an integer` to append, to change the writing chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=an integer` to the first append, to set the TOTAL number of expectedrows that PyTables will 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.

1.6 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.

1.6.1 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

1.6.2 API changes

Deprecated DataFrame BINOP TimeSeries special case behavior

The default behavior of binary operations between a DataFrame and a Series has always been to align on the
DataFrame’s columns and broadcast down the rows, except in the special case that the DataFrame contains time
series. Since there are now method for each binary operator enabling you to specify how you want to broadcast, we
are phasing out this special case (Zen of Python: Special cases aren’t special enough to break the rules). Here’s what
I’m talking about:

In [1]: import pandas as pd

In [2]: df = pd.DataFrame(np.random.randn(6, 4),
             ...:     index=pd.date_range(’1/1/2000’, periods=6))
...:

```
In [3]: df
Out[3]:
   0     1     2     3
2000-01-01 -0.892402 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
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
...
2000-01-04 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, Length: 25

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: int64

# 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: int64

• 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 unnec-
essary.

• 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
1  3    No  4

[2 rows x 3 columns]

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

Series.apply will now operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame

```
In [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
[5 rows x 2 columns]
```

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:

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

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

### 1.6.3 New features

#### 1.6.4 Wide DataFrame Printing

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

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

In [36]: wide_frame
Out[36]:
```
The old behavior of printing out summary information can be achieved via the `expand_frame_repr` print option:

```python
In [37]: pd.set_option('expand_frame_repr', False)
In [38]: wide_frame
Out[38]:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
0  2.520045 1.570114 -0.360875 -0.880096 0.235532 0.207232 -1.983857 -1.702547 -1.621234 -0.906840 1.014601 -0.475108 -0.358944 1.262942 -0.412451 -0.462580
1  0.422194 0.288403 -0.487393 -0.777639 0.055865 1.383381 0.158399 0.246392 0.965887 0.246354 -0.727728 -0.094414 -0.276854 -1.642511 0.432560
2  0.585174 -0.564705 -0.719412 1.191340 -0.456362 0.089931 0.776079 0.752889 -1.195795 -1.425911 -0.548829 0.774225 0.740501 1.510263 -0.369325 -1.502617
3  1.218080 -0.564705 -0.581790 0.286071 0.048725 1.002440 1.276582 0.054399 0.241963 -0.471786 0.314510 -0.059986 -2.069319 -1.115104 -0.341265 1.844536
4 -0.376280 0.511936 -0.116412 -0.625256 -0.550627 1.261433 -0.552429 1.695803 -1.025917 -0.910942 0.426805 -0.131749 0.432600 0.044671 -0.341265 1.844536
```

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

```python
In [39]: pd.set_option('line_width', 40)
In [40]: wide_frame
Out[40]:
  0   1   2   3   4   5   6   7   8
0  2.520045 1.570114 -0.360875 -0.880096 0.235532 0.207232 -1.983857 -1.702547
1  0.422194 0.288403 -0.487393 -0.777639 0.055865 1.383381 0.158399 0.246392
2  0.585174 -0.564705 -0.719412 1.191340 -0.456362 0.089931 0.776079 0.752889
3  1.218080 -0.564705 -0.581790 0.286071 0.048725 1.002440 1.276582 0.054399
4 -0.376280 0.511936 -0.116412 -0.625256 -0.550627 1.261433 -0.552429 1.695803
```

[5 rows x 16 columns]
1.6.5 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]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-2.036047</td>
<td>0.000830</td>
<td>-0.955697</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.898872</td>
<td>-0.725411</td>
<td>0.059904</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.449644</td>
<td>1.082900</td>
<td>-1.221265</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.361078</td>
<td>1.330704</td>
<td>0.855932</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-1.216718</td>
<td>1.488887</td>
<td>0.018993</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.877046</td>
<td>0.045976</td>
<td>0.437274</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.567182</td>
<td>-0.888657</td>
<td>-0.556383</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.655457</td>
<td>1.117949</td>
<td>-2.782376</td>
</tr>
</tbody>
</table>

[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  1.330704  0.855932
2000-01-04  0.361078  1.488887  0.018993
2000-01-05 -1.216718  1.117949 -2.782376
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]: 8

In [55]: store.select('wp')
Out[55]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to D

# deleting a store
In [56]: del store['df']

In [57]: store
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp wide_table (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])

Enhancements

• added ability to hierarchical keys

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
/wp wide_table (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])
/food/apple frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])

# 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
/wp wide_table (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah frame (shape->[8,3])

• 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
0 2000-01-01 -2.036047 0.000830 string 1
1 2000-01-02 -0.898872 -0.725411 string 1
pandas: powerful Python data analysis toolkit, Release 0.13.1

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.

1.6.6 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 [70]: p4d = Panel4D(randn(2, 2, 5, 4),
....:     labels=[‘Label1’,‘Label2’],
....:     items=['Item1', 'Item2'],
....:     major_axis=date_range(‘1/1/2000’, periods=5),

1.6. v0.10.0 (December 17, 2012)
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.

1.7 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.

1.7.1 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  0  1  1
1  0  1  1
2  0  0  1
3  1  1  0
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  NaN NaN NaN
1  NaN NaN NaN
2  1  2  3
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
[6 rows x 3 columns]

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
[6 rows x 3 columns]

- DataFrame has new where and mask methods to select values according to a given boolean mask (GH2109, GH2151)

      DataFrame currently supports slicing via a boolean vector the same length as the DataFrame (inside the [[]]). The returned DataFrame has the same number of columns as the original, but is sliced on its index.

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.

```
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

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

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  0.816335  1.514102  1.298184
4  0.089527  0.576687  0.737750

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  3.000000  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

DataFrame.mask is the inverse boolean operation of where.

In [17]: df.mask(df<=0)
Out[17]:
```
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)

```python
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)

### 1.7.2 API changes

- Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

```python
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)

In [23]: p = Period('2012')

In [24]: p.end_time
Out[24]: Timestamp('2012-12-31 23:59:59.999999999', tz=None)

• File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

In [25]: data = 'A,B,C
00001,001,5
00002,002,6'

In [26]: from cStringIO import StringIO

In [27]: read_csv(StringIO(data), converters={'A': lambda x: x.strip()})
Out [27]:
   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.

1.8 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.

1.8.1 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)

1.8.2 API changes

• The default column names when header=None and no columns names passed to functions like read_csv has changed to be more Pythonic and amenable to attribute access:
In [1]: from StringIO import StringIO

In [2]: data = '0,0,1
   ...: 1,1,0
   ...: 0,1,0'

In [3]: df = read_csv(StringIO(data), header=None)

In [4]: df
Out[4]:
          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 [5]: s1 = Series([1, 2, 3])

In [6]: s1
Out[6]:
   0  1
   1  2
   2  3
dtype: int64

In [7]: s2 = Series(s1, index=['foo', 'bar', 'baz'])

In [8]: s2
Out[8]:
   foo  NaN
   bar  NaN
   baz  NaN
dtype: float64

- Deprecated `day_of_year` API removed from PeriodIndex, use `dayofyear` (GH1723)
- Don’t modify NumPy suppress printoption to True at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by HDFStore (GH1834, GH1824)
- Legacy cruft removed: pandas.stats.misc.quantileTS
- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)
- Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)
- Setting parts of DataFrame/Panel using `ix` now aligns input Series/DataFrame (GH1630)
- `first` and `last` methods in `GroupBy` no longer drop non-numeric columns (GH1809)
- Resolved inconsistencies in specifying custom NA values in text parser. `na_values` of type dict no longer override default NAs unless `keep_default_na` is set to false explicitly (GH1657)
- `DataFrame.dot` will not do data alignment, and also work with Series (GH1915)
See the full release notes or issue tracker on GitHub for a complete list.

1.9  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.

1.9.1 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)

1.9.2 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

1.10  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.

1.10.1 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)
1.10.2 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.

1.10.3 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

1.10.4 Other new features

• New cut and qcut functions (like R’s cut function) for computing a categorical variable from a continuous variable by binning values either into value-based (cut) or quantile-based (qcut) bins

• Rename Factor to Categorical and add a number of usability features

• Add limit argument to fillna/reindex

• More flexible multiple function application in GroupBy, and can pass list (name, function) tuples to get result in particular order with given names

• Add flexible replace method for efficiently substituting values

• Enhanced read_csv/read_table for reading time series data and converting multiple columns to dates

• Add comments option to parser functions: read_csv, etc.

• Add :ref:`dayfirst <io.dayfirst>` option to parser functions for parsing international DD/MM/YYYY dates

• Allow the user to specify the CSV reader dialect to control quoting etc.

• Handling thousands separators in read_csv to improve integer parsing.

• Enable unstacking of multiple levels in one shot. Alleviate pivot_table bugs (empty columns being introduced)

• Move to klib-based hash tables for indexing; better performance and less memory usage than Python’s dict

• Add first, last, min, max, and prod optimized GroupBy functions

• New ordered_merge function

• Add flexible comparison instance methods eq, ne, lt, gt, etc. to DataFrame, Series

• Improve scatter_matrix plotting function and add histogram or kernel density estimates to diagonal

• Add ‘kde’ plot option for density plots

• Support for converting DataFrame to R data.frame through rpy2

• Improved support for complex numbers in Series and DataFrame

• Add pct_change method to all data structures

• Add max_colwidth configuration option for DataFrame console output

• Interpolate Series values using index values

• Can select multiple columns from GroupBy
• Add `update` methods to Series/DataFrame for updating values in place
• Add `any` and `all` method to DataFrame

### 1.10.5 New plotting methods

`Series.plot` now supports a secondary_y option:

In [1]: plt.figure()
Out[1]: <matplotlib.figure.Figure at 0x144eafdd0>

In [2]: fx['FR'].plot(style='g')
Out[2]: <matplotlib.axes.AxesSubplot at 0x144ea050>

In [3]: fx['IT'].plot(style='k--', secondary_y=True)
Out[3]: <matplotlib.axes.AxesSubplot at 0x98d3890>

Vytuutas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, `'kde'` is a new option:

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 0x12fa4250>

In [6]: s.hist(normed=True, alpha=0.2)
Out[6]: <matplotlib.axes.AxesSubplot at 0x144ca8d0>

In [7]: s.plot(kind='kde')
Out[7]: <matplotlib.axes.AxesSubplot at 0x144ca8d0>
1.10.6 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.

1.10.7 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', tz=None)

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
```

See the plotting page for much more.
pandas’s Timestamp object is a subclass of datetime.datetime that has nanosecond support (the nanosecond field store the nanosecond value between 0 and 999). It should substitute directly into any code that used datetime.datetime values before. Thus, I recommend not casting DatetimeIndex to regular NumPy arrays.

If you have code that requires an array of datetime.datetime objects, you have a couple of options. First, the asobject property of DatetimeIndex produces an array of Timestamp objects:

```python
In [15]: stamp_array = rng.asobject

In [16]: stamp_array
```

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', tz=None)

To get an array of proper datetime.datetime objects, use the to_pydatetime method:

```python
In [18]: dt_array = rng.to_pydatetime()

In [19]: dt_array
```

Out[19]:

```
array([datetime.datetime(2000, 1, 1, 0, 0),
       datetime.datetime(2000, 1, 2, 0, 0),
       datetime.datetime(2000, 1, 3, 0, 0),
       datetime.datetime(2000, 1, 4, 0, 0),
       datetime.datetime(2000, 1, 5, 0, 0),
       datetime.datetime(2000, 1, 6, 0, 0),
       datetime.datetime(2000, 1, 7, 0, 0),
       datetime.datetime(2000, 1, 8, 0, 0),
       datetime.datetime(2000, 1, 9, 0, 0),
       datetime.datetime(2000, 1, 10, 0, 0)], dtype=object)
```

```
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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01, ..., 2000-01-10]
Length: 10, Freq: D, Timezone: None

In [23]: np.asarray(rng)
Out[23]:
array(['2000-01-01T02:00:00.000000000+0200',
      '2000-01-02T02:00:00.000000000+0200',
      '2000-01-03T02:00:00.000000000+0200',
      '2000-01-04T02:00:00.000000000+0200',
      '2000-01-05T02:00:00.000000000+0200',
      '2000-01-06T02:00:00.000000000+0200',
      '2000-01-07T02:00:00.000000000+0200',
      '2000-01-08T02:00:00.000000000+0200',
      '2000-01-09T02:00:00.000000000+0200',
      '2000-01-10T02:00:00.000000000+0200'], 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.

1.11 v.0.7.3 (April 12, 2012)

This is a minor release from 0.7.2 and fixes many minor bugs and adds a number of nice new features. There are also a couple of API changes to note; these should not affect very many users, and we are inclined to call them “bug fixes” even though they do constitute a change in behavior. See the full release notes or issue tracker on GitHub for a complete list.

1.11.1 New features

- New fixed width file reader, read_fwf
- New scatter_matrix function for making a scatter plot matrix

from pandas.tools.plotting import scatter_matrix
scatter_matrix(df, alpha=0.2)
• Add `stacked` argument to `Series` and `DataFrame`'s `plot` method for stacked bar plots.

```
df.plot(kind='bar', stacked=True)
```
df.plot(kind='barh', 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

1.11.2 NA Boolean Comparison API Change

Reverted some changes to how NA values (represented typically as NaN or None) are handled in non-numeric Series:

In [1]: series = Series(['Steve', np.nan, 'Joe'])
In [2]: series == 'Steve'
Out[2]:
0   True
1   False
2   False
dtype: bool

In [3]: series != 'Steve'
Out[3]:
0   False
1   True
2   True
dtype: bool

In comparisons, NA / NaN will always come through as False except with != which is True. Be very careful with boolean arithmetic, especially negation, in the presence of NA data. You may wish to add an explicit NA filter into boolean array operations if you are worried about this:

In [4]: mask = series == 'Steve'
In [5]: series[mask & series.notnull()]
Out[5]:
0   Steve
dtype: object

While propagating NA in comparisons may seem like the right behavior to some users (and you could argue on purely technical grounds that this is the right thing to do), the evaluation was made that propagating NA everywhere, including
in numerical arrays, would cause a large amount of problems for users. Thus, a “practicality beats purity” approach was taken. This issue may be revisited at some point in the future.

1.11.3 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]:
     A     B         C         D
0  foo  one  0.144909  1.387310
1  bar  one -1.033812  0.063490
2  foo  two  0.197333  1.437656
3  bar  three -0.059730 -0.814844
4  foo  two  0.087205 -0.482060
5  bar  two -1.607906  1.521442
6  foo  one -1.275249  0.882182
7  foo  three -0.054460 -0.108020

[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%  0.087205
       75%  0.144909
       max  0.197333
Length: 16, 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
```
1.12 v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

1.12.1 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)

1.12.2 Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)

1.13 v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

1.13.1 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

1.13.2 Performance improvements

- Improve performance and memory usage of fillna on DataFrame
- Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH787)
1.14 v.0.7.0 (February 9, 2012)

1.14.1 New features

- New unified **merge function** for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains (GH220, GH249, GH267)

- New **unified concatenation function** for concatenating Series, DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of Series.append and DataFrame.append (GH468, GH479, GH273)

- **Can** pass multiple DataFrames to DataFrame.append to concatenate (stack) and multiple Series to Series.append too

- **Can** pass list of dicts (e.g., a list of JSON objects) to DataFrame constructor (GH526)

- You can now **set multiple columns** in a DataFrame via __getitem__, useful for transformation (GH342)

- Handle differently-indexed output values in DataFrame.apply (GH498)

```python
In [1]: df = DataFrame(randn(10, 4))
In [2]: df.apply(lambda x: x.describe())
Out[2]:
   0      1      2      3
count 10.000000 10.000000 10.000000 10.000000
mean  0.119046  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.

### 1.14.2 API Changes to integer indexing

One of the potentially riskiest API changes in 0.7.0, but also one of the most important, was a complete review of how integer indexes are handled with regard to label-based indexing. Here is an example:

```python
In [3]: s = Series(randn(10), index=range(0, 20, 2))
In [4]: s
Out[4]:
0   -0.392051
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
```

```python
In [5]: s[0]
Out[5]: -0.39205110783730279
```

```python
In [6]: s[2]
Out[6]: -0.18953739573269102
```

```python
In [7]: s[4]
Out[7]: 0.88617008348573789
```

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  0.8843 -0.3363  0.1787  0.0316
2  0.1445 -0.1415  0.2504  0.5837
4  1.4478 -0.9186  1.4996  0.2716
6 -0.2659 -2.4184 -0.2658  0.1150
8 -0.5878  0.3144  0.8566  0.6194
10 0.1094  0.7175  1.0108  0.4799
12 1.1692 -0.3087  0.6049  0.4354
14 0.0734  0.3410  0.0424  0.1603

In [5]: df.ix[3]
KeyError: 3

In order to support purely integer-based indexing, the following methods have been added:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.iget_value(i)</td>
<td>Retrieve value stored at location i</td>
</tr>
<tr>
<td>Series.iget(i)</td>
<td>Alias for iget_value</td>
</tr>
<tr>
<td>DataFrame.irow(i)</td>
<td>Retrieve the i-th row</td>
</tr>
<tr>
<td>DataFrame.icol(j)</td>
<td>Retrieve the j-th column</td>
</tr>
<tr>
<td>DataFrame.iget_value(i, j)</td>
<td>Retrieve the value at row i and column j</td>
</tr>
</tbody>
</table>

1.14.3 API tweaks regarding label-based slicing

Label-based slicing using ix now requires that the index be sorted (monotonic) unless both the start and endpoint are contained in the index:

In [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:

In [10]: s.ix['k':'e']
Out[10]:
k  2.160843
a  0.533532
e  2.371548
dtype: float64

But this is not:
In [12]: s.ix['b':'h']  
KeyError 'b'

If the index had been sorted, the “range selection” would have been possible:

In [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

In [13]: s2.ix['b':'h']
Out[13]:
c  0.562726  
e  -2.371548  
g  1.269713  
dtype: float64

### 1.14.4 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
dtype: float64

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use ix.

1.14.5 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)

1.14.6 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)

1.15 v.0.6.1 (December 13, 2011)

1.15.1 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
1.15.2 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)

1.16 v.0.6.0 (November 25, 2011)

1.16.1 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)
• *Implemented* 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)

### 1.16.2 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)

### 1.17 v.0.5.0 (October 24, 2011)

#### 1.17.1 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

### 1.17.2 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

### 1.18 v.0.4.3 through v0.4.1 (September 25 - October 9, 2011)

#### 1.18.1 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)

### 1.18.2 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
You have the option to install an official release or to build the development version. If you choose to install from source and are running Windows, you will have to ensure that you have a compatible C compiler (MinGW or Visual Studio) installed. How to install MinGW on Windows

### 2.1 Python version support

Officially Python 2.6 to 2.7 and Python 3.2+. Python 2.4 and Python 2.5 are no longer supported since the userbase has shrunk significantly. Continuing Python 2.4 and 2.5 support will require either monetary development support or someone contributing to the project to restore compatibility.

### 2.2 Binary installers

#### 2.2.1 All platforms

Stable installers available on [PyPI](https://pypi.org)

Preliminary builds and installers on the [Pandas download page](https://pandas.pydata.org/pandas-dev/Install.html).
### 2.2.2 Overview

<table>
<thead>
<tr>
<th>Platform</th>
<th>Distribution</th>
<th>Status</th>
<th>Download / Repository Link</th>
<th>Install method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>all</td>
<td>stable</td>
<td><em>All platforms</em></td>
<td><em>pip install pandas</em></td>
</tr>
<tr>
<td>Mac</td>
<td>all</td>
<td>stable</td>
<td><em>All platforms</em></td>
<td><em>pip install pandas</em></td>
</tr>
<tr>
<td>Linux</td>
<td>Debian</td>
<td>stable</td>
<td><em>official Debian repository</em></td>
<td><em>sudo apt-get install python-pandas</em></td>
</tr>
<tr>
<td>Linux</td>
<td>Debian &amp; Ubuntu</td>
<td>unstable (latest packages)</td>
<td>NeuroDebian</td>
<td><em>sudo apt-get install python-pandas</em></td>
</tr>
<tr>
<td>Linux</td>
<td>Ubuntu</td>
<td>stable</td>
<td><em>official Ubuntu repository</em></td>
<td><em>sudo apt-get install python-pandas</em></td>
</tr>
<tr>
<td>Linux</td>
<td>Ubuntu</td>
<td>unstable (daily builds)</td>
<td><em>PythonXY PPA; activate by: sudo add-apt-repository ppa:pythonxy/pythonxy-devel &amp; &amp; sudo apt-get update</em></td>
<td><em>sudo apt-get install python-pandas</em></td>
</tr>
<tr>
<td>Linux</td>
<td>OpenSuse &amp; Fedora</td>
<td>stable</td>
<td><em>OpenSuse Repository</em></td>
<td><em>zypper in python-pandas</em></td>
</tr>
</tbody>
</table>

### 2.3 Dependencies

- **NumPy**: 1.6.1 or higher
- **python-dateutil 1.5**
- **pytz**
  - Needed for time zone support

### 2.4 Recommended Dependencies

- **numexpr**: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups.
- **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.

### 2.5 Optional Dependencies

- **Cython**: Only necessary to build development version. Version 0.17.1 or higher.
• **SciPy**: miscellaneous statistical functions
• **PyTables**: necessary for HDF5-based storage
• **matplotlib**: for plotting
• **statsmodels**
  – Needed for parts of `pandas.stats`
• **openpyxl, xlrd/xlwtt**
  – openpyxl version 1.6.1 or higher
  – Needed for Excel I/O
• **XlsxWriter**
  – Alternative Excel writer.
• **boto**: necessary for Amazon S3 access.
• 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 bq Command Line Tool** *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.

**Warning:**
– if you install **BeautifulSoup4** you must install either **lxml** or **html5lib** or both. `read_html()` will **not** work with **only** **BeautifulSoup4** installed.
– You are highly encouraged to read *HTML reading gotchas*. It explains issues surrounding the installation and usage of the above three libraries
– You may need to install an older version of **BeautifulSoup4**:
  * Versions 4.2.1, 4.1.3 and 4.0.2 have been confirmed for 64 and 32-bit Ubuntu/Debian
– Additionally, if you’re using **Anaconda** you should definitely read *the gotchas about HTML parsing libraries*

**Note:**
– if you’re on a system with **apt-get** you can do
  ```bash
  sudo apt-get build-dep python-lxml
  ```
  to get the necessary dependencies for installation of **lxml**. This will prevent further headaches down the line.

**Note:** Without the optional dependencies, many useful features will not work. Hence, it is highly recommended that you install these. A packaged distribution like the Enthought Python Distribution may be worth considering.
2.6 Installing from source

**Note:** Installing from the git repository requires a recent installation of Cython as the cythonized C sources are no longer checked into source control. Released source distributions will contain the built C files. I recommend installing the latest Cython via `easy_install -U Cython`.

The source code is hosted at [http://github.com/pydata/pandas](http://github.com/pydata/pandas), it can be checked out using git and compiled / installed like so:

```bash
git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install
```

Make sure you have Cython installed when installing from the repository, rather then a tarball or pypi.

On Windows, I suggest installing the MinGW compiler suite following the directions linked to above. Once configured property, run the following on the command line:

```bash
python setup.py build --compiler=mingw32
python setup.py install
```

Note that you will not be able to import pandas if you open an interpreter in the source directory unless you build the C extensions in place:

```bash
python setup.py build_ext --inplace
```

The most recent version of MinGW (any installer dated after 2011-08-03) has removed the `-mno-cygwin` option but Distutils has not yet been updated to reflect that. Thus, you may run into an error like “unrecognized command line option `-mno-cygwin`”. Until the bug is fixed in Distutils, you may need to install a slightly older version of MinGW (2011-08-02 installer).

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
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.................................S..................................................
.........................S........................................................
....
Ran 818 tests in 21.631s
OK (SKIP=2)
```
FREQUENTLY ASKED QUESTIONS (FAQ)

3.1 How do I control the way my DataFrame is displayed?

Pandas users rely on a variety of environments for using pandas: scripts, terminal, IPython qtconsole/ notebook, (IDLE, spyder, etc'). Each environment has it’s own capabilities and limitations: HTML support, horizontal scrolling, auto-detection of width/height. To appropriately address all these environments, the display behavior is controlled by several options, which you’re encouraged to tweak to suit your setup.

As of 0.13, these are the relevant options, all under the `display` namespace, (e.g. `display.width`, etc.):

- `notebook_repr_html`: if True, IPython frontends with HTML support will display dataframes as HTML tables when possible.
- `large_repr` (default 'truncate'): when a `DataFrame` exceeds `max_columns` or `max_rows`, it can be displayed either as a truncated table or, with this set to ‘info’, as a short summary view.
- `max_columns` (default 20): max dataframe columns to display.
- `max_rows` (default 60): max dataframe rows display.
- `show_dimensions` (default True): controls the display of the row/col counts footer.

Two additional options only apply to displaying DataFrames in terminals, not to the HTML view:

- `expand_repr` (default True): when the frame width cannot fit within the screen, the output will be broken into multiple pages.
- `width`: width of display screen in characters, used to determine the width of lines when `expand_repr` is active. Setting this to None will trigger auto-detection of terminal width.

IPython users can use the IPython startup file to import pandas and set these options automatically when starting up.

3.2 Adding Features to your Pandas Installation

Pandas is a powerful tool and already has a plethora of data manipulation operations implemented, most of them are very fast as well. It’s very possible however that certain functionality that would make your life easier is missing. In that case you have several options:

1. Open an issue on Github, explain your need and the sort of functionality you would like to see implemented.
2. Fork the repo, Implement the functionality yourself and open a PR on Github.
3. Write a method that performs the operation you are interested in and Monkey-patch the pandas class as part of your IPython profile startup or PYTHONSTARTUP file.

For example, here is an example of adding an `just_foo_cols()` method to the dataframe class:

```python
import pandas as pd

def just_foo_cols(self):
    """Get a list of column names containing the string 'foo'
    """
    return [x for x in self.columns if 'foo' in x]

def just_foo_cols(self):
    """Get a list of column names containing the string 'foo'
    """
    return [x for x in self.columns if 'foo' in x]

def just_foo_cols(self):
    """Get a list of column names containing the string 'foo'
    """
    return [x for x in self.columns if 'foo' in x]

def just_foo_cols(self):
    """Get a list of column names containing the string 'foo'
    """
    return [x for x in self.columns if 'foo' in x]

pd.DataFrame.just_foo_cols = just_foo_cols # monkey-patch the DataFrame class
df = pd.DataFrame([list(range(4))], columns=['A','foo','foozball','bar'])
df.just_foo_cols()
del pd.DataFrame.just_foo_cols # you can also remove the new method
```

Monkey-patching is usually frowned upon because it makes your code less portable and can cause subtle bugs in some circumstances. Monkey-patching existing methods is usually a bad idea in that respect. When used with proper care, however, it’s a very useful tool to have.

### 3.3 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`:

```python
In [1]: pnow('D') # scikits.timeseries.now()
Out[1]: Period('2014-02-03', 'D')

In [2]: Period(year=2007, month=3, day=15, freq='D')
Out[2]: Period('2007-03-15', 'D')

In [3]: p = Period('1984Q3')

In [4]: p
Out[4]: Period('1984Q3', 'Q-DEC')
```
In [5]: p.asfreq('D', 'start')
Out[5]: Period('1984-07-01', 'D')

In [6]: p.asfreq('D', 'end')
Out[6]: Period('1984-09-30', 'D')

In [7]: (p + 3).asfreq('T') + 6 * 60 + 30
Out[7]: Period('1985-07-01 06:29', 'T')

In [8]: rng = period_range('1990', '2010', freq='A')

In [9]: rng
Out[9]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: A-DEC
[1990, ..., 2010]
length: 21

In [10]: rng.asfreq('B', 'end') - 3
Out[10]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: B
[1990-12-26, ..., 2010-12-28]
length: 21

3.3.1 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>Date</td>
<td>Period</td>
<td>A span of time, from yearly through to secondly</td>
</tr>
<tr>
<td>DateArray</td>
<td>PeriodIndex</td>
<td>An array of timespans</td>
</tr>
<tr>
<td>convert</td>
<td>resample</td>
<td>Frequency conversion in scikits.timeseries</td>
</tr>
<tr>
<td>convert_to_annual</td>
<td>pivot_annual</td>
<td>currently supports up to daily frequency, see GH736</td>
</tr>
</tbody>
</table>

3.3.2 Frequency conversion

Frequency conversion is implemented using the resample method on TimeSeries and DataFrame objects (multiple time series). resample also works on panels (3D). Here is some code that resamples daily data to montly:

In [11]: rng = period_range('Jan-2000', periods=50, freq='M')

In [12]: data = Series(np.random.randn(50), index=rng)

In [13]: data
Out[13]:
In [14]: data.resample('A', how=np.mean)
Out[14]:
2000 -0.394510
2001 -0.244628
2002 -0.221633
2003 -0.453773
2004  0.850481
Freq: A-DEC, dtype: float64

3.3.3 Plotting

Much of the plotting functionality of scikits.timeseries has been ported and adopted to pandas’s data structures. For example:

In [15]: rng = period_range('1987Q2', periods=10, freq='Q-DEC')

In [16]: data = Series(np.random.randn(10), index=rng)

In [17]: plt.figure(); data.plot()
Out[17]: <matplotlib.axes.AxesSubplot at 0x7af4850>
3.3.4 Converting to and from period format

Use the `to_timestamp` and `to_period` instance methods.

3.3.5 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.

3.3.6 Resampling with timestamps and periods

`resample` has a kind argument which allows you to resample time series with a DatetimeIndex to PeriodIndex:

```
In [18]: rng = date_range(‘1/1/2000’, periods=200, freq=’D’)

In [19]: data = Series(np.random.randn(200), index=rng)

In [20]: data[:10]
Out[20]:
2000-01-01   -0.076467
2000-01-02   -1.187678
2000-01-03    1.130127
2000-01-04   -1.436737
2000-01-05    1.413681
2000-01-06    1.607920
2000-01-07    1.024180
```
In [21]: data.index
Out[21]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01, ..., 2000-07-18]
Length: 200, Freq: D, Timezone: None

In [22]: data.resample('M', kind='period')
Out[22]:
2000-01 -0.175775
2000-02 0.094874
2000-03 0.124949
2000-04 0.066215
2000-05 -0.040364
2000-06 0.116263
2000-07 -0.263235
Freq: M, dtype: float64

Similarly, resampling from periods to timestamps is possible with an optional interval ('start' or 'end') convention:

In [23]: rng = period_range('Jan-2000', periods=50, freq='M')
In [24]: data = Series(np.random.randn(50), index=rng)
In [25]: resampled = data.resample('A', kind='timestamp', convention='end')
In [26]: resampled.index
Out[26]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-12-31, ..., 2004-12-31]
Length: 5, Freq: A-DEC, Timezone: None

3.4 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 [27]: x = np.array(list(range(10)), '>i4') # big endian
In [28]: newx = x.byteswap().newbyteorder() # force native byteorder
In [29]: s = Series(newx)

See the NumPy documentation on byte order for more details.
3.5 Visualizing Data in Qt applications

There is experimental support for visualizing DataFrames in PyQt4 and PySide applications. At the moment you can display and edit the values of the cells in the DataFrame. Qt will take care of displaying just the portion of the DataFrame that is currently visible and the edits will be immediately saved to the underlying DataFrame.

To demonstrate this we will create a simple PySide application that will switch between two editable DataFrames. For this we will use the DataFrameModel class that handles the access to the DataFrame, and the DataFrameWidget, which is just a thin layer around the QTableView.

```python
import numpy as np
import pandas as pd
from pandas.sandbox.qtpandas import DataFrameModel, DataFrameWidget
from PySide import QtGui, QtCore

# Or if you use PyQt4:
# from PyQt4 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')
        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)
```

3.5. Visualizing Data in Qt applications 107
if __name__ == '__main__':
    import sys

    # Initialize the application
    app = QtGui.QApplication(sys.argv)
    mw = MainWidget()
    mw.show()
    app.exec_()
PACKAGE OVERVIEW

pandas consists of the following things

- A set of labeled array data structures, the primary of which are Series/TimeSeries 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

4.1 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>1</td>
<td>Time-Series</td>
<td>Series with index containing datetimes</td>
</tr>
<tr>
<td>2</td>
<td>DataFrame</td>
<td>General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns</td>
</tr>
<tr>
<td>3</td>
<td>Panel</td>
<td>General 3D labeled, also size-mutable array</td>
</tr>
</tbody>
</table>

4.1.1 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
```

### 4.2 Mutability and copying of data

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

### 4.3 Getting Support

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

Longer discussions occur on the developer mailing list, and commercial support inquiries for Lambda Foundry should be sent to: support@lambdafoundry.com

### 4.4 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.

### 4.5 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.

### 4.6 License
pandas is distributed under a 3-clause ("Simplified" or "New") BSD license. Parts of NumPy, SciPy, numpydoc, bottleneck, which all have 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
=====================

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:

```
#------------------------------------------------------------------------------
# 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.
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*

Traditionally, we import as follows

In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

### 5.1 Object Creation

See the *Data Structure Intro section*

Creating a *Series* by passing a list of values, letting pandas create a default integer index

In [4]: s = pd.Series([1,3,5,np.nan,6,8])

In [5]: s
Out[5]:
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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01, ..., 2013-01-06]
Length: 6, Freq: D, Timezone: None

In [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))

In [9]: df
Out[9]:

```
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' : 'foo' })
```

```
In [11]: df2
Out[11]:
    A       B       C       D       E
0  1.0 2013-01-02  1.0 3.0     foo
1  1.0 2013-01-02  1.0 3.0     foo
2  1.0 2013-01-02  1.0 3.0     foo
3  1.0 2013-01-02  1.0 3.0     foo
```

Having specific dtypes

```
In [12]: df2.dtypes
Out[12]:
    A        B            C        D       E
dtype: float64, dtype: datetime64[ns], dtype: float32, dtype: int32, 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>
```

As you can see, the columns A, B, C, and D are automatically tab completed. E is there as well; the rest of the attributes have been truncated for brevity.

### 5.2 Viewing Data

See the Basics section

See the top & bottom rows of the frame

```
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
[5 rows x 4 columns]

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
[3 rows x 4 columns]

Display the index, columns, and the underlying numpy data

In [16]: df.index
Out[16]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01, ..., 2013-01-06]
Length: 6, Freq: D, Timezone: 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]:
A    B    C    D
count 6.000000 6.000000 6.000000 6.000000
mean  0.073711 -0.431125 -0.687758 -0.233103
std   0.843157  0.922818  0.779887  0.973118
min  -0.861849 -2.104569 -1.509059 -1.135632
25%  -0.611510 -0.600794 -1.368714 -1.076610
50%   0.022070 -0.228039 -0.767252 -0.386188
75%   0.658444  0.041933 -0.034326  0.461706
max   1.212112  0.567020  0.276232  1.071804
[8 rows x 4 columns]

Transposing your data

In [20]: df.T
Out[20]:
          A          B          C          D
2013-01-01  0.469112  1.212112 -0.861849  0.721555
2013-01-02  1.212112  0.469112 -0.282863  0.721555
2013-01-03 -0.861849  0.469112 -0.282863 -0.861849
2013-01-04  0.721555 -0.282863  1.704569 -0.424972
2013-01-05 -0.424972  0.721555 -0.282863  0.220700
2013-01-06 -0.673690  0.113648 -1.478427  0.524988

5.2. Viewing Data
pandas: powerful Python data analysis toolkit, Release 0.13.1

C  -1.509059  0.119209  -0.494929  -1.039575  0.276232  -1.478427
D  -1.135632  -1.044236  1.071804  0.271860  -1.087401  0.524988

[4 rows x 6 columns]

Sorting by an axis

In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:

D  C  B  A
2013-01-01  -1.135632  -1.509059  -0.282863  0.469112
2013-01-02   -1.044236  0.119209  -0.173215  1.212112
2013-01-03   1.071804  -0.494929  -2.104569  -0.861849
2013-01-04    0.271860  -1.039575  -0.706771  0.721555
2013-01-05  -1.087401  0.276232  0.567020  -0.424972
2013-01-06  0.524988  -1.478427  0.113648  -0.673690

[6 rows x 4 columns]

Sorting by values

In [22]: df.sort(columns='B')
Out[22]:

A  B  C  D
2013-01-03  -0.861849  -2.104569  -0.494929  1.071804
2013-01-04    0.721555  -0.706771  -1.039575  0.271860
2013-01-01    0.469112  -0.282863  -1.509059  -1.135632
2013-01-02    1.212112  -0.173215  -0.706771  0.721555
2013-01-06  -0.673690  0.113648  -1.478427  0.524988
2013-01-05  -0.424972  0.567020  0.276232  -1.087401

[6 rows x 4 columns]

5.3 Selection

Note: While standard Python / Numpy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, .at, .iat, .loc, .iloc and .ix.

See the Indexing section and below.

5.3.1 Getting

Selecting a single column, which yields a Series, equivalent to df.A

In [23]: df['A']
Out[23]:

2013-01-01  0.469112
2013-01-02  1.212112
2013-01-03  -0.861849
2013-01-04   0.721555
2013-01-05  -0.424972
2013-01-06  -0.673690
Freq: D, Name: A, dtype: float64
Selecting via [], which slices the rows.

```python
In [24]: df[0:3]
Out[24]:
    A   B   C   D
2013-01-01 0.469112 -0.282863 -1.509059 -1.135632
2013-01-02 1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
[3 rows x 4 columns]
```

```python
In [25]: df['20130102':'20130104']
Out[25]:
    A   B   C   D
2013-01-02 1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
[3 rows x 4 columns]
```

### 5.3.2 Selection by Label

See more in `Selection by Label`

For getting a cross section using a label

```python
In [26]: df.loc[dates[0]]
Out[26]:
    A    B
2013-01-01 0.469112
2013-01-02 1.212112
2013-01-03 -0.861849
2013-01-04 0.721555
2013-01-05 -0.424972
2013-01-06 -0.673690
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label

```python
In [27]: df.loc[:,['A','B']]
Out[27]:
     A    B
2013-01-01 0.469112
2013-01-02 1.212112
2013-01-03 -0.861849
2013-01-04 0.721555
2013-01-05 -0.424972
2013-01-06 -0.673690
[6 rows x 2 columns]
```

Showing label slicing, both endpoints are included

```python
In [28]: df.loc['20130102':'20130104',['A','B']]
Out[28]:
     A    B
2013-01-02 1.212112
2013-01-03 -0.861849
2013-01-04 0.721555
[3 rows x 2 columns]
```

### 5.3. Selection
Reduction in the dimensions of the returned object

```
In [29]: df.loc['20130102', ['A', 'B']]
Out[29]:
   A     B
0  1.212112 -0.173215
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value

```
In [30]: df.loc[dates[0], 'A']
Out[30]: 0.46911229990718628
```

For getting fast access to a scalar (equiv to the prior method)

```
In [31]: df.at[dates[0], 'A']
Out[31]: 0.46911229990718628
```

### 5.3.3 Selection by Position

See more in *Selection by Position*

Select via the position of the passed integers

```
In [32]: df.iloc[3]
Out[32]:
   A     B     C     D
0  0.721555 -0.706771 -1.039575  0.271860
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
          A     B
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
[2 rows x 2 columns]
```

By lists of integer position locations, similar to the numpy/python style

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
          A     C
   A C
0  1.212112  0.119209
1 -0.861849 -0.494929
2 -0.424972  0.276232
[3 rows x 2 columns]
```

For slicing rows explicitly

```
In [35]: df.iloc[1:3, :]
Out[35]:
   A     B     C     D
0  1.212112 -0.173215  0.119209 -1.044236
1  1.212112  0.119209 -1.044236
2  1.212112  0.119209 -1.044236
[3 rows x 4 columns]
```
For slicing columns explicitly

In [36]: df.iloc[:,1:3]
Out[36]:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>-0.282863</td>
<td>-1.509059</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.173215</td>
<td>0.119209</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-2.104569</td>
<td>-0.494929</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.706771</td>
<td>-1.039575</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>0.567020</td>
<td>0.276232</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>0.113648</td>
<td>-1.478427</td>
</tr>
</tbody>
</table>

[6 rows x 2 columns]

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

There is one significant departure from standard python/numpy slicing semantics. python/numpy allow slicing past the end of an array without an associated error.

# these are allowed in python/numpy.

In [39]: x = list('abcdef')
In [40]: x[4:10]
Out[40]: ['e', 'f']
In [41]: x[8:10]
Out[41]: []

Pandas will detect this and raise IndexError, rather than return an empty structure.

>>> df.iloc[:,8:10]
IndexError: out-of-bounds on slice (end)

5.3.4 Boolean Indexing

Using a single column’s values to select data.

In [42]: df[df.A > 0]
Out[42]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.469112</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>2013-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
</tbody>
</table>

[3 rows x 4 columns]

A where operation for getting.
In [43]: df[df > 0]
Out [43]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.469112</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>NaN</td>
<td>0.119209</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>1.071804</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.721555</td>
<td>NaN</td>
<td>NaN</td>
<td>0.271860</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>NaN</td>
<td>0.567020</td>
<td>0.276232</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>NaN</td>
<td>0.113648</td>
<td>NaN</td>
<td>0.524988</td>
</tr>
</tbody>
</table>

[6 rows x 4 columns]

5.3.5 Setting

Setting a new column automatically aligns the data by the indexes

In [44]: s1 = pd.Series([1,2,3,4,5,6],index=pd.date_range('20130102',periods=6))

In [45]: s1
Out [45]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>2</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>3</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>4</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>5</td>
</tr>
</tbody>
</table>

Freq: D, dtype: int64

In [46]: df['F'] = s1

Setting values by label

In [47]: df.at[dates[0],'A'] = 0

Setting values by position

In [48]: df.iat[0,1] = 0

Setting by assigning with a numpy array

In [49]: df.loc[::,'D'] = np.array([5] * len(df))

The result of the prior setting operations

In [50]: df
Out [50]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-1.509059</td>
<td>5</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

[6 rows x 5 columns]

A where operation with setting.
In [51]: df2 = df.copy()

In [52]: df2[df2 > 0] = -df2

In [53]: df2
Out[53]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-1.509059</td>
<td>-5</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-1.212112</td>
<td>-0.173215</td>
<td>-0.119209</td>
<td>-5</td>
<td>-1</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>-5</td>
<td>-3</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.424972</td>
<td>-0.567020</td>
<td>-0.276232</td>
<td>-5</td>
<td>-4</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-0.673690</td>
<td>-0.113648</td>
<td>-1.478427</td>
<td>-5</td>
<td>-5</td>
</tr>
</tbody>
</table>

[6 rows x 5 columns]

## 5.4 Missing Data

Pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the [Missing Data section](#).

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data.

In [54]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])

In [55]: df1.loc[dates[0]:dates[1],'E'] = 1

In [56]: df1
Out[56]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-1.509059</td>
<td>5</td>
<td>NaN</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>5</td>
<td>2</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>5</td>
<td>3</td>
<td>NaN</td>
</tr>
</tbody>
</table>

[4 rows x 6 columns]

To drop any rows that have missing data.

In [57]: df1.dropna(how='any')
Out[57]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

[1 rows x 6 columns]

Filling missing data

In [58]: df1.fillna(value=5)
Out[58]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-1.509059</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

[4 rows x 6 columns]

### 5.4. Missing Data
To get the boolean mask where values are `nan`

```
In [59]: pd.isnull(df1)
Out[59]:
        A     B     C     D     F     E
2013-01-01  False  False  False  False   True  False
2013-01-02  False  False  False  False  False  False
2013-01-03  False  False  False  False   True   True
2013-01-04  False  False  False  False   True   True
```

[4 rows x 6 columns]

### 5.5 Operations

See the *Basic section on Binary Ops*

#### 5.5.1 Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic

```
In [60]: df.mean()
Out[60]:
         A     B     C     D     F
2013-01-01 -0.004474 -0.383981 -0.687758 5.000000 3.000000
         dtype: float64
```

Same operation on the other axis

```
In [61]: df.mean(1)
Out[61]:
2013-01-01  0.872735
2013-01-02  1.431621
2013-01-03  0.707758
2013-01-04  1.395042
2013-01-05  1.883656
2013-01-06  1.592306
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension.

```
In [62]: s = pd.Series([1,3,5,np.nan,6,8],index=dates).shift(2)
```

```
In [63]: s
Out[63]:
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

In [64]: df.sub(s, axis='index')
Out[64]:
   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 rows x 5 columns]

5.5.2 Apply

Applying functions to the data

In [65]: df.apply(np.cumsum)
Out[65]:
   A    B    C    D    F
2013-01-01 0.000000 0.000000 -1.509059 5 NaN
2013-01-02 1.212112 -0.173215 -1.389850 10  1
2013-01-03 0.350263 -2.277784 -1.884779 15  3
2013-01-04 1.071818 -2.984555 -2.924354 20  6
2013-01-05 0.646846 -2.417535 -2.648122 25 10
2013-01-06 -0.026844 -2.303886 -4.126549 30 15

[6 rows x 5 columns]

In [66]: df.apply(lambda x: x.max() - x.min())
Out[66]:
   A    B    C    D    F
dtype: float64
   2.073961
   2.671590
   1.785291
   0.000000
   4.000000

5.5.3 Histogramming

See more at Histogramming and Discretization

In [67]: s = pd.Series(np.random.randint(0,7,size=10))

In [68]: s
Out[68]:
0  4
1  2
2  1
3  2
4  6
5  4
6  4
7  6

5.5. Operations
In [69]: s.value_counts()
Out[69]:
4   5
6   2
2   2
1   1
dtype: int64

### 5.5.4 String Methods

See more at Vectorized String Methods

In [70]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [71]: s.str.lower()
Out[71]:
0   a
1   b
2   c
3  aaba
4  baca
5   NaN
6   caba
7   dog
8   cat
dtype: object

### 5.6 Merge

#### 5.6.1 Concat

Pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the Merging section

Concatenating pandas objects together

In [72]: df = pd.DataFrame(np.random.randn(10, 4))

In [73]: df
Out[73]:
   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 -0.454267  0.406774 -0.116068 -0.771215
9  0.621286 -0.810538  0.153198 -1.205150
8 -0.575247 0.254161 -1.143704 0.215897
9 1.193555 -0.077118 -0.408530 -0.862495

[10 rows x 4 columns]

# break it into pieces
In [74]: pieces = [df[:3], df[3:7], df[7:]]

In [75]: pd.concat(pieces)
Out[75]:
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

[10 rows x 4 columns]

5.6.2 Join

SQL style merges. See the Database style joining

In [76]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [77]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [78]: left
Out[78]:
  key lval
0  foo  1
1  foo  2

[2 rows x 2 columns]

In [79]: right
Out[79]:
  key rval
0  foo  4
1  foo  5

[2 rows x 2 columns]

In [80]: pd.merge(left, right, on='key')
Out[80]:
  key lval rval
0  foo   1   4
1  foo   1   5
2  foo   2   4
3  foo   2   5

[4 rows x 3 columns]

5.6. Merge
5.6.3 Append

Append rows to a dataframe. See the `Appending`

```
In [81]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])
```
```
In [82]: df
Out[82]:
   A          B         C         D
0 1.346061  1.511763 -0.990582  0.024580
1 -0.441652  1.211526  0.268520  1.591431
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346 -0.339355  1.591431
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351 -0.708758 -0.264610
```
```
In [83]: s = df.iloc[3]
```
```
In [84]: df.append(s, ignore_index=True)
Out[84]:
   A          B         C         D
0 1.346061  1.511763 -0.990582  0.024580
1 -0.441652  1.211526  0.268520  1.591431
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346 -0.339355  1.591431
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351 -0.708758 -0.264610
8  1.453749  1.208843  0.024580  1.591431
```

5.7 Grouping

By “group by” we are referring to a process involving one or more of the following steps

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

See the `Grouping section`

```
In [85]: df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar', 'foo', 'foo'],
                      'B' : ['one', 'one', 'two', 'three',
                           'two', 'two', 'one', 'three'],
                      'C' : np.random.randn(8),
                      'D' : np.random.randn(8))
```
```
In [86]: df
```
Grouping and then applying a function `sum` to the resulting groups.

```
In [87]: df.groupby('A').sum()
Out[87]:
       C       D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958

[2 rows x 2 columns]
```

Grouping by multiple columns forms a hierarchical index, which we then apply the function.

```
In [88]: df.groupby(["A", "B"]).sum()
Out[88]:
       C       D
A B
bar one  1.814470  2.395985
      three -0.595447  0.166599
two -0.392670 -0.136473
foo one -1.195665 -0.616981
      three  1.928123 -1.623033
two  2.414034  1.600434

[6 rows x 2 columns]
```

## 5.8 Reshaping

See the section on `Hierarchical Indexing` and see the section on `Reshaping`.

### 5.8.1 Stack

```
In [89]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                           'foo', 'foo', 'qux', 'qux'],
                        ['one', 'two', 'one', 'two',
                           'one', 'two', 'one', 'two']])))

In [90]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [91]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])
```
In [92]: df2 = df[:4]

In [93]: df2
Out[93]:
   A     B
first  
   bar  one  0.029399 -0.542108
         two  0.282696 -0.087302
   baz  one -1.575170  1.771208
         two  0.816482  1.100230

[4 rows x 2 columns]

The stack function “compresses” a level in the DataFrame’s columns.

In [94]: stacked = df2.stack()

In [95]: stacked
Out[95]:
   A     B
first  
   bar  one  0.029399
         B -0.542108
         two  0.282696
              B -0.087302
   baz  one -1.575170
         B  1.771208
         two  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 [96]: stacked.unstack()
Out[96]:
   A     B
first  
   bar  one  0.029399 -0.542108
         two  0.282696 -0.087302
   baz  one -1.575170  1.771208
         two  0.816482  1.100230

[4 rows x 2 columns]

In [97]: stacked.unstack(1)
Out[97]:
   second
   one      two
first
   bar  A  0.029399  0.282696
         B -0.542108 -0.087302
   baz  A -1.575170  0.816482
         B  1.771208  1.100230

[4 rows x 2 columns]

In [98]: stacked.unstack(0)
Out[98]:
   first
   bar  
         baz  
   second
5.8.2 Pivot Tables

See the section on *Pivot Tables*.

```python
In [99]: 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 [100]: df
Out[100]:
     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      0.100900  -1.425638
6    two  A  foo      0.100900  -1.425638
7    three  B  foo     -1.035018   1.024098
8     one  C  foo      0.314665  -0.106062
9     one  A  bar      -0.773723   1.824375
10   two  B  bar     -1.170653   0.595974
11   three  C  bar      0.648740   1.167115
```

We can produce pivot tables from this data very easily:

```python
In [101]: pd.pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
Out[101]:
           C
        bar  foo
       A
    one  -0.773723  1.418757
    B   -0.029716 -1.879024
    C   -1.146178  0.314665
     three  1.006160   NaN
       B      NaN  -1.035018
       C  0.648740  NaN
   two     NaN  0.100900
      B -1.170653   NaN
      C  NaN  0.536826
```

5.8. Reshaping
5.9 Time Series

Pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the Time Series section.

In [102]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [103]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [104]: ts.resample('5Min', how='sum')
Out[104]:
2012-01-01  25083
Freq: 5T, dtype: int64

Time zone representation

In [105]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [106]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [107]: ts
Out[107]:
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 [108]: ts_utc = ts.tz_localize('UTC')

In [109]: ts_utc
Out[109]:
2012-03-06  00:00:00+00:00   0.464000
2012-03-07  00:00:00+00:00   0.227371
2012-03-08  00:00:00+00:00  -0.496922
2012-03-09  00:00:00+00:00   0.306389
2012-03-10  00:00:00+00:00  -2.290613
Freq: D, dtype: float64

Convert to another time zone

In [110]: ts_utc.tz_convert('US/Eastern')

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

Converting between time span representations

In [111]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [112]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [113]: ts
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 [117]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')
In [118]: ts = pd.Series(np.random.randn(len(prng)), prng)
In [119]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9
```

5.10 Plotting

Plotting docs.

```python
In [121]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [122]: ts = ts.cumsum()
In [123]: ts.plot()
```
On DataFrame, `plot` is a convenience to plot all of the columns with labels:

```python
In [124]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                         columns=['A', 'B', 'C', 'D'])
.....:
```  
```python
In [125]: df = df.cumsum()
```  
```python
In [126]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[126]: <matplotlib.legend.Legend at 0x6fc5910>
```
5.11 Getting Data In/Out

5.11.1 CSV

Writing to a csv file

In [127]: df.to_csv('foo.csv')

Reading from a csv file

In [128]: pd.read_csv('foo.csv')

Out [128]:

<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>
<tr>
<td>7</td>
<td>-1.318492</td>
<td>0.003142</td>
<td>-3.863379</td>
<td>-0.791151</td>
</tr>
<tr>
<td>8</td>
<td>-1.552842</td>
<td>1.292518</td>
<td>-4.772843</td>
<td>-0.471664</td>
</tr>
<tr>
<td>9</td>
<td>-1.621025</td>
<td>0.074253</td>
<td>-5.866093</td>
<td>-0.162509</td>
</tr>
<tr>
<td>10</td>
<td>-2.418239</td>
<td>-0.640980</td>
<td>-5.895733</td>
<td>-0.362802</td>
</tr>
<tr>
<td>11</td>
<td>-3.350633</td>
<td>-0.358341</td>
<td>-5.917620</td>
<td>-0.444849</td>
</tr>
<tr>
<td>12</td>
<td>-3.268737</td>
<td>-0.795976</td>
<td>-6.240176</td>
<td>0.497327</td>
</tr>
<tr>
<td>13</td>
<td>-2.786138</td>
<td>-1.017654</td>
<td>-4.260442</td>
<td>0.631441</td>
</tr>
<tr>
<td>14</td>
<td>-4.261365</td>
<td>-0.721180</td>
<td>-3.918269</td>
<td>-0.118960</td>
</tr>
</tbody>
</table>

... ... ... ... ...
5.11.2 HDF5

Reading and writing to **HDFStores**

Writing to a HDF5 Store

In [129]: df.to_hdf('foo.h5','df')

Reading from a HDF5 Store

In [130]: pd.read_hdf('foo.h5','df')

Out[130]:

<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>0.266457</td>
<td>-0.399641</td>
<td>-0.219582</td>
<td>1.186860</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-1.170732</td>
<td>-0.345873</td>
<td>1.653061</td>
<td>-0.282953</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-1.734933</td>
<td>0.530468</td>
<td>2.060811</td>
<td>-0.515536</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.555121</td>
<td>1.452620</td>
<td>0.239859</td>
<td>-1.156896</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.578117</td>
<td>0.511371</td>
<td>0.103552</td>
<td>-2.428202</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.478344</td>
<td>0.449933</td>
<td>-0.741620</td>
<td>-1.962409</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.235339</td>
<td>-0.091757</td>
<td>-1.543861</td>
<td>-1.084753</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.318492</td>
<td>0.003142</td>
<td>-3.863379</td>
<td>-0.791151</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-1.552842</td>
<td>1.292518</td>
<td>-4.772843</td>
<td>-0.471664</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-1.621025</td>
<td>0.074253</td>
<td>-5.866093</td>
<td>-0.162509</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>-2.418239</td>
<td>-0.640980</td>
<td>-5.895733</td>
<td>-0.362802</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-3.350633</td>
<td>-0.358341</td>
<td>-5.917620</td>
<td>-0.444849</td>
</tr>
<tr>
<td>2000-01-13</td>
<td>-3.268737</td>
<td>-0.795976</td>
<td>-6.240176</td>
<td>0.497327</td>
</tr>
<tr>
<td>2000-01-14</td>
<td>-2.786138</td>
<td>-1.017654</td>
<td>-4.260442</td>
<td>0.631441</td>
</tr>
<tr>
<td>2000-01-15</td>
<td>-4.261365</td>
<td>-0.721180</td>
<td>-3.918269</td>
<td>-0.118960</td>
</tr>
</tbody>
</table>

[1000 rows x 4 columns]

5.11.3 Excel

Reading and writing to **MS Excel**

Writing to an excel file

In [131]: df.to_excel('foo.xlsx', sheet_name='Sheet1')

Reading from an excel file

In [132]:pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])

Out[132]:

<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>0.266457</td>
<td>-0.399641</td>
<td>-0.219582</td>
<td>1.186860</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-1.170732</td>
<td>-0.345873</td>
<td>1.653061</td>
<td>-0.282953</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-1.734933</td>
<td>0.530468</td>
<td>2.060811</td>
<td>-0.515536</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.555121</td>
<td>1.452620</td>
<td>0.239859</td>
<td>-1.156896</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.578117</td>
<td>0.511371</td>
<td>0.103552</td>
<td>-2.428202</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.478344</td>
<td>0.449933</td>
<td>-0.741620</td>
<td>-1.962409</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.235339</td>
<td>-0.091757</td>
<td>-1.543861</td>
<td>-1.084753</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.318492</td>
<td>0.003142</td>
<td>-3.863379</td>
<td>-0.791151</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-1.552842</td>
<td>1.292518</td>
<td>-4.772843</td>
<td>-0.471664</td>
</tr>
</tbody>
</table>
2000-01-10 | -1.621025 | 0.074253 | -5.866093 | -0.162509  
2000-01-11 | -2.418239 | -0.640980 | -5.895733 | -0.362802  
2000-01-12 | -3.350633 | -0.358341 | -5.917620 | -0.444849  
2000-01-13 | -3.268737 | -0.795976 | -6.240176 | 0.497327    
2000-01-14 | -2.786138 | -1.017654 | -4.260442 | 0.631441    
2000-01-15 | -4.261365 | -0.721180 | -3.918269 | -0.118960 
... ... ... ...  
[1000 rows x 4 columns]

## 5.12 Gotchas

If you are trying an operation and you see an exception like:

```python
>>> if pd.Series([False, True, False]):
    print("I was true")
Traceback
...  
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *Comparisons* for an explanation and what to do.

See *Gotchas* as well.
This is a guide to many pandas tutorials, geared mainly for new users.

6.1 Internal Guides

Pandas own *10 Minutes to Pandas*

More complex recipes are in the *Cookbook*

6.2 Pandas Cookbook

The goal of this cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that that entails.

Here are links to the v0.1 release. For an up-to-date table of contents, see the pandas-cookbook GitHub repository.

- **A quick tour of the IPython Notebook**: Shows off IPython’s awesome tab completion and magic functions.
- **Chapter 1**: Reading your data into pandas is pretty much the easiest thing. Even when the encoding is wrong!
- **Chapter 2**: It’s not totally obvious how to select data from a pandas dataframe. Here we explain the basics (how to take slices and get columns)
- **Chapter 3**: Here we get into serious slicing and dicing and learn how to filter dataframes in complicated ways, really fast.
- **Chapter 4**: Groupby/aggregate is seriously my favorite thing about pandas and I use it all the time. You should probably read this.
- **Chapter 5**: Here you get to find out if it’s cold in Montreal in the winter (spoiler: yes). Web scraping with pandas is fun! Here we combine dataframes.
- **Chapter 6**: Strings with pandas are great. It has all these vectorized string operations and they’re the best. We will turn a bunch of strings containing “Snow” into vectors of numbers in a trice.
- **Chapter 7**: Cleaning up messy data is never a joy, but with pandas it’s easier.
- **Chapter 8**: Parsing Unix timestamps is confusing at first but it turns out to be really easy.
6.3 Lessons for New Pandas Users

For more resources, please visit the main repository.

• 01 - Lesson: - Importing libraries - Creating data sets - Creating data frames - Reading from CSV - Exporting to CSV - Finding maximums - Plotting data
• 02 - Lesson: - Reading from TXT - Exporting to TXT - Selecting top/bottom records - Descriptive statistics - Grouping/sorting data
• 03 - Lesson: - Creating functions - Reading from EXCEL - Exporting to EXCEL - Outliers - Lambda functions - Slice and dice data
• 04 - Lesson: - Adding/deleting columns - Index operations
• 05 - Lesson: - Stack/Unstack/Transpose functions
• 06 - Lesson: - GroupBy function
• 07 - Lesson: - Ways to calculate outliers
• 08 - Lesson: - Read from Microsoft SQL databases
• 09 - Lesson: - Export to CSV/EXCEL/TXT
• 10 - Lesson: - Converting between different kinds of formats
• 11 - Lesson: - Combining data from various sources

6.4 Excel charts with pandas, vincent and xlsxwriter

• Using Pandas and XlsxWriter to create Excel charts

6.5 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.

This is a great First Pull Request (to add interesting links and/or put short code inline for existing links)

### 7.1 Idioms

These are some neat pandas idioms

- How to do if-then-else?
- How to do if-then-else #2
- How to split a frame with a boolean criterion?
- How to select from a frame with complex criteria?
- Select rows closest to a user-defined number
- How to reduce a sequence (e.g. of Series) using a binary operator

### 7.2 Selection

The indexing docs.

- Indexing using both row labels and conditionals, see here
- Use loc for label-oriented slicing and iloc positional slicing, see here
- Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions, see here
- Mask a panel by using np.where and then reconstructing the panel with the new masked values here
- Using ~ to take the complement of a boolean array, see here
- Efficiently creating columns using applymap

### 7.3 MultiIndexing

The multindexing docs.
Creating a multi-index from a labeled frame

### 7.3.1 Arithmetic

Performing arithmetic with a multi-index that needs broadcasting

### 7.3.2 Slicing

Slicing a multi-index with xs
Slicing a multi-index with xs #2
Setting portions of a multi-index with xs

### 7.3.3 Sorting

Multi-index sorting
Partial Selection, the need for sortedness

### 7.3.4 Levels

Prepending a level to a multiindex
Flatten Hierarchical columns

### 7.3.5 panelnd

The *panelnd* docs.
Construct a 5D panelnd

### 7.4 Missing Data

The *missing data* docs.
Fill forward a reversed timeseries

```python
In [1]: df = pd.DataFrame(np.random.randn(6,1), index=pd.date_range('2013-08-01', periods=6, freq='B'), columns=list('A'))
In [2]: df.ix[3,'A'] = np.nan
In [3]: df
```

```
Out[3]:
          A
2013-08-01  0.469112
2013-08-02 -0.282863
2013-08-05 -1.509059
2013-08-06      NaN
2013-08-07  1.212112
2013-08-08 -0.173215
```
In [4]: df.reindex(df.index[::-1]).ffill()
Out[4]:
   A
2013-08-08 -0.173215
2013-08-07  1.212112
2013-08-06  1.212112
2013-08-05 -1.509059
2013-08-02 -0.282863
2013-08-01  0.469112

cumsum reset at NaN values

### 7.4.1 Replace

Using replace with backrefs

### 7.5 Grouping

The grouping docs.

Basic grouping with apply

Using get_group

Apply to different items in a group

Expanding Apply

Replacing values with groupby means

Sort by group with aggregation

Create multiple aggregated columns

Create a value counts column and reassign back to the DataFrame

### 7.5.1 Expanding Data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

### 7.5.2 Splitting

Splitting a frame
7.5.3 Pivot

The Pivot docs.
Partial sums and subtotals
Frequency table like plyr in R

7.5.4 Apply

Turning embedded lists into a multi-index frame
Rolling apply with a DataFrame returning a Series
Rolling apply with a DataFrame returning a Scalar

7.6 Timeseries

Between times
Using indexer between time
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?

7.6.1 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

7.7 Merge

The Concat docs. The Join docs.
emulate R rbind
Self Join
How to set the index and join
KDB like asof join
Join with a criteria based on the values
7.8 Plotting

The *Plotting* docs.

Make Matplotlib look like R
Setting x-axis major and minor labels
Plotting multiple charts in an ipython notebook
Creating a multi-line plot
Plotting a heatmap
Annotate a time-series plot
Annotate a time-series plot #2
Generate Embedded plots in excel files using Pandas, Vincent and xlsxwriter

7.9 Data In/Out

Performance comparison of SQL vs HDF5

7.9.1 CSV

The *CSV* docs

read_csv in action
appending to a csv
Reading a csv chunk-by-chunk
Reading only certain rows of a csv chunk-by-chunk
Reading the first few lines of a frame

Reading a file that is compressed but not by gzip/bz2 (the native compressed formats which read_csv understands). This example shows a WinZipped file, but is a general application of opening the file within a context manager and using that handle to read. See here

Inferring dtypes from a file
Dealing with bad lines
Dealing with bad lines II
Reading CSV with Unix timestamps and converting to local timezone
Write a multi-row index CSV without writing duplicates

7.9.2 SQL

The *SQL* docs

Reading from databases with SQL
7.9.3 Excel

The Excel docs

Reading from a filelike handle Reading HTML tables from a server that cannot handle the default request header

7.9.4 HDFStore

The HDFStore docs

Simple Queries with a Timestamp Index
Managing heterogeneous data using a linked multiple table hierarchy
Merging on-disk tables with millions of rows
Deduplicating 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
Counting with a HDFStore
Troubleshoot HDFStore exceptions
Setting min_itemsize with strings
Using ptrepack to create a completely-sorted-index on a store

Storing Attributes to a group node

In [5]: df = DataFrame(np.random.randn(8, 3))

In [6]: store = HDFStore('test.h5')

In [7]: store.put('df', df)

# you can store an arbitrary python object via pickle
In [8]: store.get_storer('df').attrs.my_attribute = dict(A = 10)

In [9]: store.get_storer('df').attrs.my_attribute
Out[9]: {'A': 10}

7.10 Computation

Numerical integration (sample-based) of a time series
7.11 Miscellaneous

The *Timedeltas* docs.
Operating with timedeltas
Create timedeltas with date differences
Adding days to dates in a dataframe

7.12 Aliasing Axis Names

To globally provide aliases for axis names, one can define these 2 functions:

```python
In [10]: 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
    ....:
```
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:

```python
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:

```python
import pandas as pd
```

### 8.1 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:

```python
>>> 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 -1.344
b  0.845
c  1.076
d -0.109
e  1.644
dtype: float64
```

```
In [6]: s.index
Out[6]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

```
In [7]: Series(randn(5))
Out[7]:
0  -1.469
1   0.357
2  -0.675
3  -1.777
4  -0.969
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`.

```
In [11]: Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[11]:
a   5
b   5
c   5
d   5
e   5
dtype: float64
```

### 8.1.1 Series is ndarray-like

Series acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, things like slicing also slice the index.

```
In [12]: s[0]
Out[12]: -1.3443118127316671

In [13]: s[:3]
Out[13]:
a  -1.344
b   0.845
c   1.076
dtype: float64

In [14]: s[s > s.median()]
Out[14]:
c   1.076
e   1.644
dtype: float64

In [15]: s[[4, 3, 1]]
Out[15]:
e   1.644
d  -0.109
b   0.845
dtype: float64

In [16]: np.exp(s)
Out[16]:
a   0.261
b   2.328
c   2.932
d   0.897
e   5.174
dtype: float64
```

We will address array-based indexing in a separate section.

### 8.1.2 Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:
In [17]: s['a']
Out[17]: -1.3443118127316671

In [18]: s['e'] = 12.

In [19]: s
Out[19]:
a   -1.344
b   0.845
c   1.076
d  -0.109
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:

>>> 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.

8.1.3 Vectorized operations and label alignment with Series

When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usually not necessary. Series can be also be passed into most NumPy methods expecting an ndarray.

In [24]: s + s
Out[24]:
a   -2.689
b   1.690
c   2.152
d  -0.218
e   24.000
dtype: float64

In [25]: s * 2
Out[25]:
a   -2.689
b   1.690
c   2.152
d  -0.218
e   24.000
dtype: float64

In [26]: np.exp(s)
A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

The result of an operation between unaligned Series will have the union of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note: In general, we chose to make the default result of operations between differently indexed objects yield the union of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the dropna function.

8.1.4 Name attribute

Series can also have a name attribute:

```python
In [28]: s = Series(np.random.randn(5), name='something')
```

```python
In [29]: s
Out[29]:
0  -1.295
1   0.414
2   0.277
3  -0.472
4  -0.014
Name: something, dtype: float64
```

```python
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.
8.2 DataFrame

**DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

### 8.2.1 From dict of Series or dicts

The result **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will be first converted to Series. If no columns are passed, the columns will be the sorted list of dict keys.

```python
In [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

[4 rows x 2 columns]

In [34]: DataFrame(d, index=['d', 'b', 'a'])
Out[34]:
       one  two
    d   NaN    4
    b    2    2
    a    1    1

[3 rows x 2 columns]

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([u'a', u'b', u'c', u'd'], dtype='object')

In [37]: df.columns
Out[37]: Index([u'one', u'two'], dtype='object')
```

### 8.2.2 From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
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

[4 rows x 2 columns]

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

[4 rows x 2 columns]

### 8.2.3 From structured or record array

This case is handled identically to a dict of arrays.

```
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  2.0 Hello
```
1 2 3 World

[2 rows x 3 columns]

**In [44]:** DataFrame(data, index=[‘first’, ‘second’])

**Out[44]:**

```
          A  B  C
first    1  2  Hello
second   2  3  World
```

[2 rows x 3 columns]

**In [45]:** DataFrame(data, columns=[‘C’, ‘A’, ‘B’])

**Out[45]:**

```
      C  A  B
0  Hello 1  2
1  World 2  3
```

[2 rows x 3 columns]

**Note:** DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

### 8.2.4 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
```

[2 rows x 3 columns]

**In [48]:** DataFrame(data2, index=[‘first’, ‘second’])

**Out[48]:**

```
     a b c
first 1 2 NaN
second 5 10 20
```

[2 rows x 3 columns]

**In [49]:** DataFrame(data2, columns=[‘a’, ‘b’])

**Out[49]:**

```
     a b
 0  1  2
1  5 10
```

[2 rows x 2 columns]

### 8.2.5 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 \texttt{np.nan} for those values which are missing. Alternatively, you may pass a \texttt{numpy.MaskedArray} as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

8.2.6 Alternate Constructors

\texttt{DataFrame.from_dict}

\texttt{DataFrame.from_dict} takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the DataFrame constructor except for the orient parameter which is ‘columns’ by default, but which can be set to ‘index’ in order to use the dict keys as row labels. \texttt{DataFrame.from_records}

\texttt{DataFrame.from_records} takes a list of tuples or an \texttt{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:

\begin{verbatim}
In [50]: data
Out[50]:
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [51]: data = DataFrame.from_records(data, index='C')

In [52]: data
Out[52]:
     A  B
    C
Hello  1  2
World  2  3

[2 rows x 2 columns]

\texttt{DataFrame.from_items}

\texttt{DataFrame.from_items} works analogously to the form of the \texttt{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:

\begin{verbatim}
In [53]: data = DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])

In [54]: data
Out[54]:
     A  B
  0  1  4
  1  2  5
  2  3  6

[3 rows x 2 columns]

If you pass orient=’index’, the keys will be the row labels. But in this case you must also pass the desired column names:

\begin{verbatim}
In [55]: data = DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
                               orient='index', columns=['one', 'two', 'three'])

In [56]: data
Out[56]:
     one  two  three
  A  1    2    3
  B  4    5    6

\end{verbatim}
8.2.7 Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

In [54]: df['one']
Out[54]:
   a  1
   b 2
   c 3
d  NaN
Name: one, dtype: float64

In [55]: df['three'] = df['one'] * df['two']

In [56]: df['flag'] = df['one'] > 2

In [57]: df
Out[57]:
   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 [58]: del df['two']

In [59]: three = df.pop('three')

In [60]: df
Out[60]:
   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 [61]: df['foo'] = 'bar'

In [62]: df
Out[62]:
   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 [63]: df['one_trunc'] = df['one'][:2]
```

```
In [64]: df
Out[64]:
     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 `insert` function is available to insert at a particular location in the columns:

```
In [65]: df.insert(1, 'bar', df['one'])
```

```
In [66]: df
Out[66]:
     one  bar  flag  foo  one_trunc
  a   1   1 False  bar     1
  b   2   2 False  bar     2
  c   3   3  True  bar   NaN
  d  NaN  NaN  False  bar  NaN
```

### 8.2.8 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><code>df[col]</code></td>
<td>Series</td>
</tr>
<tr>
<td>Select row by label</td>
<td><code>df.loc[label]</code></td>
<td>Series</td>
</tr>
<tr>
<td>Select row by integer location</td>
<td><code>df.iloc[loc]</code></td>
<td>Series</td>
</tr>
<tr>
<td>Slice rows</td>
<td><code>df[5:10]</code></td>
<td>DataFrame</td>
</tr>
<tr>
<td>Select rows by boolean vector</td>
<td><code>df[bool_vec]</code></td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [67]: df.loc['b']
Out[67]:
  one  2
  bar  2
  flag False
  foo bar
  one_trunc 2
Name: b, dtype: object
```

```
In [68]: df.iloc[2]
```
8.2.9 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 [69]: df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])
In [70]: df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])
In [71]: df + df2
```

```
Out[71]:
     A   B   C   D
0 -1.473 -0.626 -0.773 NaN
1  0.073 -0.519  2.742 NaN
2  1.744 -1.325  0.075 NaN
3 -1.366 -1.238 -1.782 NaN
4  0.275 -0.613 -2.263 NaN
5  1.263  2.338  1.260 NaN
6 -1.216  3.371 -1.992 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 [72]: df - df.iloc[0]
```

```
Out[72]:
     A   B   C   D
0  0.000  0.000  0.000  0.000
1  1.168 -1.200  3.489  0.536
2  1.703 -1.164  0.697 -0.485
3  1.176  0.138  0.096 -0.972
4 -0.825  1.136 -0.514 -2.309
5  1.970  1.030  1.493 -0.020
6 -1.849  0.981 -1.084 -1.306
7  0.284  0.552 -0.296 -2.123
8  1.132 -1.275  0.195 -1.017
9  0.265  0.702  1.265  0.064
```

In the special case of working with time series data, if the Series is a TimeSeries (which it will be automatically if the index contains datetime objects), and the DataFrame index also contains dates, the broadcasting will be column-wise:
In [73]: index = date_range('1/1/2000', periods=8)

In [74]: df = DataFrame(randn(8, 3), index=index, columns=list('ABC'))

In [75]: df
Out[75]:
   A    B    C
2000-01-01 3.357 -0.317 -1.236
2000-01-02 0.896 -0.488 -0.082
2000-01-03 -2.183  0.380  0.085
2000-01-04 0.432  1.520 -0.494
2000-01-05 0.600  0.274  0.133
2000-01-06 -0.024  2.410  1.451
2000-01-07 0.206 -0.252 -2.214
2000-01-08 1.063  1.266  0.299
[8 rows x 3 columns]

In [76]: type(df['A'])
Out[76]: pandas.core.series.Series

In [77]: df - df['A']
Out[77]:
   A    B    C
2000-01-01 0 -3.675 -4.594
2000-01-02 0 -1.384 -0.978
2000-01-03 0  2.563  2.268
2000-01-04 0  1.088 -0.926
2000-01-05 0 -0.326 -0.467
2000-01-06 0  2.434  1.474
2000-01-07 0 -0.458 -2.420
2000-01-08 0  0.203 -0.764
[8 rows x 3 columns]

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 [78]: df * 5 + 2
Out[78]:
   A    B    C
2000-01-01 18.787  0.413 -4.181
2000-01-02  6.481 -0.438  1.589
2000-01-03 -8.915  3.902  2.424
2000-01-04  4.162  9.600 -0.468
2000-01-05  5.001  3.371  2.664
2000-01-07  3.030  0.740 -9.068
2000-01-08  7.317  8.331  3.497

8.2. DataFrame
In [79]: 1 / df
Out[79]:
       A     B     C
2000-01-01  0.298 -3.150 -0.809
2000-01-02  1.116 -2.051 -12.159
2000-01-03 -0.458  0.658  -2.026
2000-01-04  2.313  0.667  7.525
2000-01-05  1.666  3.647  7.525
2000-01-06 -42.215  0.415  0.689
2000-01-07  4.853 -3.970 -0.452
2000-01-08  0.940  0.790  3.340

[8 rows x 3 columns]

In [80]: df ** 4
Out[80]:
       A        B        C
2000-01-01  1.271e+02  0.010  2.336e+00
2000-01-02  6.450e-01  0.057  4.574e-05
2000-01-03  2.271e+01  0.021  5.182e-05
2000-01-04  3.495e-02  5.338  5.939e-02
2000-01-05  1.298e-01  0.006  3.118e-04
2000-01-06  3.149e-07 33.744  4.427e+00
2000-01-07  1.803e-03  0.004  2.401e+01
2000-01-08  1.278e+00  2.570  8.032e-03

[8 rows x 3 columns]

Boolean operators work as well:

In [81]: df1 = DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)

In [82]: df2 = DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)

In [83]: df1 & df2
Out[83]:
       a      b
0  False  False
1   True   True
2  False  False

[3 rows x 2 columns]

In [84]: df1 | df2
Out[84]:
       a      b
0   True   True
1   True   True
2   True   True

[3 rows x 2 columns]

In [85]: df1 ^ df2
Out[85]:
        a      b
0     True   True
8.2.10 Transposing

To transpose, access the \texttt{T} attribute (also the \texttt{transpose} function), similar to an ndarray:

```python
# only show the first 5 rows
In [87]: df[:5].T
Out[87]:
A   3.357  0.896  -2.183   0.432   0.600
B  -0.317  -0.488   0.380   1.520   0.274
C  -1.236  -0.082   0.085  -0.494   0.133
```

[3 rows x 5 columns]

8.2.11 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:

```python
In [88]: np.exp(df)
Out[88]:
A   28.715    0.728    0.290
B    2.450    0.614    0.921
C    0.113    1.463    1.089
```

[8 rows x 3 columns]

```python
In [89]: np.asarray(df)
Out[89]:
array([[ 3.3574, -0.3174, -1.2363],
       [ 0.8962, -0.4876, -0.0822],
       [-2.1829,  0.3804,  0.0848],
       [ 0.4324,  1.520 , -0.4937],
       [ 0.6002,  0.2742,  0.1329],
       [-0.0237,  2.4102,  1.4505]], dtype=float64)
```
The dot method on DataFrame implements matrix multiplication:

```
In [90]: df.T.dot(df)
Out[90]:
   A  B  C
A 18.562 -0.274 -4.715
B -0.274 10.344 4.184
C -4.715 4.184 8.897
```

Similarly, the dot method on Series implements dot product:

```
In [91]: s1 = Series(np.arange(5,10))
In [92]: s1.dot(s1)
Out[92]: 255
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics are quite different in places from a matrix.

### 8.2.12 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 [93]: baseball = read_csv('data/baseball.csv')
In [94]: print(baseball)
```

```
id  year  stint  team  lg  g  ab  r
88641  womacto01  2006  2  CHN  NL  19  50  6 ...
88643  schicu01  2006  1  BOS  AL  31  2  0 ...
88645  myersmi01  2006  1  NYA  AL  62  0  0 ...
88649  helliri01  2006  1  MIL  NL  20  3  0 ...
88650  johnsra05  2006  1  NYA  AL  33  6  0 ...
...  ...  ...  ...  ...  ...  ...
[100 rows x 22 columns]
```

```
In [95]: baseball.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 88641 to 89534
Data columns (total 22 columns):
id  100 non-null object
year  100 non-null int64
stint  100 non-null int64
team  100 non-null object
lg  100 non-null object
ab  100 non-null int64
g  100 non-null object
r  100 non-null int64
h  100 non-null int64
X2b  100 non-null int64
X3b  100 non-null int64
hr  100 non-null int64
```
pandas: powerful Python data analysis toolkit, Release 0.13.1

rbi 100 non-null float64
sb 100 non-null float64
cs 100 non-null float64
bb 100 non-null int64
so 100 non-null float64
ibb 100 non-null float64
hbp 100 non-null float64
sh 100 non-null float64
sf 100 non-null float64
gidp 100 non-null float64
dtypes: float64(9), int64(10), object(3)

However, using to_string will return a string representation of the DataFrame in tabular form, though it won’t always fit the console width:

```
In [96]: print(baseball.iloc[-20:, :12].to_string())
```

```
id   year  stint  team  lg  g  ab  r  h  X2b  X3b  hr
89474 finlesto1 2007  1 COL  NL 43  94  9 17   3   0   1
89480 embrea101 2007  1 OAK  AL  4  0  0  0   0   0   0
89481 edmonjio1 2007  1 SLN  NL 117 365 39 92  15  2   12
89482 easleda01 2007  1 NYN  NL  76 193 24 54   6   0   10
89489 delgaca01 2007  1 NYN  NL 139 538 71 139  30  0   24
89493 cormirh01 2007  1 CIN  NL  6  0  0  0   0   0   0
89494 coninje01 2007  1 CIN  NL  21  41  2  8   2   0   0
89495 coninje01 2007  1 CIN  NL  80 215 23 57  11  1   6
89497 clemero02 2007  1 NYA  AL  2  2  0  1   0   0   0
89498 claytro01 2007  2 BOS  AL  8  6  1  0   0   0   0
89499 claytro01 2007  1 TOR  AL  69 189 23 48  14  0   1
89501 cirilije01 2007  2 ARI  NL 80  8  6  8   4   0   0
89502 cirilije01 2007  1 MIN  AL 50 153 18 40  9   2   2
89521 bondasba01 2007  1 SFN  NL 126 340 75 94  14  0   28
89522 biggicr01 2007  1 HOU  NL 141 517 68 130  31  3   10
89523 benitaro1 2007  2 FLO  NL  3  0  0  0   0   0   0
89526 benitaro1 2007  1 SFN  NL  9  0  0  0   0   0   0
89530 ausmubr01 2007  1 HOU  NL 117 349 38 82  16  3   3
89533 aloumo01 2007  1 NYN  NL 87 328 51 112  19  1   13
89534 alomasa02 2007  1 NYN  NL  8  22  1  3   1   0   0

New since 0.10.0, wide DataFrames will now be printed across multiple rows by default:

```
In [97]: DataFrame(randn(3, 12))
```

```
Out[97]:
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
8.2.13 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 [100]: df = DataFrame({'foo1' : np.random.randn(5),
                   'foo2' : np.random.randn(5))
...

In [101]: df.foo1
Out[101]:
0    0.281957
1    1.523962
2   -0.902937
3    0.068159
4   -0.057873
dtype: float64

In [102]: df.fo<TAB>
df.foo1 df.foo2
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```python
In [5]: df.fo<TAB>
df.foo1 df.foo2
```
8.3 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:

### 8.3.1 From 3D ndarray with optional axis labels

```
In [103]: 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 [104]: wp
```

```
Out[104]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

### 8.3.2 From dict of DataFrame objects

```
In [105]: data = {'Item1' : DataFrame(randn(4, 3)),
           'Item2' : DataFrame(randn(4, 2))}

In [106]: Panel(data)
```

```
Out[106]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

Note that the values in the dict need only be convertible to DataFrame. Thus, they can be any of the other valid inputs to DataFrame as per above.

One helpful factory method is `Panel.from_dict`, which takes a dictionary of DataFrames as above, and the following named parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersect</td>
<td>False</td>
<td>drops elements whose indices do not align</td>
</tr>
<tr>
<td>orient</td>
<td>items</td>
<td>use minor to use DataFrames’ columns as panel items</td>
</tr>
</tbody>
</table>

For example, compare to the construction above:
Orient is especially useful for mixed-type DataFrames. If you pass a dict of DataFrame objects with mixed-type columns, all of the data will get upcasted to `dtype=object` unless you pass `orient='minor'`:

```python
In [108]: df = DataFrame({'a': [‘foo’, ‘bar’, ‘baz’],
                   ‘b’: np.random.randn(3))
```

```python
In [109]: df
Out[109]:
   a   b
0  foo -1.035260
1  bar -0.438229
2  baz  0.503703
[3 rows x 2 columns]
```

```python
In [110]: data = {‘item1’: df, ‘item2’: df}
```

```python
In [111]: panel = Panel.from_dict(data, orient='minor')
```

```python
In [112]: panel[‘a’]
Out[112]:
   item1  item2
0  foo    foo
1  bar    bar
2  baz    baz
[3 rows x 2 columns]
```

```python
In [113]: panel[‘b’]
Out[113]:
   item1  item2
0 -1.035260 -1.035260
1 -0.438229 -0.438229
2  0.503703  0.503703
[3 rows x 2 columns]
```

```python
In [114]: panel[‘b’].dtypes
Out[114]:
   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.
8.3.3 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 [115]: midx = MultiIndex(levels=[['one', 'two'], ['x','y']], labels=[[1,1,0,0],[1,0,1,0]])

In [116]: df = DataFrame({'A' : [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [117]: df.to_panel()
```

```
Out[117]:
<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
```

8.3.4 Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

```python
In [118]: wp['Item1']
```

```
Out[118]:
     A       B       C       D
2000-01-01 -1.069094 -1.099248  0.255269  0.009750
2000-01-02  0.661084  0.379319 -0.008434  1.952541
2000-01-03 -1.056652  0.533946 -1.226970  0.040403
2000-01-04 -0.507516 -0.230096  0.394500 -1.934370
2000-01-05 -1.652499  1.488753 -0.896484  0.576897
[5 rows x 4 columns]
```

```python
In [119]: wp['Item3'] = wp['Item1'] / wp['Item2']
```

The API for insertion and deletion is the same as for DataFrame. And as with DataFrame, if the item is a valid python identifier, you can access it as an attribute and tab-complete it in IPython.

8.3.5 Transposing

A Panel can be rearranged using its transpose method (which does not make a copy by default unless the data are heterogeneous):

```python
In [120]: wp.transpose(2, 0, 1)
```

```
Out[120]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 5 (minor_axis)
Items axis: A to D
Major_axis axis: Item1 to Item3
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```
8.3.6 Indexing / Selection

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select item</td>
<td><code>wp[item]</code></td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at major_axis label</td>
<td><code>wp.major_xs(val)</code></td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at minor_axis label</td>
<td><code>wp.minor_xs(val)</code></td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

For example, using the earlier example data, we could do:

In [121]: `wp['Item1']`

Out[121]:

```
   A     B     C     D
2000-01-01 -1.069094 -1.099248 0.255269 0.009750
2000-01-02  0.661084  0.379319 -0.008434 1.952541
2000-01-03  1.056652  0.533946 -1.226970  0.040403
2000-01-04 -0.507516 -0.230096  0.394500 -1.934370
2000-01-05 -1.652499  1.488753 -0.896484  0.576897
```

[5 rows x 4 columns]

In [122]: `wp.major_xs(wp.major_axis[2])`

Out[122]:

```
          Item1  Item2  Item3
   A   -1.056652  1.375020  0.768463
   B    0.533946  0.928797  0.574879
   C   -1.226970 -0.308853  3.972672
   D    0.040403 -0.681087  0.059321
```

[4 rows x 3 columns]

In [123]: `wp.minor_axis`

Out[123]: Index([u'A', u'B', u'C', u'D'], dtype='object')

In [124]: `wp.minor_xs('C')`

Out[124]:

```
          Item1  Item2  Item3
   2000-01-01  0.255269  0.604603  0.422209
   2000-01-02 -0.008434  0.967661 -0.008715
   2000-01-03 -1.226970 -0.308853  3.972672
   2000-01-04  0.394500 -2.461467 -0.160270
   2000-01-05 -0.896484  1.771740 -0.505991
```

[5 rows x 3 columns]

8.3.7 Squeezing

Another way to change the dimensionality of an object is to squeeze a 1-len object, similar to `wp['Item1']`

In [125]: `wp.reindex(items=['Item1']).squeeze()`

Out[125]:

```
   A     B     C     D
2000-01-01 -1.069094 -1.099248 0.255269 0.009750
2000-01-02  0.661084  0.379319 -0.008434 1.952541
2000-01-03  1.056652  0.533946 -1.226970  0.040403
2000-01-04 -0.507516 -0.230096  0.394500 -1.934370
2000-01-05 -1.652499  1.488753 -0.896484  0.576897
```
8.3.8 Conversion to DataFrame

A Panel can be represented in 2D form as a hierarchically indexed DataFrame. See the section \textit{hierarchical indexing} for more on this. To convert a Panel to a DataFrame, use the \texttt{to_frame} method:

```python
In [127]: 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'])

In [128]: panel.to_frame()
```

```
Out[128]:
     one  two  three
major minor          
2000-01-01 a  0.413086 -0.033277 -1.132896
   b -1.139050  0.281151 -2.006481
   c  0.660342 -1.298915  0.301016
   d  0.464794 -2.819487  0.059117
2000-01-02 a -0.309337 -0.851985  1.138469
   b -0.649593 -1.106952 -2.400634
   c  0.683758 -0.937731 -0.280853
   d -0.643834 -1.537770  0.025653
2000-01-03 a  0.421287  0.555759 -1.386071
   b  1.032814 -2.277282  0.863937
   c -1.290493 -0.390201  0.252462
   d  0.787872  1.207122  1.500571
2000-01-04 a  1.515707  0.178690  1.053202
   b -0.276487 -1.004168 -2.338595
   c -0.223762 -1.377627 -0.374279
   d  1.397431  0.499281 -2.359958
2000-01-05 a  1.503874 -1.405256 -1.157886
   b -0.478905  0.162565 -0.551865
   c -0.135950 -0.067785  1.592673
   d -0.730327 -1.260006  1.559318
```

[20 rows x 3 columns]
• **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

### 8.4.1 From 4D ndarray with optional axis labels

```
In [129]: p4d = Panel4D(randn(2, 2, 5, 4),
        labels=['Label1','Label2'],
        items=['Item1', 'Item2'],
        major_axis=date_range('1/1/2000', periods=5),
        minor_axis=['A', 'B', 'C', 'D'])
```

```
In [130]: p4d
Out[130]:
<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
```

### 8.4.2 From dict of Panel objects

```
In [131]: data = { 'Label1' : Panel({ 'Item1' : DataFrame(randn(4, 3)) }),
        'Label2' : Panel({ 'Item2' : DataFrame(randn(4, 2)) }) }
```

```
In [132]: Panel4D(data)
Out[132]:
<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.

### 8.4.3 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
In [133]: p4d[‘Label1’]
Out[133]:
<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

In [134]: p4d.ix[:, :, :, ‘A’]
Out[134]:
<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 [135]: p4d.ix[:, :, 0, ‘A’]
Out[135]:
Label1  Label2
Item1  1.562443  0.159653
Item2  -0.015601  0.136235
[2 rows x 2 columns]
4D -> Series

In [136]: p4d.ix[:, 0, 0, ‘A’]
Out[136]:
Label1  1.562443
Label2  0.159653
Name: A, dtype: float64

8.4.4 Transposing

A Panel4D can be rearranged using its transpose method (which does not make a copy by default unless the data are heterogeneous):

In [137]: p4d.transpose(3, 2, 1, 0)
Out[137]:
<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

8.5 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 [138]: from pandas.core import panelnd

In [139]: 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 [140]: p5d = Panel5D(dict(C1 = p4d))

In [141]: p5d
Out[141]:
<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 [142]: p5d.ix['C1',::,0:3,::]
Out[142]:
<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 [143]: p5d.transpose(1,2,3,4,0)
Out[143]:
<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 [144]: p5d.shape
Out[144]: (1, 2, 2, 5, 4)

In [145]: p5d.ndim
Out[145]: 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 = date_range('1/1/2000', periods=8)

In [2]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = DataFrame(randn(8, 3), index=index, columns=['A', 'B', 'C'])

In [4]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'], major_axis=date_range('1/1/2000', periods=5), minor_axis=['A', 'B', 'C', 'D'])

9.1 Head and Tail

To view a small sample of a Series or DataFrame object, use the head and tail methods. The default number of elements to display is five, but you may pass a custom number.

In [5]: long_series = Series(randn(1000))

In [6]: long_series.head()
Out[6]:
0  -0.199038
1   1.095864
2  -0.200875
3   0.162291
4  -0.430489
   dtype: float64

In [7]: long_series.tail(3)
Out[7]:
997  -1.198693
998   1.238029
999  -1.344716
   dtype: float64
9.2 Attributes and the raw ndarray(s)

Pandas objects have a number of attributes enabling you to access the metadata:

- **shape**: gives the axis dimensions of the object, consistent with ndarray

- **Axis labels**
  - **Series**: `index` (only axis)
  - **DataFrame**: `index` (rows) and `columns`
  - **Panel**: `items`, `major_axis`, and `minor_axis`

Note, these attributes can be safely assigned to!

```python
In [8]: df[:2]
Out[8]:
     A    B    C
2000-01-01  0.232465 -0.789552 -0.364308
2000-01-02 -0.534541  0.822239 -0.443109
[2 rows x 3 columns]
```

```python
In [9]: df.columns = [x.lower() for x in df.columns]
```

```python
In [10]: df
Out[10]:
     a    b    c
2000-01-01  0.232465 -0.789552 -0.364308
2000-01-02 -0.534541  0.822239 -0.443109
2000-01-03 -2.119990 -0.460149  1.813962
2000-01-04 -1.053571  0.009412 -0.165966
2000-01-05 -0.848662 -0.495553 -0.176421
2000-01-06 -0.423595 -1.035433 -1.035374
2000-01-07 -2.369079  0.524408 -0.871120
2000-01-08  1.585433  0.039501  2.274101
[8 rows x 3 columns]
```

To get the actual data inside a data structure, one need only access the `values` property:

```python
In [11]: s.values
Out[11]: array([ 1.1292, 0.2313, -0.1847, -0.1386, -0.9243])
```

```python
In [12]: df.values
Out[12]:
array([[ 0.2325, -0.7896, -0.3643],
       [-0.5345,  0.8222, -0.4431],
       [-2.12 , -0.4601,  1.814 ],
       [-1.0536,  0.0094, -0.166 ],
       [-0.8487, -0.4956, -0.1764],
       [-0.4236, -1.0354, -1.0354],
       [-2.3691,  0.5244, -0.8711],
       [ 1.5854,  0.0395,  2.2741]])
```

```python
In [13]: wp.values
Out[13]:
array([[-1.1181,  0.4313,  0.5547, -1.3336],
       [-0.3322, -0.4859,  1.7259,  1.7993],
       [ 1.5854,  0.0395,  2.2741]])
```
If a DataFrame or Panel contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame’s columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

Note: When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

9.3 Accelerated operations

Pandas has support for accelerating certain types of binary numerical and boolean operations using the numexpr library (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.

9.4 Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

9.4.1 Matching / broadcasting behavior

DataFrame has the methods add, sub, mul, div and related functions radd, rsub, ... for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the index or columns via the axis keyword:
In [14]: d = {'one' : Series(randn(3), index=['a', 'b', 'c']),
        'two' : Series(randn(4), index=['a', 'b', 'c', 'd']),
        'three' : Series(randn(3), index=['b', 'c', 'd'])}

In [15]: df = df_orig = DataFrame(d)

In [16]: df
Out[16]:
    one three two
   a  -0.701368 NaN -0.087103
   b   0.109333 -0.354359  0.637674
   c  -0.231617 -0.148387 -0.002666
   d   NaN   0.167407  0.104044

[4 rows x 3 columns]

In [17]: row = df.ix[1]

In [18]: column = df['two']

In [19]: df.sub(row, axis='columns')
Out[19]:
    one three two
   a  -0.810701 NaN -0.724777
   b    0.000000  0.000000  0.000000
   c  -0.340950  0.205973 -0.640340
   d   NaN   0.186952 -0.533630

[4 rows x 3 columns]

In [20]: df.sub(row, axis=1)
Out[20]:
    one three two
   a  -0.810701 NaN -0.724777
   b    0.000000  0.000000  0.000000
   c  -0.340950  0.205973 -0.640340
   d   NaN   0.186952 -0.533630

[4 rows x 3 columns]

In [21]: df.sub(column, axis='index')
Out[21]:
    one three two
   a  -0.614265 NaN  0
   b  -0.528341 -0.992033  0
   c  -0.228950 -0.145720  0
   d   NaN  -0.271451  0

[4 rows x 3 columns]

In [22]: df.sub(column, axis=0)
Out[22]:
    one three two
   a  -0.614265 NaN  0
   b  -0.528341 -0.992033  0
   c  -0.228950 -0.145720  0
   d   NaN  -0.271451  0
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:

**In [23]:** \( \text{major\_mean} = \text{wp.mean(axis='major')} \)

**In [24]:** `major_mean`

**Out[24]:**

```
   Item1  Item2
A -0.688773 -0.021497
B  0.114982 -0.094183
C  0.035674 -0.156470
D -0.204142 -0.606887
```

**In [25]:** `wp.sub(major_mean, axis='major')`

**Out[25]:**

```
<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...

### 9.4.2 Missing data / operations with fill values

In Series and DataFrame (though not yet in Panel), the arithmetic functions have the option of inputting a *fill_value*, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using `fillna` if you wish).

**In [26]:** `df`

**Out[26]:**

```
  one  three  two
a -0.701368 NaN -0.087103
b  0.109333 -0.354359 0.637674
c -0.231617 -0.148387 -0.002666
d   NaN      0.167407  0.104044
```

**In [27]:** `df2`

**Out[27]:**

```
  one  three  two
a -0.701368 1.000000 -0.087103
b  0.109333 -0.354359 0.637674
c -0.231617 -0.148387 -0.002666
d   NaN      0.167407  0.104044
```
9.4.3 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 [30]: df.gt(df2)
Out[30]:
   one   three   two
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [31]: df2.ne(df)
Out[31]:
   one   three   two
a  False   True  False
b  False  False  False
c  False  False  False
d   True  False  False

These operations produce a pandas object the same type as the left-hand-side input that if of dtype bool. These boolean objects can be used in indexing operations, see here

9.4.4 Boolean Reductions

You can apply the reductions: empty, any(), all(), and bool() to provide a way to summarize a boolean result.
In [32]: (df>0).all()
Out[32]:
one     False
three   False
two     False
dtype: bool

In [33]: (df>0).any()
Out[33]:
one      True
three    False
two      True
dtype: bool

You can reduce to a final boolean value.

In [34]: (df>0).any().any()
Out[34]: True

You can test if a pandas object is empty, via the empty property.

In [35]: df.empty
Out[35]: False

In [36]: DataFrame(columns=list('ABC')).empty
Out[36]: True

To evaluate single-element pandas objects in a boolean context, use the method .bool():

In [37]: Series([True]).bool()
Out[37]: True

In [38]: Series([False]).bool()
Out[38]: False

In [39]: DataFrame([[True]]).bool()
Out[39]: True

In [40]: DataFrame([[False]]).bool()
Out[40]: False

Warning: You might be tempted to do the following:

```python
>>> if df:
...     ...
```

Or

```python
>>> df and df2
```

These both will raise as you are trying to compare multiple values.

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all()
```

See gotchas for a more detailed discussion.
9.4.5 Comparing if objects are equivalent

Often you may find there is more than one way to compute the same result. As a simple example, consider \(df+df\) and \(df*2\). To test that these two computations produce the same result, given the tools shown above, you might imagine using \((df+df == df*2).all()\). But in fact, this expression is False:

```
In [41]: df+df == df*2
Out[41]:
       one  three  two
  a  True  False  True
  b  True   True  True
  c  True   True  True
  d   False  True  True
```

Notice that the boolean DataFrame \(df+df == df*2\) contains some False values! That is because NaNs do not compare as equals:

```
In [43]: np.nan == np.nan
Out[43]: 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 [44]: (df+df).equals(df*2)
Out[44]: True
```

9.4.6 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 [45]: df1 = DataFrame({'A': [1., np.nan, 3., 5., np.nan],
                      'B': [np.nan, 2., 3., np.nan, 6.]})
   ....: 
   ....:

In [46]: df2 = DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
                      'B': [np.nan, np.nan, 3., 4., 6., 8.]})
   ....: 
   ....:

In [47]: df1
Out[47]:
      A    B
  0  NaN  NaN
  1  NaN  2
```
9.4.7 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 (ie, columns whose names are the same).

So, for instance, to reproduce `combine_first` as above:

```python
In [50]: combiner = lambda x, y: np.where(isnull(x), y, x)
In [51]: df1.combine(df2, combiner)
```

[6 rows x 2 columns]
9.5 Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on `Series`, `DataFrame`, and `Panel`. Most of these are aggregations (hence producing a lower-dimensional result) like `sum`, `mean`, and `quantile`, but some of them, like `cumsum` and `cumprod`, produce an object of the same size. Generally speaking, these methods take an `axis` argument, just like `ndarray.sum`, `std`, `...`, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: “index” (axis=0, default), “columns” (axis=1)
- **Panel**: “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

```python
In [52]: df
Out[52]:
   one     three     two
a  0.701368  NaN   0.087103
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
d  NaN     -0.167407  0.104044

[4 rows x 3 columns]
```

```python
In [53]: df.mean(0)
Out[53]:
   one     three
a  -0.274551  NaN
b   0.162987
```

```python
In [54]: df.mean(1)
Out[54]:
   a     b     c     d
-0.394235  0.130882 -0.127557 -0.031682
```

All such methods have a `skipna` option signaling whether to exclude missing data (True by default):

```python
In [55]: df.sum(0, skipna=False)
Out[55]:
   one
a  0.788471
b -0.392647
c -0.382670
d  0.104044
```

```python
In [56]: df.sum(axis=1, skipna=True)
Out[56]:
   a     b     c     d
-0.788471  0.392647 -0.382670 -0.063363
```

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 [57]: ts_stand = (df - df.mean()) / df.std()

In [58]: ts_stand.std()
Out[58]:
one  1
three 1
two  1
dtype: float64

In [59]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [60]: xs_stand.std(1)
Out[60]:
a  1
b  1
c  1
d  1
dtype: float64

Note that methods like cumsum and cumprod preserve the location of NA values:

In [61]: df.cumsum()
Out[61]:
   one   three   two
a -0.701368  NaN -0.087103
b -0.592035 -0.354359  0.550570
c -0.823652 -0.502746  0.547904
d  NaN -0.670153  0.651948

[4 rows x 3 columns]

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>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:

9.5. Descriptive statistics
In [62]: np.mean(df['one'])
Out[62]: -0.27455055654271204

In [63]: np.mean(df['one'].values)
Out[63]: nan

Series also has a method nunique which will return the number of unique non-null values:

In [64]: series = Series(randn(500))
In [65]: series[20:500] = np.nan
In [66]: series[10:20] = 5
In [67]: series.nunique()
Out[67]: 11

9.5.1 Summarizing data: describe

There is a convenient describe function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

In [68]: series = Series(randn(1000))
In [69]: series[::2] = np.nan
In [70]: series.describe()
Out[70]:
            count  500.000000
            mean   -0.019898
            std     1.019180
            min    -2.628792
            25%    -0.649795
            50%    -0.059405
            75%     0.651932
            max     3.240991
            dtype: float64

In [71]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [72]: frame.ix[::2] = np.nan
In [73]: frame.describe()
Out[73]:
         count  a       b       c       d       e
         count  500.000000  500.000000  500.000000  500.000000  500.000000
         mean     0.051388   0.053476  -0.035612   0.015388   0.057804
         std      0.989217   0.995961   0.977047   0.968385   1.022528
         min     -3.224136  -2.606460  -2.762875  -2.961757  -2.829100
         25%     -0.657420  -0.597123  -0.688961  -0.695019  -0.738097
         50%      0.042928   0.018837  -0.071830  -0.011326   0.073287
         75%      0.702445   0.693542   0.600454   0.680924   0.807670
         max      3.034008   3.104512   2.812028   2.623914   3.542846

[8 rows x 5 columns]

For a non-numerical Series object, describe will give a simple summary of the number of unique values and most
frequently occurring values:

```python
In [74]: s = Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])
In [75]: s.describe()
Out[75]:
   count 9
   unique 4
   top    a
   freq  5
  dtype: object
```

There also is a utility function, `value_range` which takes a DataFrame and returns a series with the minimum/maximum values in the DataFrame.

### 9.5.2 Index of Min/Max Values

The `idxmin` and `idxmax` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```python
9.5.2 Index of Min/Max Values

The `idxmin` and `idxmax` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

In [76]: s1 = Series(randn(5))
In [77]: s1
Out[77]:
   0   -0.574018
   1    0.668292
   2    0.303418
   3   -1.190271
   4    0.138399
  dtype: float64
In [78]: s1.idxmin(), s1.idxmax()
Out[78]: (3, 1)
In [79]: df1 = DataFrame(randn(5,3), columns=['A','B','C'])
In [80]: df1
Out[80]:
   A          B          C
  0 -0.184355 -1.054354 -1.613138
  1 -0.050807 -2.130168 -1.852271
  2  0.455674  2.571061 -1.152538
  3 -1.638940 -0.364831 -0.348520
  4  0.202856  0.777088 -0.358316
[5 rows x 3 columns]
In [81]: df1.idxmin(axis=0)
Out[81]:
   A  B  C
 0  3  1  1
dtype: int64
In [82]: df1.idxmax(axis=1)
Out[82]:
  0  A
  1  A
```

9.5. Descriptive statistics
When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin` and `idxmax` return the first matching index:

```python
In [83]: df3 = DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))
In [84]: df3
Out[84]:
   A
e 2
d 1
c 1
b 3
a NaN
[5 rows x 1 columns]
In [85]: df3['A'].idxmin()
Out[85]: 'd'
```

**Note:** `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

### 9.5.3 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:

```python
In [86]: data = np.random.randint(0, 7, size=50)
In [87]: data
Out[87]:
array([4, 6, 6, 1, 2, 1, 0, 5, 3, 2, 4, 3, 1, 3, 5, 3, 0, 0, 4, 4, 6, 1, 0,
      4, 3, 2, 1, 3, 1, 5, 6, 3, 1, 2, 4, 4, 3, 3, 2, 2, 3, 2, 3, 0, 1,
      2, 4, 5, 5])
In [88]: s = Series(data)
In [89]: s.value_counts()
Out[89]:
3    11
2     9
1     8
5     5
0     5
6     4
type: int64
In [90]: value_counts(data)
Out[90]:
3    11
2     9
```
Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

\[
\text{In [91]: } s5 = \text{Series([1, 1, 3, 3, 5, 5, 7, 7, 7])}
\]

\[
\text{In [92]: } s5.mode()
\]

\[
\text{Out[92]:}
\]

\[
\begin{array}{l}
0\ 3 \\
1\ 7 \\
\end{array}
\]

dtype: int64

\[
\text{In [93]: } df5 = \text{DataFrame({"A": np.random.randint(0, 7, size=50),}
\]

\[
\ldots:
\]

\[
\ldots:
\]

\[
\text{In [94]: } df5.mode()
\]

\[
\text{Out[94]:}
\]

\[
\begin{array}{ll}
A & B \\
0 & 5 \ -4 \\
1 & 6 \ NaN \\
\end{array}
\]

[2 rows x 2 columns]

### 9.5.4 Discretization and quantiling

Continuous values can be discretized using the *cut* (bins based on values) and *qcut* (bins based on sample quantiles) functions:

\[
\text{In [95]: } arr = \text{np.random.randn(20)}
\]

\[
\text{In [96]: } \text{factor = cut(arr, 4)}
\]

\[
\text{In [97]: } \text{factor}
\]

\[
\text{Out[97]:}
\]

\[
\begin{array}{ll}
(-0.886, -0.0912] \\
(-0.886, -0.0912] \\
(-0.886, -0.0912] \\
(0.701, 1.493] \\
\ldots
\end{array}
\]

\[
\begin{array}{ll}
(-0.0912, 0.701] \\
(-0.886, -0.0912] \\
(0.701, 1.493] \\
(0.701, 1.493] \\
(-0.0912, 0.701] \\
(1.493, 2.285] \\
\end{array}
\]

\[
\text{Levels (4): Index(["(-0.886, -0.0912]", "(-0.0912, 0.701]", 
}\]

\[
\ldots
\]

\[
\text{\ldots:}
\]

\[
\text{"(-0.0912, 0.701]", "(-0.0912, 0.701]", 
}\]

\[
\ldots:
\]

\[
\text{"(1.493, 2.285]"], dtype=object)}
\]

\[
\text{Length: 20}
\]

\[
\text{In [98]: } \text{factor = cut(arr, [-5, -1, 0, 1, 5])}
\]

### 9.5. Descriptive statistics
In [99]: factor
Out[99]:
(-1, 0)
(-1, 0)
(-1, 0)
(1, 5)
(1, 5)
...
(0, 1)
(-1, 0)
(0, 1)
(0, 1)
(0, 1)
(1, 5)
Levels (4): Index([’(-5, -1]’, ’(-1, 0]’, ’(0, 1]’, ’(1, 5]’], dtype=object)
Length: 20

qcut computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

In [100]: arr = np.random.randn(30)
In [101]: factor = qcut(arr, [0, .25, .5, .75, 1])
In [102]: factor
Out[102]:
[-1.861, -0.487]
(0.0554, 0.658]
(0.658, 2.259]
...
(0.0554, 0.658]
(0.658, 2.259]
Levels (4): Index([’[-1.861, -0.487]’, ’(-0.487, 0.0554]’,
’(0.0554, 0.658]’, ’(0.658, 2.259]’], dtype=object)
Length: 30

In [103]: value_counts(factor)
Out[103]:
(0.658, 2.259] 8
[-1.861, -0.487] 8
(0.0554, 0.658] 7
(-0.487, 0.0554] 7
dtype: int64

We can also pass infinite values to define the bins:

In [104]: arr = np.random.randn(20)
In [105]: factor = cut(arr, [-np.inf, 0, np.inf])
In [106]: factor
(0, inf]
(0, inf]
(-inf, 0]
(0, inf]
(-inf, 0]
...
(-inf, 0]
(0, inf]
(-inf, 0]
(0, inf]
(-inf, 0]
Levels (2): Index([’(-inf, 0]’, ’(0, inf]’], dtype=object)
Length: 20

9.6 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:

```python
In [107]: df.apply(np.mean)
Out[107]:
one   -0.274551
three -0.223384
two   0.162987
dtype: float64

In [108]: df.apply(np.mean, axis=1)
Out[108]:
a   -0.394235
b   0.130882
c  -0.127557
d  -0.031682
dtype: float64

In [109]: df.apply(lambda x: x.max() - x.min())
Out[109]:
one     0.810701
three  0.205973
two    0.724777
dtype: float64

In [110]: df.apply(np.cumsum)
Out[110]:
one three two
 a -0.701368  NaN  -0.087103
b -0.592035 -0.354359  0.550570
c -0.823652 -0.502746  0.547904
d  NaN   -0.670153  0.651948
[4 rows x 3 columns]

In [111]: df.apply(np.exp)
Out[111]:
one three two
 a  0.495907  NaN  0.916583
```
Depending on the return type of the function passed to `apply`, the result will either be of lower dimension or the same dimension.

`apply` combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```python
In [112]: tsdf = DataFrame(randn(1000, 3), columns=['A', 'B', 'C'],
                      index=date_range('1/1/2000', periods=1000))
```

```python
In [113]: tsdf.apply(lambda x: x.idxmax())
```

```
Out[113]:
   A      B      C
2002-08-19 -1.226159 0.173875 -0.798063
2000-11-30 -1.134931 0.808796 0.661270
2002-01-10 -0.465633 0.737792 0.857375
```

You may also pass additional arguments and keyword arguments to the `apply` method. For instance, consider the following function you would like to apply:

```python
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```python
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```python
In [114]: tsdf
Out[114]:
     A         B         C
2000-01-01 -1.226159 0.173875 -0.798063
2000-01-02  0.127076 0.141070 -2.186743
2000-01-03 -1.804229 0.879800  0.465165
2000-01-04  NaN       NaN       NaN
2000-01-05  NaN       NaN       NaN
2000-01-06  NaN       NaN       NaN
2000-01-07  NaN       NaN       NaN
2000-01-08  1.542261 0.524780  1.445690
2000-01-09 -1.104998 -0.470200  0.336180
2000-01-10 -0.947692 -0.262122 -0.423769
```

```python
In [115]: tsdf.apply(Series.interpolate)
Out[115]:
    A         B         C
2000-01-01 -1.226159 0.173875 -0.798063
2000-01-02  0.127076 0.141070 -2.186743
2000-01-03 -1.804229 0.879800  0.465165
2000-01-04 -1.134931 0.808796  0.661270
2000-01-05 -0.465633 0.737792  0.857375
2000-01-06  0.203665 0.666788  1.053480
```
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.

### 9.6.1 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:

```python
In [116]: df4
Out[116]:
     one   three  two
a -0.701368  NaN -0.087103
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
d   NaN     -0.167407  0.104044

[4 rows x 3 columns]
```

```python
In [117]: f = lambda x: len(str(x))
```

```python
In [118]: df4[‘one’].map(f)
Out[118]:
a    15
b    14
c    15
d     3
Name: one, dtype: int64
```

```python
In [119]: df4.applymap(f)
Out[119]:
     one   three  two
a    15     3    16
b    14    15    14
c    15    15    17
d     3    15    14

[4 rows x 3 columns]
```

`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:

```python
In [120]: s = Series([‘six’, ‘seven’, ‘six’, ‘seven’, ‘six’],
          index=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’])
```

9.6. Function application
In [121]: t = Series({'six' : 6., 'seven' : 7.})

In [122]: s
Out[122]:
   a  six
   b  seven
   c  six
   d  seven
   e  six
dtype: object

In [123]: s.map(t)
Out[123]:
   a   6
   b   7
   c   6
   d   7
   e   6
dtype: float64

9.6.2 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 [124]: import pandas.util.testing as tm
In [125]: panel = tm.makePanel(5)
In [126]: panel
Out[126]:
<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 [127]: panel['ItemA']
Out[127]:
      A       B       C       D
2000-01-03  0.166882 -0.597361  1.200639  0.174260
2000-01-04 -1.759496 -1.514940  2.872993 -0.581163
2000-01-05  0.901336 -1.640398  0.825210  0.087916
2000-01-06 -0.317478 -1.130643 -0.392715  0.416971
2000-01-07 -0.681335 -0.245890 -1.994150  0.666084
[5 rows x 4 columns]
A transformational apply.
In [128]: result = panel.apply(lambda x: x**2, axis='items')
In [129]: result
Out[129]:
<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 [130]: result['ItemA']
Out[130]:
      A   B    C   D
2000-01-03 0.333764 -1.194722 -2.401278  0.348520
2000-01-05  1.802673 -3.280796  1.650421  0.175832
2000-01-06 -0.634955 -2.261286 -0.785430  0.833943
2000-01-07 -1.362670 -0.491779 -3.988300  1.332168
[5 rows x 4 columns]

A reduction operation.

In [131]: panel.apply(lambda x: x.dtype, axis='items')
Out[131]:
      A   B    C   D
2000-01-03  float64  float64  float64  float64
2000-01-04  float64  float64  float64  float64
2000-01-05  float64  float64  float64  float64
2000-01-06  float64  float64  float64  float64
2000-01-07  float64  float64  float64  float64
[5 rows x 4 columns]

A similar reduction type operation

In [132]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[132]:
        ItemA  ItemB  ItemC
A -1.690090  1.840259  0.010754
B -5.129232  0.860182  0.178018
C -4.635286  0.545328  2.456520
D  0.764068 -3.623586  1.761541
[4 rows x 3 columns]

This last reduction is equivalent to

In [133]: panel.sum('major_axis')
Out[133]:
        ItemA  ItemB  ItemC
A -1.690090  1.840259  0.010754
B -5.129232  0.860182  0.178018
C -4.635286  0.545328  2.456520
D  0.764068 -3.623586  1.761541
[4 rows x 3 columns]

A transformation operation that returns a Panel, but is computing the z-score across the major_axis.
Apply can also accept multiple axes in the `axis` argument. This will pass a DataFrame of the cross-section to the applied function.

```
In [137]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T
In [138]: result = panel.apply(f, axis = ['items','major_axis'])
```

This is equivalent to the following

```
In [141]: result = Panel(dict([(ax,f(panel.loc[:,:,ax])) for ax in panel.minor_axis ]))
```

In [142]: result
Out[142]: <class 'pandas.core.panel.Panel'>
9.7 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:

In [144]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [145]: s.reindex(['e', 'b', 'f', 'd'])

Out[145]:
    e  0.005240
    b -1.069046
    f  NaN
    d -0.944778
dtype: float64

Here, the f label was not contained in the Series and hence appears as NaN in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

In [147]: df

Out[147]:
   one   three  two
a -0.701368  NaN  -0.087103
b  0.109333 -0.354359  0.637674
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:

```python
In [149]: rs = s.reindex(df.index)
```

```python
Out[150]:
  a   1.112686
  b  -1.069046
  c  -1.218080
dtype: float64
```

```python
In [151]: rs.index is df.index
```

```python
Out[151]: True
```

This means that the reindexed Series’s index is the same Python object as the DataFrame’s index.

**See Also:**

`Advanced indexing` is an even more concise way of doing reindexing.

---

**Note:** When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data.** Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

### 9.7.1 Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like` method is available to make this simpler:

```python
In [152]: df2
```

```python
Out[152]:
   one  two
  a -0.701368 -0.087103
  b  0.109333  0.637674
c -0.231617 -0.002666
```

---

196 Chapter 9. Essential Basic Functionality
9.7.2 Reindexing with `reindex_axis`

9.7.3 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
- `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 [155]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [156]: s1 = s[:4]
In [157]: s2 = s[1:]
In [158]: s1.align(s2)
Out[158]:
(a 0.479090
 b 0.686579
c -0.949750
d -0.257472
e NaN
dtype: float64, a   NaN
 b 0.686579
c -0.949750
d -0.257472
e -0.568459
dtype: float64)
```

```
In [159]: s1.align(s2, join='inner')
```
Out[159]:
(b 0.686579
c -0.949750
d -0.257472
dtype: float64, b 0.686579
c -0.949750
d -0.257472
dtype: float64)

In [160]: s1.align(s2, join='left')
Out[160]:
(a 0.479090
b 0.686579
c -0.949750
d -0.257472
dtype: float64, a NaN
b 0.686579
c -0.949750
d -0.257472
dtype: float64)

For DataFrames, the join method will be applied to both the index and the columns by default:

In [161]: df.align(df2, join='inner')
Out[161]:
( one two
a -0.701368 -0.087103
b 0.109333 0.637674
c -0.231617 -0.002666
[3 rows x 2 columns], one two
a -0.701368 -0.087103
b 0.109333 0.637674
c -0.231617 -0.002666
[3 rows x 2 columns])

You can also pass an axis option to only align on the specified axis:

In [162]: df.align(df2, join='inner', axis=0)
Out[162]:
( one three two
a -0.701368 NaN -0.087103
b 0.109333 -0.354359 0.637674
c -0.231617 -0.148387 -0.002666
[3 rows x 3 columns], one two
a -0.701368 -0.087103
b 0.109333 0.637674
c -0.231617 -0.002666
[3 rows x 2 columns])

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 [163]: df.align(df2.ix[0], axis=1)
Out[163]:
( one three two
a -0.701368  NaN  -0.087103
b  0.109333  -0.354359  0.637674
c -0.231617  -0.148387  -0.002666
d  NaN  0.167407  0.104044

[4 rows x 3 columns], one -0.701368
two  NaN
three  -0.087103
Name: a, dtype: float64)

9.7.4 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 / fill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
</tbody>
</table>

Other fill methods could be added, of course, but these are the two most commonly used for time series data. In a way they only make sense for time series or otherwise ordered data, but you may have an application on non-time series data where this sort of “interpolation” logic is the correct thing to do. More sophisticated interpolation of missing values would be an obvious extension.

We illustrate these fill methods on a simple TimeSeries:

In [164]: rng = date_range('1/3/2000', periods=8)

In [165]: ts = Series(randn(8), index=rng)

In [166]: ts2 = ts[[0, 3, 6]]

9.7. Reindexing and altering labels
2000-01-08 NaN
2000-01-09 0.688500
2000-01-10 NaN
Freq: D, dtype: float64

In [170]: ts2.reindex(ts.index, method='ffill')
Out[170]:
2000-01-03 -0.059786
2000-01-04 -0.059786
2000-01-05 -0.059786
2000-01-06 0.836517
2000-01-07 0.836517
2000-01-08 0.836517
2000-01-09 0.688500
2000-01-10 0.688500
Freq: D, dtype: float64

In [171]: ts2.reindex(ts.index, method='bfill')
Out[171]:
2000-01-03 -0.059786
2000-01-04 0.836517
2000-01-05 0.836517
2000-01-06 0.836517
2000-01-07 0.688500
2000-01-08 0.688500
2000-01-09 0.688500
2000-01-10 NaN
Freq: D, dtype: float64

Note these methods require that the indexes are order increasing.

Note the same result could have been achieved using fillna:

In [172]: ts2.reindex(ts.index).fillna(method='ffill')
Out[172]:
2000-01-03 -0.059786
2000-01-04 0.836517
2000-01-05 0.836517
2000-01-06 0.836517
2000-01-07 0.688500
2000-01-08 0.688500
2000-01-09 0.688500
2000-01-10 NaN
Freq: D, dtype: float64

Note that reindex will raise a ValueError if the index is not monotonic. fillna will not make any checks on the order of the index.

9.7.5 Dropping labels from an axis

A method closely related to reindex is the drop function. It removes a set of labels from an axis:

In [173]: df
Out[173]:
   one   three   two
a -0.701368    NaN  -0.087103
b   0.109333  -0.354359  0.637674
c -0.231617  -0.148387 -0.002666
d             NaN -0.167407  0.104044
[4 rows x 3 columns]

In [174]: df.drop(['a', 'd'], axis=0)
Out[174]:
     one  three  two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
[2 rows x 3 columns]

In [175]: df.drop(['one'], axis=1)
Out[175]:
      three  two
a   NaN -0.087103
b -0.354359  0.637674
c -0.148387 -0.002666
d -0.167407  0.104044
[4 rows x 2 columns]

Note that the following also works, but is a bit less obvious / clean:

In [176]: df.reindex(df.index - ['a', 'd'])
Out[176]:
     one  three  two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
[2 rows x 3 columns]

9.7.6 Renaming / mapping labels

The rename method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

In [177]: s
Out[177]:
   a    0.479090
   b    0.686579
   c   -0.949750
   d   -0.257472
   e   -0.568459
dtype: float64

In [178]: s.rename(str.upper)
Out[178]:
      A    0.479090
      B    0.686579
      C   -0.949750
      D   -0.257472
      E   -0.568459
dtype: float64

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). But if you pass a dict or Series, it need only contain a subset of the labels as keys:
In [179]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
.....: index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})

Out[179]:
          foo    three   bar
apple  -0.701368   NaN -0.087103
banana  0.109333 -0.354359  0.637674
c      -0.231617 -0.148387 -0.002666
durian   NaN    -0.167407  0.104044

[4 rows x 3 columns]

The `rename` method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place. The `Panel` class has a related `rename_axis` class which can rename any of its three axes.

9.8 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 [180]: for col in df:
.....:     print(col)

one
three
two

9.8.1 iteritems

Consistent with the dict-like interface, `iteritems` iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, `Series`) pairs
- **Panel**: (item, DataFrame) pairs

For example:

In [181]: for item, frame in wp.iteritems():
.....:     print(item)
.....:     print(frame)

Item1
   A    B    C    D
2000-01-01 -1.118121  0.431279  0.554724 -1.333649
2000-01-02  0.332174 -0.485882  1.725945  1.799276
2000-01-03  0.968916 -0.779465 -2.000701 -1.866630
2000-01-04 -1.101268  1.957478  0.058889  0.758071
2000-01-05  0.076612  -0.548502  -0.160485  -0.377780
[5 rows x 4 columns]

Item2
A   B   C   D
2000-01-01 0.249911  -0.341270  -0.272599  -0.277446
2000-01-02 -1.102896   0.100307   1.602814   0.920139
2000-01-03 0.643870   0.060336  -0.434942  -0.787062
2000-01-04 0.737973   0.451632   0.334124   2.395763
2000-01-05 0.51396   -0.741919   1.193881  -2.395763
[5 rows x 4 columns]

9.8.2 iterrows

New in v0.7 is the ability to iterate efficiently through rows of a DataFrame. It returns an iterator yielding each index value along with a Series containing the data in each row:

In [182]: for row_index, row in df2.iterrows():
    ....:     print(‘%s

    ....:     %s’ % (row_index, row))
    ....:
a
    one   -0.701368
    two   -0.087103
    Name: a, dtype: float64

b
    one   0.109333
    two   0.637674
    Name: b, dtype: float64

c
    one   -0.231617
    two   -0.002666
    Name: c, dtype: float64

For instance, a contrived way to transpose the DataFrame would be:

In [183]: df2 = DataFrame({‘x’: [1, 2, 3], ‘y’: [4, 5, 6]})

In [184]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6
[3 rows x 2 columns]

In [185]: print(df2.T)
   0  1  2
x  1  2  3
y  4  5  6
[2 rows x 3 columns]

In [186]: df2_t = DataFrame(dict((idx,values) for idx, values in df2.iterrows()))

In [187]: print(df2_t)
   0  1  2
**9.8.3 itertuples**

This 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,

```python
In [192]: for r in df2.itertuples():
    print(r)
(0, 1, 4)
(1, 2, 5)
(2, 3, 6)
```

**9.9 Vectorized string methods**

Series is equipped (as of pandas 0.8.1) 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) build-in string methods:

**9.9.1 Splitting and Replacing Strings**

```python
In [194]: s.str.lower()
Out[194]:
0   a
1   b
2   c
3  aaba
4  baca
5  NaN
```
6     caba
7      dog
8      cat
dtype: object

In [195]: s.str.upper()
Out[195]:
0    A
1    B
2    C
3   AAAA
4   BACA
5      NaN
6     CABA
7     DOG
8     CAT
dtype: object

In [196]: s.str.len()
Out[196]:
0    1
1    1
2    1
3    4
4    4
5   NaN
6    4
7    3
8    3
dtype: float64

Methods like split return a Series of lists:

In [197]: s2 = Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [198]: s2.str.split('_')
Out[198]:
0    [a, b, c]
1    [c, d, e]
2       NaN
3    [f, g, h]
dtype: object

Elements in the split lists can be accessed using get or [] notation:

In [199]: s2.str.split('_').str.get(1)
Out[199]:
0    b
1    d
2   NaN
3    g
dtype: object

In [200]: s2.str.split('_').str[1]
Out[200]:
0    b
1    d
2   NaN
3    g
Methods like replace and.findall take regular expressions, too:

```python
In [201]: s3 = Series(['A', 'B', 'C', 'Aaba', 'Baca', .....:
        ' ', np.nan, 'CABA', 'dog', 'cat'])
......:

In [202]: s3
Out[202]:
0    A
1    B
2    C
3    Aaba
4    Baca
5    NaN
6    CABA
7    dog
8    cat
dtype: object
```

```python
In [203]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
Out[203]:
0      A
1      B
2      C
3    XX-XX ba
4    XX-XX ca
5      NaN
6    XX-XX BA
7    XX-XX
8    XX-XX
type: object
```

### 9.9.2 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.

```python
In [204]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
Out[204]:
0  a 1
1  b 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.

```python
In [205]: Series(['a1', 'b2', 'c3']).str.extract('(\[\d])(\d)')
Out[205]:
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.

Named groups like

```python
In [206]: Series(["a1", 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)')
```

```plaintext
Out[206]:
   letter  digit
0     a     1
1     b     2
2   NaN   NaN
```

and optional groups like

```python
In [207]: Series(["a1", 'b2', '3']).str.extract('(?P<letter>[ab])?(?P<digit>\d)')
```

```plaintext
Out[207]:
   letter  digit
0     a     1
1     b     2
2   NaN     3
```

can also be used.

### 9.9.3 Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

```python
In [208]: pattern = r'\[a-z]\[0-9]'
```

```python
In [209]: Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
```

```plaintext
Out[209]:
0  False
1  False
2  False
3  False
4  False
dtype: bool
```

or match a pattern:

```python
In [210]: Series(['1', '2', '3a', '3b', '03c']).str.match(pattern, as_indexer=True)
```

```plaintext
Out[210]:
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:

In [211]: s4 = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [212]: s4.str.contains('A', na=False)
Out[212]:
0    True
1     False
2     False
3     True
4     False
5     False
6     True
7     False
8     False
dtype: bool

<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>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>
<tr>
<td><code>center</code></td>
<td>Equivalent to <code>pad(side='both')</code></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>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>
</tbody>
</table>

### 9.9.4 Getting indicator variables from seperated strings

You can extract dummy variables from string columns. For example if they are seperated by a `' | '`: 
In [213]: s = pd.Series(["a", "a|b", np.nan, "a|c"])

In [214]: s.str.get_dummies(sep='|')
Out[214]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1

[4 rows x 3 columns]

See also get_dummies().

9.10 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 [215]: unsorted_df = df.reindex(index=["a", "d", "c", "b"],
                             columns=["three", "two", "one"])

In [216]: unsorted_df.sort_index()
Out[216]:
   three  two  one
a  NaN    NaN    NaN
b -0.354359 0.637674 0.109333
c -0.148387 -0.002666 -0.231617
d -0.167407 0.104044  NaN

[4 rows x 3 columns]

In [217]: unsorted_df.sort_index(ascending=False)
Out[217]:
   three  two  one
a  NaN    NaN    NaN
b -0.354359 0.637674 0.109333
c -0.148387 -0.002666 -0.231617
d -0.167407 0.104044  NaN

[4 rows x 3 columns]

In [218]: unsorted_df.sort_index(axis=1)
Out[218]:
   one  three  two
a -0.701368  NaN  -0.087103
b  NaN   -0.167407 0.104044
c -0.231617 -0.148387 -0.002666
d  0.109333 -0.354359 0.637674

[4 rows x 3 columns]

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 [219]: df1 = DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})

In [220]: df1.sort_index(by='two')
Out[220]:
   one  three  two
0   2      5    1
1   1      4    3
2   1      2    4
3   1      3    2

[4 rows x 3 columns]

The `by` argument can take a list of column names, e.g.:

In [221]: df1[['one', 'two', 'three']].sort_index(by=['one','two'])
Out[221]:
   one  two  three
2   1    2    3
1   1    3    4
3   1    4    2
0   2    1    5

[4 rows x 3 columns]

Series has the method `order` (analogous to R's `order` function) which sorts by value, with special treatment of NA values via the `na_last` argument:

In [222]: s[2] = np.nan

In [223]: s.order()
Out[223]:
0    a
1    a
3    a
2  NaN
dtype: object

In [224]: s.order(na_last=False)
Out[224]:
2  NaN
0    a
1    a
3    a
dtype: object

Some other sorting notes / nuances:

- `Series.sort` sorts a Series by value in-place. This is to provide compatibility with NumPy methods which expect the `ndarray.sort` behavior.
- `DataFrame.sort` takes a `column` argument instead of `by`. This method will likely be deprecated in a future release in favor of just using `sort_index`.

### 9.11 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.

9.12 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 [225]: dft = DataFrame(dict( A = np.random.rand(3),
                     .....:             B = 1,
                     .....:             C = 'foo',
                     .....:             D = Timestamp('20010102'),
                     .....:             E = Series([1.0]*3).astype('float32'),
                     .....:             F = False,
                     .....:             G = Series([1]*3,dtype='int8')))  
In [226]: dft
Out[226]:
         A      B       C         D          E             F          G
0  0.298496  1  foo 2001-01-02  1.0 False 1
1  0.347135  1  foo 2001-01-02  1.0 False 1
2  0.141330  1  foo 2001-01-02  1.0 False 1

[3 rows x 7 columns]
```

```python
In [227]: dft.dtypes
Out[227]:
     A    B      C     D     E      F  G
float64  int64 object  datetime64[ns]  float32  bool int8

dtype: object
```

On a Series use the dtype method.

```python
In [228]: dft['A'].dtype
Out[228]: dtype('float64')
```

If a pandas object contains data multiple dtypes IN A SINGLE COLUMN, the dtype of the column will be chosen to accommodate all of the data types (object is the most general).

```python
# these ints are coerced to floats
In [229]: Series([1, 2, 3, 4, 5, 6.])
Out[229]:
     0  1
```

9.12. dtypes
# string data forces an 'object' dtype
In [230]: Series([1, 2, 3, 6., 'foo'])
Out[230]:
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 [231]: dft.get_dtype_counts()
Out[231]:
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 [232]: df1 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')

In [233]: df1
Out[233]:
   A
0  1.111528
1 -1.189947
2 -0.652389
3 -0.216515
4 -2.163590
5  0.117168
6 -1.806383
7 -0.249113

[8 rows x 1 columns]

In [234]: df1.dtypes
Out[234]:
A  float32
dtype: object

In [235]: df2 = DataFrame(dict( A = Series(randn(8),dtype='float16'),
......: B = Series(randn(8)),
......: C = Series(np.array(randn(8),dtype='uint8')) ))

......:
In [236]: df2
Out[236]:
A   B   C
0  1.109  -0.265  0
1  0.748  -1.842   0
2 -1.319  -0.662   0
3  2.006   1.199   0
4  0.260   0.138   254
5  1.785   0.968   0
6  0.481   0.447    2
7  0.319  -0.361    2
[8 rows x 3 columns]
In [237]: df2.dtypes
Out[237]:
A  float16
B  float64
C  uint8
dtype: object

9.12.1 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 [238]: DataFrame([1, 2], columns=['a']).dtypes
Out[238]:
a  int64
dtype: object

In [239]: DataFrame({'a': [1, 2]}).dtypes
Out[239]:
a  int64
dtype: object

In [240]: DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[240]:
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 [241]: frame = DataFrame(np.array([1, 2]))

9.12.2 upcasting

Types can potentially be upcasted when combined with other types, meaning they are promoted from the current type (say int to float)

In [242]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [243]: df3
Out[243]:

9.12. dtypes
A B C
0 2.220903 -0.265597 0
1 -0.441412 -1.841820 0
2 -1.971725 -0.661921 0
3 1.789344 1.198778 0
4 -1.903092 0.138315 254
5 1.902324 0.967672 0
6 -1.324694 0.447494 2
7 0.070223 -0.361367 2

[8 rows x 3 columns]

In [244]: df3.dtypes
Out[244]:
A  float32
B  float64
C  float64
dtype: object

The `values` attribute on a DataFrame return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate ALL of the types in the resulting homogenous dtyped numpy array. This can force some upcasting.

In [245]: df3.values.dtype
Out[245]: dtype('float64')

9.12.3 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.

In [246]: df3
Out[246]:
A  B  C
0 2.220903 -0.265597 0
1 -0.441412 -1.841820 0
2 -1.971725 -0.661921 0
3 1.789344 1.198778 0
4 -1.903092 0.138315 254
5 1.902324 0.967672 0
6 -1.324694 0.447494 2
7 0.070223 -0.361367 2

[8 rows x 3 columns]

In [247]: df3.dtypes
Out[247]:
A  float32
B  float64
C  float64
dtype: object

# conversion of dtypes
In [248]: df3.astype('float32').dtypes

214 Chapter 9. Essential Basic Functionality
9.12.4 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 [249]: df3['D'] = '1.'

In [250]: df3['E'] = '1'

In [251]: df3.convert_objects(convert_numeric=True).dtypes
Out[251]:
A float32
B float64
C float64
D float64
E int64
dtype: object

# same, but specific dtype conversion
In [252]: df3['D'] = df3['D'].astype('float16')

In [253]: df3['E'] = df3['E'].astype('int32')

In [254]: df3.dtypes
Out[254]:
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 [255]: s = Series([datetime(2001,1,0,0),
       .....:   'foo', 1.0, 1, Timestamp('20010104'),
       .....:   '20010105'],dtype='O')

In [256]: s
Out[256]:
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 [257]: s.convert_objects(convert_dates='coerce')
Out[257]:
0   2001-01-01
1         NaT
2         NaT
3  2001-01-04
4   2001-01-05
dtypes: datetime64[ns]

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.

9.12.5 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 [258]: dfi = df3.astype('int32')

In [259]: dfi['E'] = 1

In [260]: dfi
Out[260]:
     A    B    C    D    E
0  2.0  0.0  0.0  1.0  1.0
1  0.0 -1.0  0.0  1.0  1.0
2 -1.0  0.0  0.0  1.0  1.0
3  1.0  1.0  0.0  1.0  1.0
4 -1.0  0.0  254.0  1.0  1.0
5  1.0  0.0  0.0  1.0  1.0
6 -1.0  0.0  2.0  1.0  1.0
7  0.0  0.0  2.0  1.0  1.0
[8 rows x 5 columns]

In [261]: dfi.dtypes
Out[261]:
A    int32
B    int32
C    int32
D    int32
E    int64
dtype: object

In [262]: casted = dfi[dfi>0]

In [263]: casted
Out[263]:
     A    B    C    D    E
0  NaN  NaN  NaN  1.0  1.0
1  NaN  NaN  NaN  1.0  1.0
2  NaN  NaN  NaN  1.0  1.0
3  1.0  1.0  NaN  1.0  1.0
4  NaN  NaN  254.0  1.0  1.0
5  1  NaN  NaN  1  1
6  NaN  NaN  2  1  1
7  NaN  NaN  2  1  1

[8 rows x 5 columns]

In [264]: casted.dtypes
Out[264]:
A    float64
B    float64
C    float64
D     int32
E     int64
dtype: object

While float dtypes are unchanged.

In [265]: dfa = df3.copy()

In [266]: dfa['A'] = dfa['A'].astype('float32')

In [267]: dfa.dtypes
Out[267]:
A    float32
B    float64
C    float64
D    float16
E     int32
dtype: object

In [268]: casted = dfa[df2>0]

In [269]: casted
Out[269]:
      A    B     C     D     E
0  2.220903  NaN    NaN    NaN    NaN
1 -0.441412  NaN    NaN    NaN    NaN
2  NaN  NaN    NaN    NaN    NaN
3  1.789344  1.198778  NaN    NaN    NaN
4 -1.903092  0.138315  254  NaN    NaN
5  1.902324  0.967672  NaN    NaN    NaN
6 -1.324694  0.447494    2  NaN    NaN
7  0.070223  NaN    2    NaN    NaN

[8 rows x 5 columns]

In [270]: casted.dtypes
Out[270]:
A    float32
B    float64
C    float64
D    float16
E    float64
dtype: object
9.13 Working with package options

New in version 0.10.1. 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 [271]: import pandas as pd
In [272]: pd.options.display.max_rows
Out[272]: 15
In [273]: pd.options.display.max_rows = 999
In [274]: pd.options.display.max_rows
Out[274]: 999
```

There is also an API composed of 4 relevant functions, available directly from the `pandas` namespace, and they are:

- `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.

**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 [275]: get_option("display.max_rows")
Out[275]: 999
In [276]: set_option("display.max_rows",101)
In [277]: get_option("display.max_rows")
Out[277]: 101
In [278]: set_option("max_r",102)
In [279]: get_option("display.max_rows")
Out[279]: 102
```

The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```python
In [280]: try:
   ....:     get_option("display.max_")
   ....:     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.
In [281]: describe_option()

display.chop_threshold: [default: None] [currently: None]
    : 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.
display.colheader_justify: [default: right] [currently: right]
    : 'left'/'right'
        Controls the justification of column headers. used by DataFrameFormatter.
display.column_space: [default: 12] [currently: 12]
    No description available.
display.date_dayfirst: [default: False] [currently: False]
    : boolean
        When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst: [default: False] [currently: False]
    : boolean
        When True, prints and parses dates with the year first, eg 2005/01/20
display.encoding: [default: UTF-8] [currently: UTF-8]
    : 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.
display.expand_frame_repr: [default: True] [currently: True]
    : 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 it’s width exceeds 'display.width'.
display.float_format: [default: None] [currently: None]
    : 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.
display.height: [default: 60] [currently: 60]
    : int
        Deprecated.
        (Deprecated, use 'display.max_rows' instead.)
display.large_repr: [default: truncate] [currently: truncate]
    : '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).
display.line_width: [default: 80] [currently: 80]
    : int
        Deprecated.
        (Deprecated, use 'display.width' instead.)
display.max_columns: [default: 20] [currently: 20]
    : int
        max_rows and max_columns are used in __repr__() methods to decide if
to_string() or info() is used to render an object to a string. In case
python/IPython is running in a terminal this can be set to 0 and pandas
will correctly auto-detect the width the terminal and swap to a smaller
format in case all columns would not fit vertically. The IPython notebook,
IPython qtconsole, or IDLE do not run in a terminal and hence it is not
possible to do correct auto-detection.
    'None' value means unlimited.

9.13. Working with package options 219
display.max_colwidth: [default: 50] [currently: 50] : 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.
display.max_info_columns: [default: 100] [currently: 100] : int
max_info_columns is used in DataFrame.info method to decide if per column information will be printed.
display.max_info_rows: [default: 1690785] [currently: 1690785] : 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.
display.max_rows: [default: 60] [currently: 60] : int
This sets the maximum number of rows pandas should output when printing out various output. For example, this value determines whether the repr() for a dataframe prints out fully or just a summary repr. 'None' value means unlimited.
display.max_seq_items: [default: 100] [currently: 100] : 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.
display.mpl_style: [default: None] [currently: None] : 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.
display.multi_sparse: [default: True] [currently: True] : boolean
"sparsify" MultiIndex display (don’t display repeated elements in outer levels within groups)
display.notebook_repr_html: [default: True] [currently: True] : boolean
When True, IPython notebook will use html representation for pandas objects (if it is available).
display.pprint_nest_depth: [default: 3] [currently: 3] : int
Controls the number of nested levels to process when pretty-printing
display.precision: [default: 7] [currently: 7] : int
Floating point output precision (number of significant digits). This is only a suggestion
display.show_dimensions: [default: True] [currently: True] : boolean
Whether to print out dimensions at the end of DataFrame repr.
display.width: [default: 80] [currently: 80] : 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.

`io.excel.xls.writer`: [default: xlwt] [currently: xlwt]
- string
  - The default Excel writer engine for 'xls' files. Available options:
    - 'xlwt' (the default).

`io.excel.xlsm.writer`: [default: openpyxl] [currently: openpyxl]
- string
  - The default Excel writer engine for 'xlsm' files. Available options:
    - 'openpyxl' (the default).

`io.excel.xlsx.writer`: [default: xlsxwriter] [currently: xlsxwriter]
- string
  - The default Excel writer engine for 'xlsx' files. Available options:
    - 'xlsxwriter' (the default), 'openpyxl'.

`io.hdf.default_format`: [default: None] [currently: None]
- format
  - default format writing format, if None, then
    put will default to 'fixed' and append will default to 'table'

`io.hdf.dropna_table`: [default: True] [currently: True]
- boolean
  - drop ALL nan rows when appending to a table

`mode.chained_assignment`: [default: warn] [currently: warn]
- string
  - Raise an exception, warn, or no action if trying to use chained assignment,
  The default is warn

`mode.sim_interactive`: [default: False] [currently: False]
- boolean
  - Whether to simulate interactive mode for purposes of testing

`mode.use_inf_as_null`: [default: False] [currently: False]
- 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).

or you can get the description for just the options that match the regexp you pass in:

```python
In [282]: describe_option("date")
```

`display.date_dayfirst`: [default: False] [currently: False]
- boolean
  - When True, prints and parses dates with the day first, eg 20/01/2005

`display.date_yearfirst`: [default: False] [currently: False]
- boolean
  - When True, prints and parses dates with the year first, eg 2005/01/20

All options also have a default value, and you can use the `reset_option` to do just that:

```python
In [283]: get_option("display.max_rows")
Out[283]: 60

In [284]: set_option("display.max_rows", 999)

In [285]: get_option("display.max_rows")
Out[285]: 999

In [286]: reset_option("display.max_rows")

In [287]: get_option("display.max_rows")
Out[287]: 60
```

It’s also possible to reset multiple options at once (using a regex):

```
In [288]: reset_option(pattern="display")[0:5]
Out[288]:
```

### 9.13. Working with package options
In [288]: reset_option("^display")

New in version 0.13.1.

In [289]: with option_context("display.max_rows",10,"display.max_columns", 5):
....:     print get_option("display.max_rows")
....:     print get_option("display.max_columns")
10
20

In [290]: print get_option("display.max_columns")
20

In [291]: print get_option("display.max_rows")
60

In [292]: print get_option("display.max_columns")
20

9.14 Console Output Formatting

Use the set_eng_float_format function in the pandas.core.common module to alter the floating-point formatting of pandas objects to produce a particular format.

For instance:

In [293]: set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [294]: s = Series(randn(5), index=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’])

In [295]: s/1.e3
Out[295]:
  a  -184.526u
  b  -615.011u
  c   104.861u
  d  -524.434u
  e    5.385u
dtype: float64

In [296]: s/1.e6
Out[296]:
  a  -184.526n
  b  -615.011n
  c   104.861n
  d  -524.434n
  e    5.385n
dtype: float64

The set_printoptions function has a number of options for controlling how floating point numbers are formatted (using the precision argument) in the console and . The max_rows and max_columns control how many rows and columns of DataFrame objects are shown by default. If max_columns is set to 0 (the default, in fact), the library will attempt to fit the DataFrame’s string representation into the current terminal width, and defaulting to the summary view otherwise.
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 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 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

See the cookbook for some advanced strategies

10.1 Different Choices for Indexing (loc, iloc, and ix)

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 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

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*, *Advanced Hierarchical* and *Fallback Indexing*

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><code>s.loc[indexer]</code></td>
</tr>
<tr>
<td>DataFrame</td>
<td><code>df.loc[row_indexer, column_indexer]</code></td>
</tr>
<tr>
<td>Panel</td>
<td><code>p.loc[item_indexer, major_indexer, minor_indexer]</code></td>
</tr>
</tbody>
</table>

### 10.2 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.

### 10.3 Basics

As mentioned when introducing the data structures in the *last section*, the primary function of indexing with `[]` (a.k.a. `__getitem__` for those familiar with implementing class behavior in Python) is selecting out lower-dimensional slices. Thus,

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Selection</th>
<th>Return Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td><code>series[label]</code></td>
<td>scalar value</td>
</tr>
<tr>
<td>DataFrame</td>
<td><code>frame[colname]</code></td>
<td>Series corresponding to colname</td>
</tr>
<tr>
<td>Panel</td>
<td><code>panel[itemname]</code></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>-2.77046</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>

[8 rows x 4 columns]

In [4]: panel = Panel({'one' : df, 'two' : df - df.mean()})

In [5]: panel
Out[5]:

Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D

Note: None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using []:

In [6]: s = df['A']

In [7]: s[dates[5]]
Out[7]: -0.67368970808837059

In [8]: panel['two']
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>
<tr>
<td>2000-01-08</td>
<td>-0.430188</td>
<td>-0.761943</td>
<td>-0.446079</td>
<td>1.044010</td>
</tr>
</tbody>
</table>

[8 rows x 4 columns]

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:
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

[8 rows x 4 columns]

In [10]: df[['B', 'A']] = df[['A', 'B']]

In [11]: df
Out[11]:
  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
2000-01-04 -0.706771  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885

[8 rows x 4 columns]

You may find this useful for applying a transform (in-place) to a subset of the columns.

10.4 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]:
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
2000-01-06    0.113648
2000-01-07    0.577046
2000-01-08   -1.157892
Freq: D, Name: A, dtype: float64

In [16]: panel.one
Out[16]:
  A   B   C   D
2000-01-01 0.469112 -0.282863 -1.509059 -1.135632

Chapter 10. Indexing and Selecting Data
Setting is allowed as well

In [17]: sa.a = 5

In [18]: sa
Out[18]:
   a 5
   b 2
   c 3
dtype: int64

In [19]: dfa.A = list(range(len(dfa.index)))

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 0.276232 -1.087401
   2000-01-06 5 -0.673690 0.113648 -1.478427 0.524988
   2000-01-07 6 0.404705 0.577046 -1.715002 -1.039268
   2000-01-08 7 -0.370647 -1.157892 -1.344312 0.844885

[8 rows x 4 columns]

Warning:
- You can use this access only if the index element is a valid python identifier, e.g. s.1 is not allowed. see here for an explanation of valid identifiers.
- The attribute will not be available if it conflicts with an existing method name, e.g. s.min is not allowed.
- 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.

10.5 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:

In [21]: s[:5]
Out[21]:
2000-01-01 -0.282863
2000-01-02 -0.173215
2000-01-03 -0.193215
2000-01-04 0.404705
2000-01-05 0.577046
2000-01-03  -2.104569
2000-01-04   -0.706771
2000-01-05    0.567020
Freq: D, Name: A, dtype: float64

In [22]: s[::2]
Out[22]:
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

In [23]: s[::-1]
Out[23]:
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:

In [24]: s2 = s.copy()

In [25]: s2[:5] = 0

In [26]: s2
Out[26]:
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.

In [27]: df[:3]
Out[27]:
   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

[3 rows x 4 columns]

In [28]: df[::-1]
Out[28]:
   A      B      C      D
2000-01-08 -1.157892 -0.370647 -1.344312  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 -0.282863  0.469112 -1.509059 -1.135632

[8 rows x 4 columns]

10.6  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

Pandas provides a suite of methods in order to have purely label based indexing. This is a strict inclusion based protocol. ALL of the labels for which you ask, must be in the index or a KeyError will be raised! When slicing, 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 [29]: s1 = Series(np.random.randn(6),index=list(’abcdef’))

In [30]: s1
Out[30]:
a  1.075770
b -0.109050
c  1.643563
d -1.469388
e  0.357021
f -0.674600
dtype: float64

In [31]: s1.loc[’c’ :]
Out[31]:
c  1.643563
d -1.469388
e  0.357021
f -0.674600
dtype: float64

In [32]: s1.loc[’b’]
Out[32]: -0.1090499752802223

Note that setting works as well:
In [33]: s1.loc['c'] = 0

In [34]: s1
Out[34]:
   a    1.07577
   b   -0.10905
   c    0.00000
   d    0.00000
   e    0.00000
   f    0.00000
dtype: float64

With a DataFrame

In [35]: df1 = DataFrame(np.random.randn(6,4),
                  index=list('abcdef'),
                  columns=list('ABCD'))

In [36]: df1
Out[36]:
     A         B         C         D
   a -1.77690 -0.968914 -1.294524  0.413738
   b  0.27666 -0.472035 -0.013960 -0.362543
   c -0.00615 -0.923061  0.895717  0.805244
   d -1.20641  2.565646  1.431256  1.340309
   e -1.17029 -0.226169  0.410835  0.813850
   f  0.13200 -0.827317 -0.076467 -1.187678

[6 rows x 4 columns]

In [37]: df1.loc[['a','b','d'],:]
Out[37]:
     A         B         C         D
   a -1.77690 -0.968914 -1.294524  0.413738
   b  0.27666 -0.472035 -0.013960 -0.362543
   d -1.20641  2.565646  1.431256  1.340309

[3 rows x 4 columns]

Accessing via label slices

In [38]: df1.loc['d':'A':'C']
Out[38]:
     A         B         C
   d -1.20641  2.565646  1.431256
   e -1.17029 -0.226169  0.410835
   f  0.13200 -0.827317 -0.076467

[3 rows x 3 columns]

For getting a cross section using a label (equiv to df.xs('a'))

In [39]: df1.loc['a']
Out[39]:
   A    -1.77690
   B   -0.968914
   C   -1.294524
   D     0.413738
Name: a, dtype: float64

For getting values with a boolean array

```
In [40]: df1.loc['a']>0
Out[40]:
   a
A  False
B  False
C  False
D  True
Name: a, dtype: bool
```

```
In [41]: df1.loc[:,df1.loc['a']>0]
Out[41]:
    a
D  0.413738
   b  0.805244
   c  1.340309
   d -1.187678
   e  0.813850
   f  1.340309
[6 rows x 1 columns]
```

For getting a value explicitly (equiv to deprecated `df.get_value('a','A')`)

```
# this is also equivalent to `df1.at['a','A']`
In [42]: df1.loc['a','A']
Out[42]: -1.7769037169718671
```

### 10.7 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`

```
In [43]: s1 = Series(np.random.randn(5),index=list(range(0,10,2)))
In [44]: s1
Out[44]:
0  1.130127
2 -1.436737
4 -1.413681
6  1.607920
8  1.024180
dtype: float64
```

```
In [45]: s1.iloc[0]
Out[45]: 1.130127
```
In [45]: s1.iloc[:3]
Out[45]:
0  1.130127  
2 -1.436737  
4 -1.413681  
dtype: float64

In [46]: s1.iloc[3]
Out[46]: 1.6079204745847746

Note that setting works as well:

In [47]: s1.iloc[:3] = 0
In [48]: s1
Out[48]:
0  0.00000  
2  0.00000  
4  0.00000  
6  1.60792  
8  1.02418  
dtype: float64

With a DataFrame

In [49]: df1 = DataFrame(np.random.randn(6,4),
    index=list(range(0,12,2)),
    columns=list(range(0,8,2)))

In [50]: df1
Out[50]:
      0    2    4    6
0  0.569605  0.875906 -2.211372  0.974466
2 -2.006747 -0.410001 -0.078638  0.545952
4 -1.219217 -1.226825  0.769804 -1.281247
6 -0.727707 -0.121306 -0.097883  0.695775
8  0.341734  0.959726 -1.110336 -0.619976
10 0.149748 -0.732339  0.687738  0.176444

[6 rows x 4 columns]

Select via integer slicing

In [51]: df1.iloc[:3]
Out[51]:
      0    2    4    6
0  0.569605  0.875906 -2.211372  0.974466
2 -2.006747 -0.410001 -0.078638  0.545952
4 -1.219217 -1.226825  0.769804 -1.281247

[3 rows x 4 columns]

In [52]: df1.iloc[1:5,2:4]
Out[52]:
       4          6
2  -0.078638  0.545952
4   0.769804 -1.281247
6  -0.097883  0.695775

Chapter 10. Indexing and Selecting Data
Select via integer list

In [53]: df1.iloc[[1,3,5],[1,3]]
Out[53]:
   2   6
   2 -0.410001  0.545952
   6 -0.121306  0.695775
  10 -0.732339  0.176444

[3 rows x 2 columns]

For slicing rows explicitly (equiv to deprecated df.irow(slice(1,3))).

In [54]: df1.iloc[1:3,:]
Out[54]:
     0     2     4     6
   2 -2.006747 -0.410001 -0.078638  0.545952
   4 -1.219217 -1.226825  0.769804 -1.281247

[2 rows x 4 columns]

For slicing columns explicitly (equiv to deprecated df.icol(slice(1,3))).

In [55]: df1.iloc[:,1:3]
Out[55]:
   2  4
   0  0.875906 -2.211372
   2 -0.410001 -0.078638
   4 -1.226825  0.769804
   6 -0.121306 -0.097883
   8  0.959726 -1.110336
  10 -0.732339  0.687738

[6 rows x 2 columns]

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 [56]: df1.iloc[1,1]
Out[56]: -0.41000056806065832

For getting a cross section using an integer position (equiv to df.xs(1))

In [57]: df1.iloc[1]
Out[57]:
     0     2     4     6
   2 -2.006747 -0.410001 -0.078638  0.545952
   4 -1.219217 -1.226825  0.769804 -1.281247
   6 -0.121306 -0.097883
   8  0.959726 -1.110336
  10 -0.732339  0.687738
Name: 2, dtype: float64

There is one significant departure from standard python/numpy slicing semantics. python/numpy allow slicing past the end of an array without an associated error.

# these are allowed in python/numpy.
In [58]: x = list('abcdef')
In [59]: x[4:10]
Out[59]: ['e', 'f']

In [60]: x[8:10]
Out[60]: []

Pandas will detect this and raise IndexError, rather than return an empty structure.

```python
>>> df.iloc[:,3:6]
IndexError: out-of-bounds on slice (end)
```

## 10.8 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 [61]: se = Series([1,2,3])

In [62]: se
Out[62]:
0  1
1  2
2  3
dtype: int64

In [63]: se[5] = 5.

In [64]: se
Out[64]:
0  1
1  2
2  3
5  5
dtype: float64

A DataFrame can be enlarged on either axis via `.loc`

In [65]: dfi = DataFrame(np.arange(6).reshape(3,2),
....:                  columns=['A','B'])
....:

In [66]: dfi
Out[66]:
   A  B
0  0  1
1  2  3
2  4  5

[3 rows x 2 columns]

In [67]: dfi.loc[:,'C'] = dfi.loc[:,'A']

In [68]: dfi
Out[68]:
   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 on the DataFrame.

In [69]: dfi.loc[3] = 5

In [70]: dfi
Out[70]:
   A  B  C
0  0  0  0
1  2  3  2
2  4  5  4
3  5  5  5

[4 rows x 3 columns]

10.9 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.

Similary to loc, at provides label based scalar lookups, while, iat provides integer based lookups analagously to iloc

In [71]: s.iat[5]
Out[71]: 0.1136484096888855

In [72]: df.at[dates[5], 'A']
Out[72]: 0.1136484096888855

In [73]: df.iat[3, 0]
Out[73]: -0.70677113363008448

You can also set using these same indexers.

In [74]: df.at[dates[5], 'E'] = 7

In [75]: df.iat[3, 0] = 7

at may enlarge the object in-place as above if the indexer is missing.

In [76]: df.at[dates[-1]+1, 0] = 7

In [77]: df
Out[77]:
   A     B     C     D     E
0 0.000000 -0.282863 0.469112 -1.509059 -1.135632 NaN NaN
1 0.000000 -0.173215 1.212112  0.119209 -1.044236 NaN NaN
2 0.000000 -2.104569 -0.861849 -0.494929  1.071804 NaN NaN
3 0.000000  7.000000  0.721555 -1.039575  0.271860 NaN NaN
4 0.000000  0.567020 -0.424972  0.276232 -1.087401 NaN NaN

10.9. Fast scalar value getting and setting
2000-01-06  0.113648  -0.673690  -1.478427  0.524988  7  NaN
2000-01-07  0.577046  0.404705  -1.715002  -1.039268  NaN  NaN
2000-01-08  -1.157892  -0.370647  -1.344312  0.844885  NaN  NaN
2000-01-09  NaN  NaN  NaN  NaN  NaN  7
[9 rows x 6 columns]

10.10 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: \texttt{|} for \texttt{or}, \texttt{&} for \texttt{and}, and \texttt{~} for \texttt{not}. These \textbf{must} be grouped by using parentheses.

Using a boolean vector to index a Series works exactly as in a numpy ndarray:

\begin{verbatim}
In [78]: s[s > 0]
Out [78]:
2000-01-05  0.567020
2000-01-06  0.113648
2000-01-07  0.577046
Freq: D, Name: A, dtype: float64

In [79]: s[(s < 0) & (s > -0.5)]
Out [79]:
2000-01-01  -0.282863
2000-01-02  -0.173215
Freq: D, Name: A, dtype: float64

In [80]: s[(s < -1) | (s > 1)]
Out [80]:
2000-01-03  -2.104569
2000-01-08  -1.157892
Name: A, dtype: float64

In [81]: s[~(s < 0)]
Out [81]:
2000-01-05  0.567020
2000-01-06  0.113648
2000-01-07  0.577046
Freq: D, Name: A, dtype: float64
\end{verbatim}

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):

\begin{verbatim}
In [82]: df[df['A'] > 0]
Out [82]:
          A          B          C          D          E
2000-01-04 7.000000  0.721555  -1.039575  0.271860  NaN  NaN
2000-01-05 0.567020  -0.424972  0.276232  -1.087401  NaN  NaN
2000-01-06 0.113648  -0.673690  -1.478427  0.524988  7  NaN
2000-01-07 0.577046  0.404705  -1.715002  -1.039268  NaN  NaN
[4 rows x 6 columns]
\end{verbatim}

List comprehensions and \texttt{map} method of Series can also be used to produce more complex criteria:

\begin{verbatim}
In [83]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   ....:     'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   ....:

[4 rows x 6 columns]
\end{verbatim}
# only want 'two' or 'three'
In [84]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [85]: df2[criterion]
Out[85]:
   a  b  c
2  two  y  0.301624
3  three  x -2.179861
4  two  y -1.369849

[3 rows x 3 columns]

# equivalent but slower
In [86]: df2[[x.startswith('t') for x in df2['a']]]
Out[86]:
   a  b  c
2  two  y  0.301624
3  three  x -2.179861
4  two  y -1.369849

[3 rows x 3 columns]

# Multiple criteria
In [87]: df2[criterion & (df2['b'] == 'x')]
Out[87]:
   a  b  c
3  three  x -2.179861

[1 rows x 3 columns]

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 [88]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
Out[88]:
   b  c
3  x -2.179861

[1 rows x 2 columns]

### 10.10.1 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 [89]: s = Series(np.arange(5),index=np.arange(5)[::-1],dtype='int64')

In [90]: s
Out[90]:
   4  0
   3  1
   2  2
   1  3
   0  4
In [91]: s.isin([2, 4])
Out[91]:
4   False
3   False
2   True
1   False
0   True
dtype: bool

In [92]: s[s.isin([2, 4])]
Out[92]:
2
0

dtype: int64

DataFrame also has an `isin` method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

In [93]: df = DataFrame({'vals': [1, 2, 3, 4], 'ids': [‘a’, ‘b’, ‘f’, ‘n’],
    ....:     ‘ids2’: [‘a’, ‘n’, ‘c’, ‘n’]})

In [94]: values = [‘a’, ‘b’, 1, 3]

In [95]: df.isin(values)
Out[95]:
   ids  ids2  vals
0  True   True   True
1  True  False  False
2 False  False   True
3 False  False  False

[4 rows x 3 columns]

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 [96]: values = {‘ids’: [‘a’, ‘b’], ‘vals’: [1, 3]}

In [97]: df.isin(values)
Out[97]:
   ids  ids2  vals
0  True False   True
1  True False  False
2 False False   True
3 False False  False

[4 rows x 3 columns]

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 [98]: values = {‘ids’: [‘a’, ‘b’], ‘ids2’: [‘a’, ‘c’], ‘vals’: [1, 3]}

In [99]: row_mask = df.isin(values).all(1)
In [100]: df[row_mask]
Out[100]:
   ids  ids2  vals
0    a      a    1
[1 rows x 3 columns]

10.11 The \texttt{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 \texttt{where} method in \texttt{Series} and \texttt{DataFrame}.

To return only the selected rows

In [101]: s[s > 0]
Out[101]:
   3    1
   2    2
   1    3
  0    4
dtype: int64

To return a Series of the same shape as the original

In [102]: s.where(s > 0)
Out[102]:
   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. \texttt{where} is used under the hood as the implementation. Equivalent is \texttt{df.where(df < 0)}

In [103]: df[df < 0]
Out[103]:
   A       B       C       D
2000-01-01 -1.743161 -0.826591 -0.345352 NaN
2000-01-02  NaN    NaN    NaN   NaN
2000-01-03  NaN -0.317441 -1.236269 NaN
2000-01-04 -0.487602 -0.082240 -2.182937 NaN
2000-01-05  NaN    NaN    NaN  -0.493662
2000-01-06  NaN    NaN    NaN  -0.023688
2000-01-07  NaN    NaN    NaN  -0.251905
2000-01-08 -2.213588    NaN    NaN   NaN
[8 rows x 4 columns]

In addition, \texttt{where} takes an optional \texttt{other} argument for replacement of values where the condition is False, in the returned copy.

In [104]: df.where(df < 0, -df)
Out[104]:
   A       B       C       D
2000-01-01 -1.743161 -0.826591 -0.345352 -1.314232
You may wish to set values based on some boolean criteria. This can be done intuitively like so:

In [105]: s2 = s.copy()

In [106]: s2[s2 < 0] = 0

In [107]: s2
Out[107]:
   0  1  2  3  4
0  4  1  2  3  4
dtype: int64

In [108]: df2 = df.copy()

In [109]: df2[df2 < 0] = 0

In [110]: df2
Out[110]:
    A      B      C      D
2000-01-01 0.000000 0.000000 0.000000 1.314232
2000-01-02 0.690579 0.995761 2.396780 0.014871
2000-01-03 3.357427 0.000000 0.000000 0.896171
2000-01-04 0.000000 0.000000 0.000000 0.380396
2000-01-05 0.084844 0.432390 1.519970 0.000000
2000-01-06 0.600178 0.274230 0.206053 0.251905
2000-01-07 2.410179 1.450520 0.206053 0.251905
2000-01-08 0.000000 1.063327 1.266143 0.299368

[8 rows x 4 columns]

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 [111]: df_orig = df.copy()

In [112]: df_orig.where(df > 0, -df, inplace=True);

In [113]: df_orig
Out[113]:
    A      B      C      D
2000-01-01 1.743161 0.826591 0.345352 1.314232
2000-01-02 0.690579 0.995761 2.396780 0.014871
2000-01-03 3.357427 0.317441 1.236269 0.896171
2000-01-04 0.487602 0.082240 2.182937 0.380396
2000-01-05 0.084844 0.432390 1.519970 0.493662
2000-01-06 0.600178 0.274230 0.132885 0.023688

[8 rows x 4 columns]
Furthermore, \texttt{where} aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via \texttt{.ix} (but on the contents rather than the axis labels).

\begin{verbatim}
In [114]: df2 = df.copy()

In [115]: df2[ df2[1:4] > 0 ] = 3

In [116]: df2
Out[116]:
A  B  C  D
2000-01-01 -1.743161 -0.826591 -0.345352 1.314232
2000-01-02  3.000000  3.000000  3.000000  3.000000
2000-01-03  3.000000 -0.317441 -1.236269  0.299368
2000-01-04 -0.487602 -0.082240 -2.182937  0.299368
2000-01-05  0.084844  0.432390  1.519970 -0.251905
2000-01-06  0.600178  0.274230  0.132885 -0.023688
2000-01-07  2.410179  1.450520  0.206053 -0.251905
2000-01-08 -2.213588  1.063327  1.266143  0.299368

[8 rows x 4 columns]
\end{verbatim}

New in version 0.13. Where can also accept \texttt{axis} and \texttt{level} parameters to align the input when performing the \texttt{where}.

\begin{verbatim}
In [117]: df2 = df.copy()

In [118]: df2.where(df2>0,df2['A'],axis='index')
Out[118]:
A  B  C  D
2000-01-01 -1.743161 -1.743161 -1.743161 1.314232
2000-01-02  0.690579  0.995761  2.396780  0.014871
2000-01-03  3.357427  3.357427  3.357427  0.896171
2000-01-04 -0.487602 -0.487602 -0.487602  0.380396
2000-01-05  0.084844  0.432390  1.519970  0.084844
2000-01-06  0.600178  0.274230  0.132885  0.600178
2000-01-07  2.410179  1.450520  0.206053  2.410179
2000-01-08 -2.213588  1.063327  1.266143  0.299368

[8 rows x 4 columns]
\end{verbatim}

This is equivalent (but faster than) the following.

\begin{verbatim}
In [119]: df2 = df.copy()

In [120]: df.apply(lambda x, y: x.where(x>0,y), y=df['A'])
Out[120]:
A  B  C  D
2000-01-01 -1.743161 -1.743161 -1.743161 1.314232
2000-01-02  0.690579  0.995761  2.396780  0.014871
2000-01-03  3.357427  3.357427  3.357427  0.896171
2000-01-04 -0.487602 -0.487602 -0.487602  0.380396
2000-01-05  0.084844  0.432390  1.519970  0.084844
2000-01-06  0.600178  0.274230  0.132885  0.600178
2000-01-07  2.410179  1.450520  0.206053  2.410179
2000-01-08 -2.213588  1.063327  1.266143  0.299368
\end{verbatim}
2000-01-06  0.600178  0.274230  0.132885  0.600178
2000-01-07  2.410179  1.450520  0.206053  2.410179
2000-01-08 -2.213588  1.063327  1.266143  0.299368
[8 rows x 4 columns]

mask

mask is the inverse boolean operation of where.

In [121]: s.mask(s >= 0)
Out[121]:
4  NaN
3  NaN
2  NaN
1  NaN
0  NaN
dtype: float64

In [122]: df.mask(df >= 0)
Out[122]:
   A         B         C         D
2000-01-01 -1.743161 -0.826591 -0.345352  NaN
2000-01-02  NaN       NaN       NaN       NaN
2000-01-03  NaN  -0.317441 -1.236269  NaN
2000-01-04  -0.487602 -0.082240 -2.182937  NaN
2000-01-05  NaN       NaN       NaN  -0.493662
2000-01-06  NaN       NaN       NaN  -0.023688
2000-01-07  NaN       NaN       NaN  -0.251905
2000-01-08  -2.213588  NaN       NaN       NaN
[8 rows x 4 columns]

10.12 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:

In [123]: n = 10

In [124]: df = DataFrame(rand(n, 3), columns=list('abc'))

In [125]: df
Out[125]:
   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  0.368824
8  0.933140  0.651378  0.397203
9  0.788730  0.316836  0.568099
# pure python
In [126]: df[(df.a < df.b) & (df.b < df.c)]
Out[126]:
   a    b    c
2 0.276 0.802 0.958

[1 rows x 3 columns]

# query
In [127]: df.query('(a < b) & (b < c)')
Out[127]:
   a    b    c
2 0.276 0.802 0.958

[1 rows x 3 columns]

Do the same thing but fallback on a named index if there is no column with the name a.

In [128]: df = DataFrame(randint(n / 2, size=(n, 2)), columns=list('bc'))

In [129]: df.index.name = 'a'

In [130]: df
Out[130]:
   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

[10 rows x 2 columns]

In [131]: df.query('a < b and b < c')
Out[131]:
   b    c
a
0 2 3

[1 rows x 2 columns]

If instead you don’t want to or cannot name your index, you can use the name index in your query expression:

In [132]: df = DataFrame(randint(n, size=(n, 2)), columns=list('bc'))

In [133]: df
Out[133]:
   b    c
0 3 1
1 2 5
2 2 5
In [134]: df.query('index < b < c')
Out[134]:
    b  c
0  1  2
1  3  5
[2 rows x 2 columns]

10.12.1 MultiIndex query() Syntax

You can also use the levels of a DataFrame with a MultiIndex as if they were columns in the frame:

In [135]: import pandas.util.testing as tm
In [136]: n = 10
In [137]: colors = tm.choice(['red', 'green'], size=n)
In [138]: foods = tm.choice(['eggs', 'ham'], size=n)
In [139]: colors
Out[139]:
array(['green', 'red', 'green', 'red', 'green', 'red', 'green', 'red',
      'green', 'red'],
       dtype='|S5')
In [140]: foods
Out[140]:
array(['ham', 'eggs', 'ham', 'eggs', 'ham', 'eggs', 'ham', 'ham',
      'ham', 'ham'],
       dtype='|S4')
In [141]: index = MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
In [142]: df = DataFrame(randn(n, 2), index=index)
In [143]: df
Out[143]:
    0      1
color food
green ham  0.565738  1.545659
red  eggs -0.974236 -0.070345
green ham  0.307969 -0.208499
red  eggs  1.033801 -2.400454
green ham  2.030604 -1.142631
red  eggs  0.211883  0.704721
green ham -0.785435  0.462060
pandas: powerful Python data analysis toolkit, Release 0.13.1

red  ham  0.704228  0.523508
green ham -0.926254  2.007843
red  ham  0.226963  -1.152659

[10 rows x 2 columns]

**In [144]:** df.query('color == "red"')

```python
Out[144]:
   0   1
0 color  food
   red  eggs -0.974236 -0.070345
eggs  1.033801 -2.400454
eggs  0.211883  0.704721
ham  0.704228  0.523508
ham  0.226963  -1.152659
```

[5 rows x 2 columns]

If the levels of the MultiIndex are unnamed, you can refer to them using special names:

**In [145]:** df.index.names = [None, None]

**In [146]:** df

```python
Out[146]:
   0   1
0 green ham  0.565738  1.545659
red  eggs -0.974236 -0.070345
green ham  0.307969  -0.208499
red  eggs  1.033801  -2.400454
green ham  2.030604 -1.142631
red  eggs  0.211883  0.704721
green ham  0.785435  0.462060
red  ham  0.704228  0.523508
green ham -0.926254  2.007843
red  ham  0.226963  -1.152659
```

[10 rows x 2 columns]

**In [147]:** df.query('ilevel_0 == "red"')

```python
Out[147]:
   0   1
0 red  eggs -0.974236 -0.070345
eggs  1.033801 -2.400454
eggs  0.211883  0.704721
ham  0.704228  0.523508
ham  0.226963  -1.152659
```

[5 rows x 2 columns]

The convention is ilevel_0, which means “index level 0” for the 0th level of the index.

### 10.12.2 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 [148]: df = DataFrame(rand(n, 3), columns=list('abc'))

In [149]: df
Out[149]:
   a  b  c
0  0.528224  0.951429  0.480359
1  0.502560  0.536878  0.819202
2  0.057116  0.669422  0.767117
3  0.708115  0.796867  0.557761
4  0.965837  0.147157  0.029647
5  0.593893  0.114066  0.950810
6  0.325707  0.193619  0.457812
7  0.920403  0.879069  0.252616
8  0.348009  0.182589  0.901796
9  0.706528  0.726658  0.900088
[10 rows x 3 columns]

In [150]: df2 = DataFrame(rand(n + 2, 3), columns=df.columns)

In [151]: df2
Out[151]:
   a  b  c
0  0.779164  0.599155  0.291125
1  0.151395  0.335175  0.657552
2  0.073343  0.055006  0.323195
3  0.590482  0.853899  0.287062
4  0.173067  0.134021  0.994654
5  0.179498  0.317547  0.568291
6  0.009349  0.900649  0.977241
7  0.556895  0.084774  0.333002
8  0.728429  0.142435  0.552469
9  0.273043  0.974495  0.667787
10  0.255653  0.108311  0.776181
11  0.782478  0.761604  0.914403
[12 rows x 3 columns]

In [152]: expr = '0.0 <= a <= c <= 0.5'

In [153]: map(lambda frame: frame.query(expr), [df, df2])
Out[153]:
   a  b  c
0  6  0.325707  0.193619  0.457812
[1 rows x 3 columns],
   a  b  c
2  0.073343  0.055006  0.323195
[1 rows x 3 columns]

10.12.3 query () Python versus pandas Syntax Comparison

Full numpy-like syntax
In [154]: df = DataFrame(randint(n, size=(n, 3)), columns=list('abc'))

In [155]: df
Out[155]:
   a  b  c
0  2  3  1
1  7  1  4
2  7  3  8
3  4  5  3
4  8  8  8
5  1  3  6
6  8  9  1
7  5  8  4
8  1  1  1
9  2  3  4

[10 rows x 3 columns]

In [156]: df.query('(a < b) & (b < c)')
Out[156]:
   a  b  c
5  1  3  6
9  2  3  4

[2 rows x 3 columns]

In [157]: df[(df.a < df.b) & (df.b < df.c)]
Out[157]:
   a  b  c
5  1  3  6
9  2  3  4

[2 rows x 3 columns]

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than &/)

In [158]: df.query('a < b < c')
Out[158]:
   a  b  c
5  1  3  6
9  2  3  4

[2 rows x 3 columns]

Use English instead of symbols

In [159]: df.query('a < b and b < c')
Out[159]:
   a  b  c
5  1  3  6
9  2  3  4

[2 rows x 3 columns]

Pretty close to how you might write it on paper

In [160]: df.query('a < b < c')
Out[160]:
   a  b  c
5  1  3  6
9  2  3  4

10.12. The query() Method (Experimental)
10.12.4 The in and not in operators

`query()` also supports special use of Python’s in and not in comparison operators, providing a succinct syntax for calling the `isin` method of a Series or DataFrame.

```
# get all rows where columns "a" and "b" have overlapping values
In [161]: df = DataFrame({
               'a': list('aabbccddeeff'),
               'b': list('aaaabbbbcccc'),
               'c': randint(5, size=12),
               'd': randint(9, size=12)}
In [162]: df
Out[162]:
     a  b  c  d
 0   a  a  2  2
 1   a  a  3  5
 2   b  a  1  8
 3   b  a  1  8
 4   c  b  4  7
 5   c  b  0  5
 6   d  b  0  7
 7   d  b  2  0
 8   e  c  1  0
 9   e  c  4  6
10  f  c  2  6
11  f  c  3  1
[12 rows x 4 columns]
```

```
In [163]: df.query('a in b')
Out[163]:
     a  b  c  d
 0   a  a  2  2
 1   a  a  3  5
 2   b  a  1  8
 3   b  a  1  8
 4   c  b  4  7
 5   c  b  0  5
[6 rows x 4 columns]
```

```
# How you’d do it in pure Python
In [164]: df[df.a.isin(df.b)]
Out[164]:
     a  b  c  d
 0   a  a  2  2
 1   a  a  3  5
 2   b  a  1  8
 3   b  a  1  8
 4   c  b  4  7
 5   c  b  0  5
[6 rows x 4 columns]
```

```
In [165]: df.query('a not in b')
Out[165]:
```
a b c d
d b 0 7
d b 2 0
e c 1 0
e c 4 6
f c 2 6
f c 3 1

[6 rows x 4 columns]

# pure Python
In [166]: df[~df.a.isin(df.b)]
Out[166]:
   a  b  c  d
  6  d  b 0 7
  7  d  b 2 0
  8  e  c 1 0
  9  e  c 4 6
 10  f  c 2 6
 11  f  c 3 1

[6 rows x 4 columns]

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 [167]: df.query('a in b and c < d')
Out[167]:
   a  b  c  d
  1  a  a 3 5
  2  b  a 1 8
  3  b  a 1 8
  4  c  b 4 7
  5  c  b 0 5

[5 rows x 4 columns]

# pure Python
In [168]: df[df.b.isin(df.a) & (df.c < df.d)]
Out[168]:
   a  b  c  d
  1  a  a 3 5
  2  b  a 1 8
  3  b  a 1 8
  4  c  b 4 7
  5  c  b 0 5
  6  d  b 0 7
  9  e  c 4 6
 10  f  c 2 6

[8 rows x 4 columns]

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
10.12.5 Special use of the == operator with list objects

Comparing a list of values to a column using ==/!= works similarly to in/not in

In [169]: df.query('b == ["a", "b", "c"]')
Out[169]:
<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>a</td>
<td>a</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>b</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>b</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>b</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>c</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>c</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>c</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

[12 rows x 4 columns]

# pure Python

In [170]: df[df.b.isin(['a', 'b', 'c'])]
Out[170]:
<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>a</td>
<td>a</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>b</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>b</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>b</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>b</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>c</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>c</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>c</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

[12 rows x 4 columns]

In [171]: df.query('c == [1, 2]')
Out[171]:
<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>a</td>
<td>a</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>b</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>c</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

[6 rows x 4 columns]

In [172]: df.query('c != [1, 2]')
Out[172]:

Chapter 10. Indexing and Selecting Data
10.12.6 Boolean Operators

You can negate boolean expressions with the word not or the ~ operator.

In [176]: df = DataFrame(rand(n, 3), columns=list('abc'))

In [177]: df['bools'] = rand(len(df)) > 0.5

In [178]: df.query('~bools')
Of course, expressions can be arbitrarily complex too

# short query syntax
In [181]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [182]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]

In [183]: shorter
Out[183]:
   a    b    c  bools
0  5 0.035597 0.171689 0.189045 False
9  9 0.504279 0.746247 0.877177 False
[2 rows x 4 columns]

In [184]: longer
Out[184]:
   a    b    c  bools
5  5 0.035597 0.171689 0.189045 False
9  9 0.504279 0.746247 0.877177 False
[2 rows x 4 columns]
[2 rows x 4 columns]

**In [185]:** shorter == longer

**Out[185]:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>bools</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>9</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

[2 rows x 4 columns]

### 10.12.7 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.
This plot was created using a DataFrame with 3 columns each containing floating point values generated using numpy.random.randn().

### 10.13 Take Methods

Similar to numpy ndarrays, pandas Index, Series, and DataFrame also provides the `take` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [186]: index = Index(randint(0, 1000, 10))

In [187]: index
Out[187]: Int64Index([399, 134, 575, 635, 358, 102, 468, 657, 848, 343], dtype='int64')

In [188]: positions = [0, 9, 3]

In [189]: index[positions]
Out[189]: Int64Index([399, 343, 635], dtype='int64')

In [190]: index.take(positions)
Out[190]: Int64Index([399, 343, 635], dtype='int64')

In [191]: ser = Series(randn(10))

In [192]: ser.ix[positions]
Out[192]:
   0   -0.921988
   9    0.391944
   3   -0.220720
dtype: float64

In [193]: ser.take(positions)
Out[193]:
   0   -0.921988
   9    0.391944
For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```python
In [194]: frm = DataFrame(randn(5, 3))

In [195]: frm.take([1, 4, 3])
Out[195]:
        0      1      2
0  1.031070  1.334887
1 -1.387499 -0.717938 -0.930118
4 -0.710203  1.263598 -2.113153
3 -2.430222  1.583772  2.991093

[3 rows x 3 columns]
```

```python
In [196]: frm.take([0, 2], axis=1)
Out[196]:
        0      2
0  1.031070  1.334887
1 -1.387499 -0.930118
2 -0.752610 -0.212412
3 -2.430222  2.991093
4 -0.710203 -2.113153

[5 rows x 2 columns]
```

It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```python
In [197]: arr = randn(10)

In [198]: arr.take([False, False, True, True])
Out[198]: array([ 0.191 ,  0.191 ,  0.2296,  0.2296])

In [199]: arr[[0, 1]]
Out[199]: array([ 0.191 ,  0.2296])

In [200]: ser = Series(randn(10))

In [201]: ser.take([False, False, True, True])
Out[201]:
        0      1
0  1.557902  1.089202
1  1.089202  1.089202
dtype: float64

In [202]: ser.ix[[0, 1]]
Out[202]:
        0      1
0  1.557902  1.089202
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.
10.14 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.

```
In [203]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                 ......:     'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                 ......:     'c' : np.random.randn(7))

In [204]: df2.duplicated(['a','b'])
Out[204]:
0    False
1    False
2    False
3    False
4     True
5     True
6    False
dtype: bool

In [205]: df2.drop_duplicates(['a','b'])
Out[205]:
   a  b   c
0  one  x -1.363210
1  one  y  0.623587
2  two  y -1.808744
3 three  x -0.367734
6  six  x -0.554902
[5 rows x 3 columns]

In [206]: df2.drop_duplicates(['a','b'], take_last=True)
Out[206]:
   a  b   c
1  one  y  0.623587
3 three  x -0.367734
4  two  y  1.787442
5  one  x -1.420214
6  six  x -0.554902
[5 rows x 3 columns]
```

10.15 Dictionary-like `get()` method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.
In [207]: s = Series([1,2,3], index=['a','b','c'])

In [208]: s.get('a')  # equivalent to s['a']
Out[208]: 1

In [209]: s.get('x', default=-1)
Out[209]: -1

10.16 Advanced Indexing with .ix

Note: The recent addition of .loc and .iloc have enabled users to be quite explicit about indexing choices. .ix allows a great flexibility to specify indexing locations by label and/or integer position. Pandas will attempt to use any passed integer as label locations first (like what .loc would do, then to fall back on positional indexing, like what .iloc would do). See Fallback Indexing for an example.

The syntax of using .ix is identical to .loc in Selection by Label, and .iloc in Selection by Position.

The .ix attribute takes the following inputs:

- An integer or single label, e.g. 5 or 'a'
- A list or array of labels ['a', 'b', 'c'] or integers [4, 3, 0]
- A slice object with ints 1:7 or labels 'a':'f'
- A boolean array

We’ll illustrate all of these methods. First, note that this provides a concise way of reindexing on multiple axes at once:

In [210]: subindex = dates[[3,4,5]]

In [211]: df.reindex(index=subindex, columns=['C', 'B'])
Out[211]:
   C   B
2000-01-04  0.036249  0.484166
2000-01-05  0.378125 -1.180301
2000-01-06  0.075871  0.441177
[3 rows x 2 columns]

In [212]: df.ix[subindex, ['C', 'B']]
Out[212]:
   C   B
2000-01-04  0.036249  0.484166
2000-01-05  0.378125 -1.180301
2000-01-06  0.075871  0.441177
[3 rows x 2 columns]

Assignment / setting values is possible when using ix:

In [213]: df2 = df.copy()

In [214]: df2.ix[subindex, ['C', 'B']] = 0

In [215]: df2
Out[215]:

10.16. Advanced Indexing with .ix 257
Indexing with an array of integers can also be done:

```
In [216]: df.ix[[4,3,1]]
Out[216]:
    A         B         C         D
2000-01-05 1.459886 -1.180301  0.378125 -0.038520
2000-01-04 0.245116  0.484166  0.036249 -0.546831
2000-01-02 0.240054 -0.057057 -0.173676 -0.119693
```

[3 rows x 4 columns]

Slicing has standard Python semantics for integer slices:

```
In [218]: df.ix[1:7, :2]
Out[218]:
    A         B
2000-01-02 0.240054 -0.057057
2000-01-03 1.315562  0.089291
2000-01-04 0.245116  0.484166
2000-01-05 1.459886 -1.180301
2000-01-06 1.926220  0.441177
2000-01-07 -0.042475 -1.265025
```

[6 rows x 2 columns]

Slicing with labels is semantically slightly different because the slice start and stop are **inclusive** in the label-based case:

```
In [219]: date1, date2 = dates[[2, 4]]
In [220]: print(date1, date2)
(Timestamp('2000-01-03 00:00:00', tz=None), Timestamp('2000-01-05 00:00:00', tz=None))
In [221]: df.ix[date1:date2]
Out[221]:
    A     B     C     D
2000-01-03 1.315562 0.089291 0.454389 0.854294
2000-01-04 0.245116 0.484166 0.036249 -0.546831
```

[8 rows x 4 columns]
Getting and setting rows in a DataFrame, especially by their location, is much easier:

In [223]: df2 = df[:5].copy()

In [224]: df2.ix[3]
Out[224]:
A 0.245116
B 0.484166
C 0.036249
D -0.546831
Name: 2000-01-04 00:00:00, dtype: float64

In [225]: df2.ix[3] = np.arange(len(df2.columns))

In [226]: df2
Out[226]:
     A     B     C     D
2000-01-01 1.438115 -0.355420 1.391176 -0.349452
2000-01-02 0.240054 -0.057057 -0.173676 -0.119693
2000-01-03 1.315562 0.089291 0.454389 0.854294
2000-01-04 0.000000 1.000000 2.000000 3.000000
2000-01-05 1.459886 -1.180301 0.378125 -0.038520

Column or row selection can be combined as you would expect with arrays of labels or even boolean vectors:

In [227]: df.ix[df['A'] > 0, 'B']
Out[227]:
2000-01-01 -0.355420
2000-01-02 -0.057057
2000-01-03 0.089291
2000-01-04 0.484166
2000-01-05 -1.180301
2000-01-06 0.441177
2000-01-08 -0.592656
Name: B, dtype: float64

In [228]: df.ix[date1:date2, 'B']
Out[228]:
2000-01-03 0.089291
2000-01-04 0.484166
2000-01-05 -1.180301
Freq: D, Name: B, dtype: float64

In [229]: df.ix[date1, 'B']
Out[229]: 0.089290642365767614
Slicing with labels is closely related to the `truncate` method which does precisely `.ix[start:stop]` but returns a copy (for legacy reasons).

### 10.17 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 [230]: df.select(lambda x: x == 'A', axis=1)
Out[230]:
       A
2000-01-01  1.438115
2000-01-02  0.240054
2000-01-03  1.315562
2000-01-04  0.245116
2000-01-05  1.459886
2000-01-06  1.926220
2000-01-07  0.042475
2000-01-08  0.518029
[8 rows x 1 columns]
```

### 10.18 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 [231]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])
In [232]: dflookup.lookup(list(range(0,10,2)), ['B','C','A','B','D'])
Out[232]: array([ 0.4254, 0.4579, 0.0333, 0.3469, 0.3085])
```

### 10.19 Float64Index

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.

```
In [233]: indexf = Index([1.5, 2, 3, 4.5, 5])
In [234]: indexf
Out[234]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='object')
In [235]: sf = Series(range(5),index=indexf)
In [236]: sf
Out[236]:
1.5  0
2.0  1
3.0  2
4.5  3
```
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 [237]: sf[3]
Out[237]: 2

In [238]: sf[3.0]
Out[238]: 2

In [239]: sf.ix[3]
Out[239]: 2

In [240]: sf.ix[3.0]
Out[240]: 2

In [241]: sf.loc[3]
Out[241]: 2

In [242]: sf.loc[3.0]
Out[242]: 2
```

The only positional indexing is via iloc

```
In [243]: sf.iloc[3]
Out[243]: 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 [244]: sf[2:4]
Out[244]:
   2   1
   3   2
dtype: int64

In [245]: sf.ix[2:4]
Out[245]:
   2   1
   3   2
dtype: int64

In [246]: sf.loc[2:4]
Out[246]:
   2   1
   3   2
dtype: int64

In [247]: sf.iloc[2:4]
Out[247]:
   3.0  2
   4.5  3
dtype: int64
```

In float indexes, slicing using floats is allowed
In [248]: sf[2.1:4.6]
Out[248]:
3.0  2
4.5  3
dtype: int64

In [249]: sf.loc[2.1:4.6]
Out[249]:
3.0  2
4.5  3
dtype: int64

In non-float indexes, slicing using floats will 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

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 [250]: dfir = concat([DataFrame(randn(5,2),
.....:   index=np.arange(5) * 250.0,
.....:   columns=list('AB')),
.....:   DataFrame(randn(6,2),
.....:   index=np.arange(4,10) * 250.1,
.....:   columns=list('AB'))])

In [251]: dfir
Out[251]:
   A    B
0  0.0 -0.384281  1.296627
250.0 -1.804211  0.560015
500.0  0.640348 -0.227414
750.0 -0.582619 -0.902874
1000.0  0.039911 -0.270138
1000.4  1.115044  0.969404
1250.5 -0.781151 -2.784845
1500.6 -1.201786 -0.231876
1750.7  0.142467  0.060178
2000.8 -0.822858  1.876000
2250.9 -0.932658 -0.635533

[11 rows x 2 columns]

Selection operations then will always work on a value basis, for all selection operators.

In [252]: dfir[0:1000.4]
Out[252]:
   A    B
0.0 -0.384281  1.296627
250.0  -1.804211  0.560015
500.0   0.640348  -0.227414
750.0  -0.582619  -0.902874
1000.0  0.039911  -0.270138
1000.4  1.115044  0.969404

[6 rows x 2 columns]

In [253]: dfir.loc[0:1001,'A']
Out[253]:
     A
0  -0.384281
250  -1.804211
500   0.640348
750  -0.582619
1000  0.039911
1000.4  1.115044
Name: A, dtype: float64

In [254]: dfir.loc[1000.4]
Out[254]:
   A   B
1000.4 1.115044 0.969404
Name: 1000.4, dtype: float64

You could then easily pick out the first 1 second (1000 ms) of data then.

In [255]: dfir[0:1000]
Out[255]:
     A   B
0  -0.384281 1.296627
250  -1.804211 0.560015
500   0.640348 -0.227414
750  -0.582619 -0.902874
1000  0.039911 -0.270138

[5 rows x 2 columns]

Of course if you need integer based selection, then use iloc

In [256]: dfir.iloc[0:5]
Out[256]:
     A   B
0  -0.384281 1.296627
250  -1.804211 0.560015
500   0.640348 -0.227414
750  -0.582619 -0.902874
1000  0.039911 -0.270138

[5 rows x 2 columns]

10.20 Returning a view versus a copy

The rules about when a view on the data is returned are entirely dependent on NumPy. Whenever an array of labels or a boolean vector are involved in the indexing operation, the result will be a copy. With single label / scalar indexing and slicing, e.g. df.ix[3:6] or df.ix[:, 'A'], a view will be returned.

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 [257]: dfb = DataFrame({'a' : ['one', 'one', 'two',
       ....:     'three', 'two', 'one', 'six'],
       ....:     'c' : np.arange(7))

# passed via reference (will stay)
In [258]: dfb['c'][dfb.a.str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```
>>> pd.set_option('mode.chained_assignment', 'warn')
>>> dfb[dfb.a.str.startswith('o')]['c'] = 42
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
```

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 [259]: dfc = DataFrame({'A': ['aaa','bbb','ccc']}, 'B':[1,2,3])

In [260]: dfc.loc[0,'A'] = 11
```

```
In [261]: dfc
Out[261]:
    A  B
0  11  1
1  bbb 2
2  ccc 3
[3 rows x 2 columns]
```

This can work at times, but is not guaranteed, and so should be avoided

```
In [262]: dfc = dfc.copy()

In [263]: dfc['A'][0] = 111
```

```
In [264]: dfc
Out[264]:
    A  B
0  111 1
1  bbb 2
2  ccc 3
[3 rows x 2 columns]
```

This will not work at all, and so should be avoided
```python
>>> pd.set_option('mode.chained_assignment','raise')
>>> dfc.loc[0]['A'] = 1111
Traceback (most recent call last)
...
SettingWithCopyException:
  A value is trying to be set on a copy of a slice from a DataFrame.
  Try using .loc[row_index,col_indexer] = value instead

Warning:  The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.

10.21 Fallback indexing

Float indexes should be used only with caution. If you have a float indexed DataFrame and try to select using an integer, the row that Pandas returns might not be what you expect. Pandas first attempts to use the integer as a label location, but fails to find a match (because the types are not equal). Pandas then falls back to back to positional indexing.

In [265]: df = pd.DataFrame(np.random.randn(4,4),
                   columns=list('ABCD'), index=[1.0, 2.0, 3.0, 4.0])

In [266]: df
Out[266]:
    A     B     C     D
  1  0.379122 -1.909492 -1.431211 1.329653
  2 -0.562165  0.585729 -0.544104 0.825851
  3 -0.062472  2.032089  0.639479 -1.550712
  4  0.903495  0.476501 -0.800435 -1.596836

[4 rows x 4 columns]

In [267]: df.ix[1]
Out[267]:
    A     B     C     D
  Name: 1.0, dtype: float64

In [268]: df.iloc[0]
Out[268]:
    A     B     C     D
  Name: 0, dtype: float64

To select the row you do expect, instead use a float label or use iloc.

In [269]: df.iloc[1.0]
Out[269]:
    A     B     C     D
  Name: 1.0, dtype: float64
Instead of using a float index, it is often better to convert to an integer index:

```python
In [270]: df_new = df.reset_index()
```

```python
In [271]: df_new[df_new['index'] == 1.0]
```

```
    index  A       B     C    D
   0 1.0  0.379122 -1.909492 -1.431211  1.329653
```

[1 rows x 5 columns]

# now you can also do "float selection"

```python
In [272]: df_new[(df_new['index'] >= 1.0) & (df_new['index'] < 2)]
```

```
    index  A       B     C    D
   0 1.0  0.379122 -1.909492 -1.431211  1.329653
```

[1 rows x 5 columns]

## 10.22 Index objects

The pandas `Index` class and its subclasses can be viewed as implementing an *ordered multiset*. Duplicates are allowed. However, if you try to convert an `Index` object with duplicate entries into a `set`, an exception will be raised.

`Index` also provides the infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create an `Index` directly is to pass a list or other sequence to `Index`:

```python
In [273]: index = Index(['e', 'd', 'a', 'b'])
```

```python
In [274]: index
```

```
Out[274]: Index(['e', 'd', 'a', 'b'], dtype='object')
```

```python
In [275]: 'd' in index
```

```
Out[275]: True
```

You can also pass a name to be stored in the index:

```python
In [276]: index = Index(['e', 'd', 'a', 'b'], name='something')
```

```python
In [277]: index.name
```

```
Out[277]: 'something'
```

Starting with pandas 0.5, the name, if set, will be shown in the console display:

```python
In [278]: index = Index(list(range(5)), name='rows')
```

```python
In [279]: columns = Index(['A', 'B', 'C'], name='cols')
```

```python
In [280]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)
```

```python
In [281]: df
```

```
cols     A       B     C
   0  0.156553 -0.074507 -0.527524
   1  0.846720 -0.433444  0.322084
   2 -0.184410 -0.724147 -0.463663
   3  1.160397 -0.487131  0.213254
   4 -0.470376  0.224740 -0.252763
```

[5 rows x 3 columns]
rows
0  0.242701  0.302298  1.249715
1  -1.524904 -0.726778  0.279579
2   1.059562 -1.783941 -1.377069
3   0.150077 -1.300946 -0.342584
4  -1.972104  0.961460  1.222320
[5 rows x 3 columns]

In [282]: df['A']
Out[282]:
rows
0  0.242701
1 -1.524904
2  1.059562
3  0.150077
4 -1.972104
Name: A, dtype: float64

10.22.1 Set operations on Index objects

The three main operations are union (|), intersection (&), and diff (-). These can be directly called as instance methods or used via overloaded operators:

In [283]: a = Index(['c', 'b', 'a'])
In [284]: b = Index(['c', 'e', 'd'])
In [285]: a.union(b)
Out[285]: Index([u'c', u'b', u'c', u'd', u'e'], dtype='object')
In [286]: a | b
Out[286]: Index([u'c', u'b', u'c', u'd', u'e'], dtype='object')
In [287]: a & b
Out[287]: Index([u'c'], dtype='object')
In [288]: a - b
Out[288]: Index([u'a', u'b'], dtype='object')

10.22.2 The isin method of Index objects

One additional operation is the isin method that works analogously to the Series.isin method found here.

10.23 Hierarchical indexing (MultiIndex)

Hierarchical indexing (also referred to as “multi-level” indexing) is brand new in the pandas 0.4 release. It 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

Note: Given that hierarchical indexing is so new to the library, it is definitely “bleeding-edge” functionality but is certainly suitable for production. But, there may inevitably be some minor API changes as more use cases are explored and any weaknesses in the design / implementation are identified. pandas aims to be “eminently usable” so any feedback about new functionality like this is extremely helpful.

### 10.23.1 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).

```
In [289]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
               ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]

In [290]: tuples = list(zip(*arrays))

In [291]: tuples
Out[291]: [('bar', 'one'),
          ('bar', 'two'),
          ('baz', 'one'),
          ('baz', 'two'),
          ('foo', 'one'),
          ('foo', 'two'),
          ('qux', 'one'),
          ('qux', 'two')]

In [292]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [293]: s = Series(randn(8), index=index)

In [294]: s
Out[294]:
first second
bar  one  0.420597
     two -0.631851
baz  one -1.054843
     two  0.588134
foo  one  1.453543
     two  0.668992
qux  one -0.024028
     two  1.269473
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 [295]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [296]: MultiIndex.from_product(iterables, names=['first', 'second'])
```
As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

```
In [297]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
               np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])
               ...
               ]

In [298]: s = Series(randn(8), index=arrays)

In [299]: s
```

```
bar one 1.039182
two 0.956255
baz one 1.448918
two 0.238470
foo one 0.174031
two -0.793292
qux one 0.051545
two 1.452842
dtype: float64
```

```
In [300]: df = DataFrame(randn(8, 4), index=arrays)

In [301]: df
```

```
      0     1     2     3
bar one 0.115255 -0.442066 -0.586551 -0.950131
two 0.890610 -0.170954 0.355509 -0.284458
baz one 1.094382 0.054720 0.030047 1.978266
two -0.428214 -0.116571 0.013297 -0.632840
foo one -0.906030 0.064289 1.046974 -0.720532
two 1.100970 0.417609 0.986436 -1.277886
qux one 1.534011 0.895957 1.944202 -0.547004
two -0.463114 -1.232976 0.881544 -1.802477
```

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 [302]: df.index.names
Out[302]: 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 [303]: df = DataFrame(randn(3, 8), index=['A', 'B', 'C'], columns=index)

In [304]: df
```

```
     first     bar     baz     foo     qux
second one    two    one    two    one
A -0.007381 -1.219794 0.145578 -0.249321 -1.046479 1.314373 0.716789
```
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:

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:
first   qux  
second  two  
A   0.385795  
B   -0.915417  
C   -1.367644  

[3 rows x 8 columns]

In [309]: pd.set_option('display.multi_sparse', True)

10.23.2 Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

In [310]: index.get_level_values(0)  
Out[310]: Index([u'bar', u'bar', u'baz', u'baz', u'foo', u'foo', u'qux', u'qux'], dtype='object')

In [311]: index.get_level_values('second')  
Out[311]: Index([u'one', u'two', u'one', u'two', u'one', u'two', u'one', u'two'], dtype='object')

10.23.3 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 [312]: df['bar']  
Out[312]:
second  one   two  
A      -0.007381   -1.219794  
B      -0.365315    0.370955  
C      -0.024817   -0.795125  

[3 rows x 2 columns]

In [313]: df['bar', 'one']  
Out[313]:
  A      -0.007381  
  B      -0.365315  
  C      -0.024817  
Name: (bar, one), dtype: float64

In [314]: df['bar']['one']  
Out[314]:
  A      -0.007381  
  B      -0.365315  
  C      -0.024817  
Name: one, dtype: float64

In [315]: s['qux']  
Out[315]:
one    0.051545  
two    1.452842  
dtype: float64

See **Cross-section with hierarchical index** for how to select on a deeper level.
10.23.4 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 [316]: s + s[:-2]
Out[316]:
bar   one  2.078365
       two  1.912509
baz   one  2.897837
       two  0.476941
foo   one  0.348063
       two -1.586583
qux   one  NaN
       two  NaN
dtype: float64
```

```
In [317]: s + s[::2]
Out[317]:
bar   one  2.078365
       NaN
baz   one  2.897837
       NaN
foo   one  0.348063
       NaN
qux   one  0.103090
       NaN
dtype: float64
```

reindex can be called with another MultiIndex or even a list or array of tuples:

```
In [318]: s.reindex(index[:3])
Out[318]:
first  second
bar   one   1.039182
       two   0.956255
baz   one   1.448918
dtype: float64
```

```
In [319]: s.reindex([('foo', 'two')], ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
Out[319]:
foo  two  -0.793292
bar  one   1.039182
qux  one   0.051545
baz  one   1.448918
dtype: float64
```

10.23.5 Advanced indexing with hierarchical index

Syntactically integrating MultiIndex in advanced indexing with .ix is a bit challenging, but we’ve made every effort to do so. For example the following works as you would expect:

```
In [320]: df = df.T
```

```
In [321]: df
Out[321]:
     A     B     C
```

```
first second
bar one -0.007381 -0.365315 -0.024817
two -1.219794 0.370955 -0.795125
baz one 0.145578 1.428502 -0.408384
two -0.249321 -0.292967 -1.849202
foo one -1.046479 -1.250595 0.781722
two 1.314373 0.333150 0.133331
qux one 0.716789 0.616471 -0.298493
two 0.385795 -0.915417 -1.367644
[8 rows x 3 columns]

In [322]: df.ix['bar']
Out[322]:
     A         B         C
second
one -0.007381 -0.365315 -0.024817
two -1.219794  0.370955 -0.795125
[2 rows x 3 columns]

In [323]: df.ix['bar', 'two']
Out[323]:
     A      B      C
Name: (bar, two), dtype: float64

Partial" slicing also works quite nicely for the topmost level:

In [324]: df.ix['baz':'foo']
Out[324]:
     A       B       C
first second
baz one 0.145578 1.428502 -0.408384
two -0.249321 -0.292967 -1.849202
foo one -1.046479 -1.250595  0.781722
two 1.314373  0.333150  0.133331
[4 rows x 3 columns]

But lower levels cannot be sliced in this way, because the MultiIndex uses its multiple index dimensions to slice along
one dimension of your object:

In [325]: df.ix[('baz', 'two'):('qux', 'one')]
Out[325]:
     A       B       C
first second
baz two -0.249321 -0.292967 -1.849202
foo one -1.046479 -1.250595  0.781722
two 1.314373  0.333150  0.133331
qux one 0.716789  0.616471 -0.298493
[4 rows x 3 columns]

In [326]: df.ix[('baz', 'two'):'foo']
Out[326]:
     A       B       C
first second
báz   two  -0.249321 -0.292967 -1.849202
foo   one   -1.046479 -1.250595  0.781722
two   one   1.314373  0.333150  0.133331

[3 rows x 3 columns]

Passing a list of labels or tuples works similar to reindexing:

```
In [327]: df.ix[['bar', 'two'], ('qux', 'one')]
```

```
Out[327]:
   A    B    C
first second
bar two -1.219794 0.370955 -0.795125
qux one  0.716789 0.616471 -0.298493
```

[2 rows x 3 columns]

The following does not work, and it's not clear if it should or not:

```
>>> df.ix[['bar', 'qux']]
```

The code for implementing `.ix` makes every attempt to “do the right thing” but as you use it you may uncover corner cases or unintuitive behavior. If you do find something like this, do not hesitate to report the issue or ask on the mailing list.

### 10.23.6 Cross-section with hierarchical index

The `xs` method of `DataFrame` additionally takes a level argument to make selecting data at a particular level of a `MultiIndex` easier.

```
In [328]: df.xs('one', level='second')
```

```
Out[328]:
   A    B    C
first
bar two -0.007381 0.145578 -1.046479
baz     0.145578 1.428502 -0.408384
foo    -1.046479 -1.250595  0.781722
qux    0.716789  0.616471 -0.298493
```

[4 rows x 3 columns]

You can also select on the columns with `xs()`, by providing the axis argument

```
In [329]: df = df.T
```

```
In [330]: df.xs('one', level='second', axis=1)
```

```
Out[330]:
   bar   baz   foo   qux
A   -0.007381 0.145578 -1.046479 0.716789
B  -0.365315 1.428502 -1.250595 0.616471
C  -0.024817 -0.408384  0.781722 -0.298493
```

[3 rows x 4 columns]

`xs()` also allows selection with multiple keys

```
In [331]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
```

```
Out[331]:
```

274 Chapter 10. Indexing and Selecting Data
first bar
second one
A -0.007381
B -0.365315
C -0.024817

[3 rows x 1 columns]

New in version 0.13.0. You can pass `drop_level=False` to `xs()` to retain the level that was selected versus the result with `drop_level=True` (the default value).

### 10.23.7 Advanced reindexing and alignment with hierarchical index

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 [332]: midx = MultiIndex(levels=[['zero', 'one'], ['x','y']],
                           labels=[[1,1,0,0],[1,0,1,0]])

In [333]: df = DataFrame(randn(4,2), index=midx)
```

```
0 1
one y 0.313092 -0.588491
x 0.203166 1.632996
zero y -0.557549 0.126204
x 1.643615 -0.067716
[4 rows x 2 columns]
```

```
In [335]: df2 = df.mean(level=0)

In [336]: print(df2)
```

```
0 1
zero 0.543033 0.029244
one 0.258129 0.522253
[2 rows x 2 columns]
```

```
In [337]: print(df2.reindex(df.index, level=0))
```

```
0 1
one y 0.258129 0.522253
x 0.258129 0.522253
zero y 0.543033 0.029244
x 0.543033 0.029244
[4 rows x 2 columns]
```

```
In [338]: df_aligned, df2_aligned = df.align(df2, level=0)

In [339]: print(df_aligned)
```

```
0 1
one y 0.313092 -0.588491
x 0.203166 1.632996
zero y -0.557549 0.126204
```
In [340]: print(df2_aligned)
0 1
one y 0.258129 0.522253
x 0.258129 0.522253
zero y 0.543033 0.029244
x 0.543033 0.029244

10.23.8 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!

In [341]: import random; random.shuffle(tuples)

In [342]: s = Series(randn(8), index=MultiIndex.from_tuples(tuples))

In [343]: s
Out[343]:
qux two 0.127064
baz one 0.396144
two 1.043289
qux one -0.229627
bar one 0.158186
two -0.281965
foo two 1.255148
bar two 3.063464
dtype: float64

In [344]: s.sortlevel(0)
Out[344]:
bar one 0.158186
two 1.255148
baz one 0.396144
two 1.043289
foo one 3.063464
two -0.281965
qux one -0.229627
two 0.127064
dtype: float64

In [345]: s.sortlevel(1)
Out[345]:
bar one 0.158186
baz one 0.396144
foo one 3.063464

276 Chapter 10. Indexing and Selecting Data
qux  one  -0.229627
bar  two  1.255148
baz  two  1.043289
foo  two  -0.281965
qux  two  0.127064
dtype: float64

Note, you may also pass a level name to `sortlevel` if the MultiIndex levels are named.

In [346]: s.index.set_names(['L1', 'L2'], inplace=True)
In [347]: s.sortlevel(level='L1')
Out[347]:
L1 L2
bar one 0.158186
two 1.255148
baz one 0.396144
two 1.043289
foo one 3.063464
two -0.281965
qux one -0.229627
two 0.127064
dtype: float64
In [348]: s.sortlevel(level='L2')
Out[348]:
L1 L2
bar one 0.158186
baz one 0.396144
foo one 3.063464
qux one -0.229627
bar two 1.255148
baz two 1.043289
foo two -0.281965
qux two 0.127064
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 [349]: s['qux']
Out[349]:
L2
one  -0.229627
two  0.127064
dtype: float64
In [350]: s.sortlevel(1)['qux']
Out[350]:
L2
one  -0.229627
two  0.127064
dtype: float64

On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

In [351]: df.T.sortlevel(1, axis=1)
Out[351]:
zero   one   zero   one
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:

```
In [352]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]
In [353]: idx = MultiIndex.from_tuples(tuples)
In [354]: idx.lexsort_depth
Out[354]: 2
In [355]: reordered = idx[[1, 0, 3, 2]]
In [356]: reordered.lexsort_depth
Out[356]: 1
In [357]: s = Series(randn(4), index=reordered)
In [358]: s.ix['a':'a']
Out[358]:
   a  b
0  0.304771
1 -0.766820
```

However:

```
>>> s.ix[('a', 'b'):('b', 'a')]
KeyError: Key length (3) was greater than MultiIndex lexsort depth (2)
```

### 10.23.9 Swapping levels with swaplevel()

The swaplevel function can switch the order of two levels:

```
In [359]: df[:5]
Out[359]:
     0     1
   one  y  0.313092 -0.588491
   x  0.203166  1.632996
   zero  y -0.557549  0.126204
    x  1.643615 -0.067716
```

```
In [360]: df[:5].swaplevel(0, 1, axis=0)
Out[360]:
     0     1
   y one  0.313092 -0.588491
   x one  0.203166  1.632996
   y zero -0.557549  0.126204
   x zero  1.643615 -0.067716
```
10.23.10 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 [361]: df[:5].reorder_levels([1,0], axis=0)
Out [361]:
0 1
y one 0.313092 -0.588491
x one 0.203166 1.632996
y zero -0.557549 0.126204
x zero 1.643615 -0.067716
```

[4 rows x 2 columns]

10.23.11 Some gory internal details

Internally, the `MultiIndex` consists of a few things: the `levels`, the integer `labels`, and the level `names`:

```python
In [362]: index
Out [362]:
MultiIndex(levels=[[u'bar', u'baz', u/foo', u/qux']],
 labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
 names=[u'first', u'second'])
```

```python
In [363]: index.levels
Out [363]: FrozenList([[u'bar', u'baz', u/foo', u/qux']], [u'one', u'two'])
```

```python
In [364]: index.labels
Out [364]: FrozenList([[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]])
```

```python
In [365]: index.names
Out [365]: FrozenList([u'first', u'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.

10.24 Setting index metadata (name(s), levels, labels)

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 inplace.

```python
In [366]: ind = Index([1, 2, 3])
```

```python
In [367]: ind.rename("apple")
Out [367]: Int64Index([1, 2, 3], dtype='int64')
```
In [368]: ind
Out[368]: Int64Index([1, 2, 3], dtype='int64')

In [369]: ind.set_names(["apple"], inplace=True)

In [370]: ind.name = "bob"

In [371]: ind
Out[371]: Int64Index([1, 2, 3], dtype='int64')

10.25 Adding an index to an existing DataFrame

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.

10.26 Add an index using DataFrame columns

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 [372]: data
Out[372]:
   a   b   c   d
  0  bar  one  z  1
  1  bar  two  y  2
  2  foo  one  x  3
  3  foo  two  w  4
[4 rows x 4 columns]

In [373]: indexed1 = data.set_index('c')

In [374]: indexed1
Out[374]:
   a   b   d
c
  z  bar  one  1
  y  bar  two  2
  x  foo  one  3
  w  foo  two  4
[4 rows x 3 columns]

In [375]: indexed2 = data.set_index(['a', 'b'])

In [376]: indexed2
Out[376]:
   c   d
  a   b
  bar  one  z  1
    two  y  2
  foo  one  x  3
    two  w  4
The `append` keyword option allows you to keep the existing index and append the given columns to a MultiIndex:

```
In [377]: frame = data.set_index('c', drop=False)

In [378]: frame = frame.set_index(['a', 'b'], append=True)

In [379]: frame
Out[379]:
    c   d
   a b
   z bar one z 1
   y bar two y 2
   x foo one x 3
   w foo two w 4

[4 rows x 2 columns]
```

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 [380]: data.set_index('c', drop=False)
Out[380]:
    a   b   c   d
   c
   z bar one z 1
   y bar two y 2
   x foo one x 3
   w foo two w 4

[4 rows x 4 columns]

In [381]: data.set_index(['a', 'b'], inplace=True)

In [382]: data
Out[382]:
    c   d
   a b
   bar one z 1
      two y 2
   foo one x 3
      two w 4

[4 rows x 2 columns]
```

10.27 Remove / reset the index, `reset_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 [383]: data
Out[383]:
    c   d
   a b
   bar one z 1
```

10.27. Remove / reset the index, `reset_index`
```
two y 2
go one x 3
two w 4
[4 rows x 2 columns]
```

In [384]: data.reset_index()
Out[384]:
```
a  b  c  d
0 bar one z 1
1 bar two y 2
2 foo one x 3
3 foo two w 4
[4 rows x 4 columns]
```

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 [385]: frame
Out[385]:
```
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
[4 rows x 2 columns]
```

In [386]: frame.reset_index(level=1)
Out[386]:
```
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
[4 rows x 3 columns]
```

`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.

### 10.28 Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
data.index = index
```
10.29 Indexing internal details

Note: The following is largely relevant for those actually working on the pandas codebase. The source code is still the best place to look at the specifics of how things are implemented.

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
- **MultiIndex**: the standard hierarchical index object
- **PeriodIndex**: An Index object with Period elements
- **DatetimeIndex**: An Index object with Timestamp elements
- **date_range**: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Python datetime objects

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
11.1 Statistical functions

11.1.1 Percent Change

Both Series and DataFrame have a method `pct_change` to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values).

```
In [1]: ser = Series(randn(8))
In [2]: ser.pct_change()
Out[2]:
   0    NaN
   1  -1.602976
   2   4.334938
   3  -0.247456
   4  -2.067345
   5  -1.142903
   6  -1.688214
   7  -9.759729
dtype: float64
```

```
In [3]: df = DataFrame(randn(10, 4))
In [4]: df.pct_change(periods=3)
Out[4]:
     0     1     2     3
0   NaN   NaN   NaN   NaN
1  NaN   NaN   NaN   NaN
2  NaN   NaN   NaN   NaN
3 -0.218320 -1.054001 1.987147 -0.510183
4 -0.439121 -1.816454 0.649715 -4.822809
5 -0.127833 -3.042065 -5.866604 -1.776977
6 -2.596833 -1.959538 -2.111697 -3.798900
7 -0.117826 -2.169058 0.036094 -0.067696
8  2.492606 -1.357320 -1.205802 -1.558697
9 -1.012977  2.324558 -1.003744 -0.371806
```

[10 rows x 4 columns]
11.1.2 Covariance

The Series object has a method `cov` to compute covariance between series (excluding NA/null values).

```python
In [5]: s1 = Series(randn(1000))
In [6]: s2 = Series(randn(1000))
In [7]: s1.cov(s2)
Out[7]: 0.0006801088174310957
```

Analogously, DataFrame has a method `cov` to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

```python
In [8]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [9]: frame.cov()
```

```
Out: [5 rows x 5 columns]
```

DataFrame.cov also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```python
In [10]: frame = DataFrame(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  1.210090 -0.430629  0.018002
b -0.430629  1.240960  0.347188
c  0.018002  0.347188  1.301149
[3 rows x 3 columns]
```

In [14]: frame.cov(min_periods=12)
Out[14]:
```
a  1.210090  NaN  0.018002
b  NaN    1.240960  0.347188
c  0.018002  0.347188  1.301149
[3 rows x 3 columns]
```

11.1.3 Correlation

Several methods for computing correlations are provided. Several kinds of correlation methods 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.

```python
In [15]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [16]: frame.ix[::2] = np.nan

In [17]: frame['a'].corr(frame['b'])
Out[17]: 0.013479040400098763
In [18]: frame['a'].corr(frame['b'], method='spearman')
Out[18]: -0.0072898851595406388

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

[5 rows x 5 columns]
```

Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```python
In [20]: frame = DataFrame(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 NaN  0.160092
   b -0.765200  1.000000  0.135967
   c  0.160092  0.135967  1.000000

[3 rows x 3 columns]
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

[3 rows x 3 columns]
```

A related method `corrwith` is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

11.1. Statistical functions  287
11.1.4 Data ranking

The `rank` method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```python
In [31]: s = 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    b    c    d     e
a  5.0  2.5  1.0  2.5  4.00
```

`rank` is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). NaN values are excluded from the ranking.

```python
In [34]: df = DataFrame(np.random.randn(10, 6))
In [36]: df
```

```
Out[36]:
     0    1    2    3    4    5
0 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
```
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

### 11.2 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_corr_pairwise</td>
<td>Pairwise correlation of DataFrame columns</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

These functions can be applied to ndarrays or Series objects:

```python
In [38]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))
In [39]: ts = ts.cumsum()
In [40]: ts.plot(style='k--')
Out[40]: <matplotlib.axes.AxesSubplot at 0x7abc7d0>
In [41]: rolling_mean(ts, 60).plot(style='k')
Out[41]: <matplotlib.axes.AxesSubplot at 0x7abc7d0>
```
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 = DataFrame(randn(1000, 4), index=ts.index,
.....:       columns=['A', 'B', 'C', 'D'])
.....:

In [43]: df = df.cumsum()

In [44]: rolling_sum(df, 60).plot(subplots=True)
Out[44]:
array([<matplotlib.axes.AxesSubplot object at 0x7c24dd0>,
       <matplotlib.axes.AxesSubplot object at 0x5c4fbd0>,
       <matplotlib.axes.AxesSubplot object at 0x699b750>,
       <matplotlib.axes.AxesSubplot object at 0x5dfb3d0>], dtype=object)
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]: rolling_apply(ts, 60, mad).plot(style='k')
Out[46]: <matplotlib.axes.AxesSubplot at 0x60054d0>
```

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 = Series(randn(10), index=date_range('1/1/2000', periods=10))

In [48]: 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  -0.622722
2000-01-06  -0.460623
2000-01-07  -0.229918
2000-01-08  -0.237308
2000-01-09  -0.335064
2000-01-10  -0.403449
Freq: D, dtype: float64

Note that the boxcar window is equivalent to rolling_mean:

In [49]: 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]: rolling_mean(ser, 5)
Out[50]:
2000-01-01   NaN
2000-01-02   NaN

11.2. Moving (rolling) statistics / moments
For some windowing functions, additional parameters must be specified:

```python
In [51]: 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   -0.261998
2000-01-06   -0.230600
2000-01-07    0.121276
2000-01-08   -0.136220
2000-01-09   -0.057945
2000-01-10   -0.199326
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.

```python
In [52]: rolling_window(ser, 5, 'boxcar')
Out[52]:
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
```

```python
In [53]: 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
```

```python
In [54]: rolling_mean(ser, 5, center=True)
Out[54]:
```

Chapter 11. Computational tools
11.2.1 Binary rolling moments

`rolling_c 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`**: compute statistic for matching column names, returning a `DataFrame`

For example:

```python
In [55]: df2 = df[:20]

In [56]: rolling_corr(df2, df2['B'], window=5)
Out[56]:
A  B  C  D
2000-01-01 NaN NaN NaN NaN
2000-01-02 NaN NaN NaN NaN
2000-01-03 NaN NaN NaN NaN
2000-01-04 NaN NaN NaN NaN
2000-01-05 -0.262853 1 0.334449 0.193380
2000-01-06 -0.083745 1 -0.521587 -0.556126
2000-01-07 -0.292940 1 -0.658532 -0.458128
2000-01-08 0.840416 1 0.796505 -0.498672
2000-01-09 0.135275 1 0.753895 -0.634445
2000-01-10 0.346229 1 -0.682232 -0.645681
2000-01-11 0.365524 1 -0.775831 -0.561991
2000-01-12 0.204761 1 -0.855874 -0.382232
2000-01-13 0.575218 1 -0.747531 0.167892
2000-01-14 0.519499 1 -0.687277 0.192822
2000-01-15 0.048982 1 0.167669 -0.061463
... ... ... ...
[20 rows x 4 columns]
```

11.2.2 Computing rolling pairwise correlations

In financial data analysis and other fields it’s common to compute correlation matrices for a collection of time series. More difficult is to compute a moving-window correlation matrix. This can be done using the `rolling_corr_pairwise` function, which yields a `Panel` whose `items` are the dates in question:
In [57]: correls = rolling_corr_pairwise(df, 50)

In [58]: correls[df.index[-50]]
Out[58]:
        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

[4 rows x 4 columns]

You can efficiently retrieve the time series of correlations between two columns using \texttt{ix} indexing:

In [59]: correls.ix[:, 'A', 'C'].plot()
Out[59]: <matplotlib.axes.AxesSubplot at 0x5b5cb90>

11.3 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 [60]: rolling_mean(df, window=len(df), min_periods=1)[:5]
Out[60]:
        A      B      C      D
2000-01-01 -1.388345  3.317290  0.344542 -0.036968
2000-01-02 -1.123132  3.622300  1.675867  0.595300
2000-01-03 -0.628502  3.626503  2.455240  1.060158
2000-01-04 -0.768740  3.888917  2.451354  1.281874
2000-01-05 -0.824034 4.108035 2.556112 1.140723

In [61]: expanding_mean(df)[:5]
Out[61]:
   A      B      C      D
2000-01-01 -1.388345 3.317290 0.344542 -0.036968
2000-01-02 -1.123132 3.622300 1.675867 0.595300
2000-01-03 -0.628502 3.626503 2.455240 1.060158
2000-01-04 -0.768740 3.888917 2.451354 1.281874
2000-01-05 -0.824034 4.108035 2.556112 1.140723

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>
<tr>
<td>expanding_corr_pairwise</td>
<td>Pairwise correlation of DataFrame columns</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 [62]: ts.plot(style='k--')
Out[62]: <matplotlib.axes.AxesSubplot at 0x731fa50>

11.3. Expanding window moment functions
11.4 Exponentially weighted moment functions

A related set of functions are exponentially weighted versions of many of the above statistics. A number of EW (exponentially weighted) functions are provided using the blending method. For example, where \( y_t \) is the result and \( x_t \) the input, we compute an exponentially weighted moving average as

\[
y_t = (1 - \alpha)y_{t-1} + \alpha x_t
\]

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:

\[
\alpha = \begin{cases} 
\frac{2}{s + 1}, & s = \text{span} \\
\frac{1}{1 + c}, & c = \text{center of mass} \\
1 - \exp^{-\log 0.5 / h}, & h = \text{half life}
\end{cases}
\]

**Note:** the equation above is sometimes written in the form

\[
y_t = \alpha' y_{t-1} + (1 - \alpha') x_t
\]

where \( \alpha' = 1 - \alpha \).

You can pass one of the three to these functions but not more. 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 $\text{com} = 9.5$. Half-life is the period of time for the exponential weight to reduce to one half. Here is the list of functions available:

<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>
<tr>
<td>ewmcov</td>
<td>EW moving covariance</td>
</tr>
</tbody>
</table>

Here are an example for a univariate time series:

In [64]: plt.close('all')

In [65]: ts.plot(style='k--')
Out[65]: <matplotlib.axes.AxesSubplot at 0x7470f50>

In [66]: ewma(ts, span=20).plot(style='k')
Out[66]: <matplotlib.axes.AxesSubplot at 0x7470f50>

Note: The EW functions perform a standard adjustment to the initial observations whereby if there are fewer observations than called for in the span, those observations are reweighted accordingly.
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

12.1 Missing data basics

12.1.1 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:

\[
In [1]: df = DataFrame(randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   ...:  columns=['one', 'two', 'three'])
   ...:
\]

\[
In [2]: df['four'] = 'bar'
\]

\[
In [3]: df['five'] = df['one'] > 0
\]

\[
In [4]: df
Out[4]:
   one  two  three   four  five
  a  0.059117  1.138469 -2.400634  bar  True
  c -0.280853  0.025653 -1.386071  bar  False
  e  0.863937  0.252462  1.500571  bar  True
  f  1.053202 -2.338595  -0.374279  bar  True
  h -2.359958 -1.157886 -0.551865  bar  False
\]

\[
[5 rows x 5 columns]
\]

\[
In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
\]
12.1.2 Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While \( \text{NaN} \) 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 \( \text{None} \) will arise and we wish to also consider that “missing” or “null”.

Until recently, for legacy reasons \( \text{inf} \) and \( -\text{inf} \) were also considered to be “null” in computations. This is no longer the case by default; use the \texttt{mode.use_inf_as_null} option to recover it. To make detecting missing values easier (and across different array dtypes), pandas provides the \texttt{isnull()} and \texttt{notnull()} functions, which are also methods on \texttt{Series} objects:

```python
In [7]: df2['one']
Out[7]:
     a     0.059117
     b     NaN
     c    -0.280853
     d     NaN
     e     0.863937
     f    1.053202
     g    -2.359958
Name: one, dtype: float64
```

```python
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
```

```python
In [9]: df2['four'].notnull()
Out[9]:
     a   True
     b  False
     c   True
     d  False
     e   True
```

```
[8 rows x 5 columns]
```
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.

12.2 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  
  a  0.059117  1.138469  -2.400634  bar   True  2012-01-01
  c -0.280853  0.025653  -1.386071  bar  False  2012-01-01
  e  0.863937  0.252462  1.500571  bar   True  2012-01-01
  f  1.053202  -2.338595  -0.374279  bar   True  2012-01-01
  h -2.359958  -1.157886  -0.551865  bar  False  2012-01-01

[5 rows x 6 columns]

In [13]: df2.ix[['a','c','h'],[‘one’,’timestamp’]] = np.nan

In [14]: df2
Out[14]:
  one     two     three    four     five  timestamp  
  a  NaN     1.138469  -2.400634  bar   True  NaT
  c  NaN    -0.280853  -1.386071  bar  False   NaN
  e  NaN     0.863937  0.252462  1.500571  bar   True  NaT
  f  NaN    -1.053202  -2.338595  -0.374279  bar   True  NaT
  h  NaN     -2.359958  -1.157886  -0.551865  bar  False   NaN

[5 rows x 6 columns]

In [15]: df2.get_dtype_counts()
Out[15]:
bool     1
datetime64[ns]     1
float64     3
object     1
dtype: int64

12.3 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.
The descriptive statistics and computational methods discussed in the data structure overview (and listed here and here) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero
- If the data are all NA, the result will be NA
- Methods like cumsum and cumprod ignore NA values, but preserve them in the resulting arrays.

```python
In [16]: a
Out[16]:
   | one   | two  |
---|-------|------|
a  | NaN   | 1.138469 |
c  | NaN   | 0.025653 |
e  | 0.863937 | 0.252462 |
f  | 1.053202 | -2.338595 |
h  | 1.053202 | -1.157886 |

[5 rows x 2 columns]

In [17]: b
Out[17]:
   | one   | two  | three |
---|-------|------|-------|
a  | NaN   | 1.138469 | -2.400634 |
c  | NaN   | 0.025653 | -1.386071 |
e  | 0.863937 | 0.252462 | 1.500571 |
f  | 1.053202 | -2.338595 | -0.374279 |
h  | NaN   | -1.157886 | -0.551865 |

[5 rows x 3 columns]

In [18]: a + b
Out[18]:
   | one   | two  | three |
---|-------|------|-------|
a  | NaN   | NaN  | 2.276938 |
c  | NaN   | NaN  | 0.051306 |
e  | 1.727874 | NaN  | 0.504923 |
f  | 2.106405 | NaN  | -4.677190 |
h  | NaN   | NaN  | -2.315772 |

[5 rows x 3 columns]

In [19]: df
Out[19]:
   | one   | two  | three |
---|-------|------|-------|
a  | NaN   | 1.138469 | -2.400634 |
c  | NaN   | 0.025653 | -1.386071 |
e  | 0.863937 | 0.252462 | 1.500571 |
f  | 1.053202 | -2.338595 | -0.374279 |
h  | NaN   | -1.157886 | -0.551865 |

[5 rows x 3 columns]

In [20]: df['one'].sum()
Out[20]: 1.917139050150438

In [21]: df.mean(1)
Out[21]:
   | a   | c   |
---|-----|-----|
a  | -0.631082 |
c  | -0.680209 |

304 Chapter 12. Working with missing data
12.3.1 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example.

12.4 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

12.4.1 Filling missing values:fillna

The \texttt{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

\texttt{In [23]: df2.fillna(0)}

\texttt{Out[24]:}

\begin{verbatim}
  one   two   three   four   five  timestamp
a 0.000000 1.138469 -2.400634 bar   True  NaT
b 0.000000 0.025653 -1.386071 bar   False   NaN
c 0.863937 0.252462 1.500571 bar   True 2012-01-01
d 1.053202 -2.338595 -0.374279 bar   True 2012-01-01
h 0.000000 -1.157886 -0.551865 bar   False 1970-01-01
\end{verbatim}

[5 rows x 6 columns]
Fill gaps forward or backward

Using the same filling arguments as reindexing, we can propagate non-null values forward or backward:

In [26]: df
Out[26]:
<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NaN</td>
<td>1.138469</td>
<td>-2.400634</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>0.025653</td>
<td>-1.386071</td>
</tr>
<tr>
<td>e</td>
<td>0.863937</td>
<td>0.252462</td>
<td>1.500571</td>
</tr>
<tr>
<td>f</td>
<td>1.053202</td>
<td>-2.338595</td>
<td>-0.374279</td>
</tr>
<tr>
<td>h</td>
<td>NaN</td>
<td>-1.157886</td>
<td>-0.551865</td>
</tr>
</tbody>
</table>

[5 rows x 3 columns]

In [27]: df.fillna(method='pad')
Out[27]:
<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NaN</td>
<td>1.138469</td>
<td>-2.400634</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>0.025653</td>
<td>-1.386071</td>
</tr>
<tr>
<td>e</td>
<td>0.863937</td>
<td>0.252462</td>
<td>1.500571</td>
</tr>
<tr>
<td>f</td>
<td>1.053202</td>
<td>-2.338595</td>
<td>-0.374279</td>
</tr>
<tr>
<td>h</td>
<td>1.053202</td>
<td>-1.157886</td>
<td>-0.551865</td>
</tr>
</tbody>
</table>

[5 rows x 3 columns]

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 [28]: df
Out[28]:
<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NaN</td>
<td>1.138469</td>
<td>-2.400634</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>0.025653</td>
<td>-1.386071</td>
</tr>
<tr>
<td>e</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>f</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>h</td>
<td>NaN</td>
<td>-1.157886</td>
<td>-0.551865</td>
</tr>
</tbody>
</table>

[5 rows x 3 columns]

In [29]: df.fillna(method='pad', limit=1)
Out[29]:
<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NaN</td>
<td>1.138469</td>
<td>-2.400634</td>
</tr>
<tr>
<td>c</td>
<td>NaN</td>
<td>0.025653</td>
<td>-1.386071</td>
</tr>
<tr>
<td>e</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>f</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>h</td>
<td>NaN</td>
<td>-1.157886</td>
<td>-0.551865</td>
</tr>
</tbody>
</table>

[5 rows x 3 columns]

To remind you, these are the available filling methods:
### 12.4.2 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.

```python
In [30]: dff = DataFrame(np.random.randn(10,3),columns=list('ABC'))

In [31]: dff.iloc[3:5,0] = np.nan

In [32]: dff.iloc[4:6,1] = np.nan

In [33]: dff.iloc[5:8,2] = np.nan

In [34]: dff
Out[34]:
   A    B     C
0  1.592673  1.559318  1.562443
1  0.763264  0.162027 -0.902704
2  1.106010 -0.199234  0.458265
3  NaN      0.128594  1.147862
4  NaN      NaN      -2.417312
5  0.972827  NaN      0.223052
6  0.086926 -0.445645  NaN
7 -1.420361 -0.015601  NaN
8 -0.798334 -0.557697  0.381353
9  1.337122 -1.531095  1.331458
[10 rows x 3 columns]

In [35]: dff.fillna(dff.mean())
Out[35]:
   A    B     C
0  1.592673  1.559318  1.562443
1  0.763264  0.162027 -0.902704
2  1.106010 -0.199234  0.458265
3  0.455016  0.128594  1.147862
4  0.455016 -0.112417 -2.417312
5  0.972827 -0.112417  0.223052
6  0.086926 -0.445645  0.223052
7 -1.420361 -0.015601  0.223052
8 -0.798334 -0.557697  0.381353
9  1.337122 -1.531095  1.331458
[10 rows x 3 columns]

In [36]: dff.fillna(dff.mean()['B':'C'])
Out[36]:
   A    B     C
0  1.592673  1.559318  1.562443
1  0.763264  0.162027 -0.902704
2  1.106010 -0.199234  0.458265
3  0.455016  0.128594  1.147862
4  0.455016 -0.112417 -2.417312
5  0.972827 -0.112417  0.223052
6  0.086926 -0.445645  0.223052
7 -1.420361 -0.015601  0.223052
8 -0.798334 -0.557697  0.381353
9  1.337122 -1.531095  1.331458
[10 rows x 3 columns]
```
A  B  C
0 1.592673 1.559318 1.562443
1 0.763264 0.162027 -0.902704
2 1.106010 -0.199234 0.458265
3 NaN 0.128594 1.147862
4 NaN -0.112417 -2.417312
5 0.972827 -0.112417 0.223052
6 0.086926 -0.445645 0.223052
7 -1.420361 -0.015601 0.223052
8 -0.798334 -0.557697 0.381353
9 1.337122 -1.531095 1.331458
[10 rows x 3 columns]

New in version 0.13. Same result as above, but is aligning the ‘fill’ value which is a Series in this case.

In [37]: dff.where(notnull(dff),dff.mean(),axis='columns')
Out[37]:
   A        B        C
0 1.592673 1.559318 1.562443
1 0.763264 0.162027 -0.902704
2 1.106010 -0.199234 0.458265
3 0.455016 0.128594 1.147862
4 0.455016 -0.112417 -2.417312
5 0.972827 -0.112417 0.223052
6 0.086926 -0.445645 0.223052
7 -1.420361 -0.015601 0.223052
8 -0.798334 -0.557697 0.381353
9 1.337122 -1.531095 1.331458
[10 rows x 3 columns]

12.4.3 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 [38]: df
Out[38]:
   one   two   three
a NaN 1.138469 -2.400634
c NaN 0.025653 -1.386071
e NaN 0.000000 0.000000
f NaN 0.000000 0.000000
h NaN -1.157886 -0.551865
[5 rows x 3 columns]

In [39]: df.dropna(axis=0)
Out[39]:
Empty DataFrame
Columns: [one, two, three]
Index: []
[0 rows x 3 columns]

In [40]: df.dropna(axis=1)
Out[40]:
In [41]: df[‘one’].dropna()
Out[41]: Series([], name: one, dtype: float64)

`dropna` is presently only implemented for Series and DataFrame, but will be eventually added to Panel. Series.dropna is a simpler method as it only has one axis to consider. DataFrame.dropna has considerably more options, which can be examined in the API.

### 12.4.4 Interpolation

New in version 0.13.0. Both Series and Dataframe objects have an `interpolate` method that, by default, performs linear interpolation at missing datapoints.

In [42]: ts
Out[42]:
2000-01-31  0.469112
2000-02-29  NaN
2000-03-31  NaN
2000-04-28  NaN
2000-05-31  NaN
...
2007-11-30 -5.485119
2007-12-31 -6.854968
2008-01-31 -7.809176
2008-02-29 -6.346480
2008-03-31 -8.089641
2008-04-30 -8.916232
Freq: BM, Length: 100

In [43]: ts.count()
Out[43]: 61

In [44]: ts.interpolate().count()
Out[44]: 100

In [45]: plt.figure()
Out[45]: <matplotlib.figure.Figure at 0xd562bd0>

In [46]: ts.interpolate().plot()
Out[46]: <matplotlib.axes.AxesSubplot at 0xee7e850>
Index aware interpolation is available via the `method` keyword:

```python
In [47]: ts2
Out[47]:
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
```

```python
In [48]: ts2.interpolate()
Out[48]:
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
```

```python
In [49]: ts2.interpolate(method='time')
Out[49]:
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
```

For a floating-point index, use `method='values'`:

```python
In [50]: ser
Out[50]:
0    0
Name: 0, dtype: float64
```
1 NaN
10 10
dtype: float64

In [51]: ser.interpolate()
Out[51]:
0 0
1 5
10 10
dtype: int64

In [52]: ser.interpolate(method='values')
Out[52]:
0 0
1 1
10 10
dtype: int64

You can also interpolate with a DataFrame:

In [53]: 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 [54]: df
Out[54]:
     A   B
0    1.0  0.25
1   2.1   NaN
2    NaN   NaN
3    4.7   4.00
4    5.6  12.20
5    6.8  14.40

[6 rows x 2 columns]

In [55]: df.interpolate()
Out[55]:
     A   B
0    1.0  0.25
1   2.1  1.50
2  3.45  2.75
3   4.7  4.00
4   5.6  12.20
5   6.8  14.40

[6 rows x 2 columns]

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 [56]: df.interpolate(method='barycentric')
Out[56]:
   A    B
0  1.00  0.250
1  2.10 -7.660
2  3.53 -4.515
3  4.70  4.000
4  5.60 12.200
5  6.80 14.400
[6 rows x 2 columns]

In [57]: df.interpolate(method='pchip')
Out[57]:
   A            B
0  1.000000  0.250000
1  2.100000  1.130135
2  3.429309  2.337586
3  4.700000  4.000000
4  5.600000 12.200000
5  6.800000 14.400000
[6 rows x 2 columns]

When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

In [58]: df.interpolate(method='spline', order=2)
Out[58]:
   A            B
0  1.000000  0.250000
1  2.100000 -0.428598
2  3.404545  1.206900
3  4.700000  4.000000
4  5.600000 12.200000
5  6.800000 14.400000
[6 rows x 2 columns]

In [59]: df.interpolate(method='polynomial', order=2)
Out[59]:
   A            B
0  1.000000  0.250000
1  2.100000 -4.161538
2  3.547059 -2.911538
3  4.700000  4.000000
4  5.600000 12.200000
5  6.800000 14.400000
[6 rows x 2 columns]

Compare several methods:

In [60]: np.random.seed(2)
In [61]: ser = Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))
In [62]: bad = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])
In [63]: ser[bad] = np.nan

In [64]: methods = ['linear', 'quadratic', 'cubic']

In [65]: df = DataFrame({m: ser.interpolate(method=m) for m in methods})

In [66]: plt.figure()
Out[66]: <matplotlib.figure.Figure at 0xad08d10>

In [67]: df.plot()
Out[67]: <matplotlib.axes.AxesSubplot at 0xae1a210>

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.

In [68]: ser = Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [69]: new_index = ser.index + Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [70]: interp_s = ser.reindex(new_index).interpolate(method='pchip')

In [71]: interp_s[49:51]
Out[71]:
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:

```python
In [72]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])

In [73]: ser.interpolate(limit=2)
Out[73]:
   0  1
   1  3
   2  5
   3  7
   4 NaN
   5 11
```

### 12.4.5 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:

```python
In [74]: ser = Series([0., 1., 2., 3., 4.])

In [75]: ser.replace(0, 5)
Out[75]:
   0  5
   1  1
   2  2
   3  3
   4  4
```

You can replace a list of values by a list of other values:

```python
In [76]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[76]:
   0  4
   1  3
   2  2
   3  1
   4  0
```

You can also specify a mapping dict:

```python
In [77]: ser.replace({0: 10, 1: 100})
Out[77]:
   0  10
   1 100
   2  2
   3  3
   4  4
```

```
For a DataFrame, you can specify individual values by column:

In [78]: df = DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})
In [79]: df.replace({'a': 0, 'b': 5}, 100)
Out[79]:
   a  b
0 100 100
1  1  6
2  2  7
3  3  8
4  4  9
[5 rows x 2 columns]

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

In [80]: ser.replace([1, 2, 3], method='pad')
Out[80]:
0 0
1 0
2 0
3 0
4 4
dtype: float64

12.4.6 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 [81]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', nan, 'd']}
In [82]: df = DataFrame(d)
In [83]: df.replace('.', nan)
Out[83]:
   a  b  c
0  0  a  a
1  1  b  b
2  NaN NaN
3  3 NaN  d
[4 rows x 3 columns]

Now do it with a regular expression that removes surrounding whitespace (regex -> regex)

In [84]: df.replace(r'\s*\.\s*', nan, regex=True)
Out[84]:
   a  b  c
0  0  a  a
1  1  b  b
2  NaN NaN
3  3 NaN  d

12.4. Cleaning / filling missing data
Replace a few different values (list -> list)

```python
In [85]: df.replace(['a', '.'], ['b', nan])
Out[85]:
    a  b  c
0  0  b  b
1  1  b  b
2  NaN NaN
3  NaN  d
```

[4 rows x 3 columns]

list of regex -> list of regex

```python
In [86]: df.replace([r'.', r'(a)'], ['dot', '\1stuff'], regex=True)
Out[86]:
    a  b  c
0  {stuff} {stuff
1  b  b
2  dot NaN
3  dot  d
```

[4 rows x 3 columns]

Only search in column ‘b’ (dict -> dict)

```python
In [87]: df.replace({'b': '.'}, {'b': nan})
Out[87]:
    a  b  c
0  a  a
1  b  b
2  NaN NaN
3  NaN  d
```

[4 rows x 3 columns]

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict)

```python
In [88]: df.replace({'b': {'b': r''}}, regex=True)
Out[88]:
    a  b  c
0  a  a
1  b
2  . NaN
3  .  d
```

[4 rows x 3 columns]
or you can pass the nested dictionary like so

```python
In [90]: df.replace(regex={'b': {r's\s*\s*': nan}})
```

```
Out[90]:
      a   b   c
0    0   a   a
1    1   b   b
2  NaN  NaN  NaN
3  NaN   d   NaN
```

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

```python
In [91]: df.replace({'b': r'(\s*\.)\s*'}, {'b': r'\1ty'}, regex=True)
```

```
Out[91]:
      a   b   c
0    0   a   a
1    1   b   b
2  .ty  NaN  NaN
3  .ty   d   NaN
```

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex)

```python
In [92]: df.replace([r'(\s*\.)\s*', r'a|b'], nan, regex=True)
```

```
Out[92]:
      a   b   c
0  NaN  NaN  NaN
1  NaN  NaN  NaN
2  NaN  NaN  NaN
3   d   NaN  NaN
```

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

```python
In [93]: df.replace(regex=[r'(\s*\.)\s*', r'\(a|b\)'], value=nan)
```

```
Out[93]:
      a   b   c
0  NaN  NaN  NaN
1  NaN  NaN  NaN
2  NaN  NaN  NaN
3   d   NaN  NaN
```

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.

12.4. Cleaning / filling missing data
12.4.7 Numeric Replacement

Similar to `DataFrame.fillna`

```
In [94]: df = DataFrame(randn(10, 2))

In [95]: df[rand(df.shape[0]) > 0.5] = 1.5

In [96]: df.replace(1.5, nan)
Out[96]:
          0  1
0 -0.844214 -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4    NaN   NaN
5    NaN   NaN
6 -0.498174  1.060799
7  0.591667  0.183257
8  1.019855 -1.482465
9    NaN    NaN

[10 rows x 2 columns]
```

Replacing more than one value via lists works as well

```
In [97]: df00 = df.values[0, 0]

In [98]: df.replace([1.5, df00], [nan, 'a'])
Out[98]:
          0  1
0       a  -1.021415
1  0.4323957 -0.323580
2  0.4238247  0.799180
3  1.2626147  0.751965
4    NaN    NaN
5    NaN    NaN
6 -0.4981742  1.060799
7  0.5916665 -0.183257
8  1.0198550 -1.482465
9    NaN    NaN

[10 rows x 2 columns]
```

```
In [99]: df[1].dtype
Out[99]: dtype('float64')
```

You can also operate on the DataFrame in place
In [100]: df.replace(1.5, nan, inplace=True)

12.5 Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we’ve established some “casting rules” when reindexing will cause missing data to be introduced into, say, a Series or DataFrame. Here they are:

<table>
<thead>
<tr>
<th>data type</th>
<th>Cast to</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>float</td>
</tr>
<tr>
<td>boolean</td>
<td>object</td>
</tr>
<tr>
<td>float</td>
<td>no cast</td>
</tr>
<tr>
<td>object</td>
<td>no cast</td>
</tr>
</tbody>
</table>

For example:

In [101]: s = Series(randn(5), index=[0, 2, 4, 6, 7])

In [102]: s > 0
Out[102]:
0   True
2   True
4   True
6   True
7   True
dtype: bool

In [103]: (s > 0).dtype
Out[103]: dtype('bool')

In [104]: crit = (s > 0).reindex(list(range(8)))

In [105]: crit
Out[105]:
0   True
1   NaN
2   True
3   NaN
4   True
5   NaN
6   True
7   True
dtype: object

In [106]: crit.dtype
Out[106]: dtype('O')

Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

In [107]: reindexed = s.reindex(list(range(8))).fillna(0)

In [108]: reindexed[crit]
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)

However, these can be filled in using `fillna` and it will work fine:

```
In [109]: reindexed[crit.fillna(False)]
Out[109]:
   0    0.126504
   2    0.696198
   4    0.697416
   6    0.601516
   7    0.003659
   dtype: float64
```

```
In [110]: reindexed[crit.fillna(True)]
Out[110]:
   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
13.1 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 \(\rightarrow\) 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:

```python
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.

### 13.1.1 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:

```python
In [13]: df.groupby('A').groups
Out[13]: {'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}
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:

```python
In [15]: grouped = df.groupby(['A', 'B'])
In [16]: grouped.groups
Out[16]:
{(‘bar’, ‘one’): [1],
```

---

13.1. Splitting an object into groups
pandas: powerful Python data analysis toolkit, Release 0.13.1

('bar', 'three'): [3],
('bar', 'two'): [5],
('foo', 'one'): [0, 6],
('foo', 'three'): [7],
('foo', 'two'): [2, 4]}

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

[2 rows x 1 columns]

In [20]: df2.groupby(['X'], sort=False).sum()
Out[20]:
Y
  X  
B   3
A   7

[2 rows x 1 columns]

GroupBy will tab complete column names (and other attributes)

In [21]: df
Out[21]:
      gender  height  weight
2000-01-01  male   42.85  157.50
2000-01-02  male   49.61  177.34
2000-01-03  male   56.29  171.52
2000-01-04  female 48.42  144.25
2000-01-05  male   46.56  152.53
2000-01-06  female 68.45  168.27
2000-01-07  female 70.76  136.43
2000-01-08  female 58.90  176.49
2000-01-09  female 76.43  174.09
2000-01-10  male   45.31  177.54

[10 rows x 3 columns]

In [22]: gb = df.groupby('gender')

In [23]: gb.<TAB>
13.1.2 GroupBy with MultiIndex

With hierarchically-indexed data, it’s quite natural to group by one of the levels of the hierarchy.

```python
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
```

```python
In [25]: grouped = s.groupby(level=0)
```

```python
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:

```python
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:

```python
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.

```python
In [29]: s
Out[29]:
   first    second    third
  bar  doo  one  1.346061
two  1.511763
  baz  bee  one  1.627081
two  2.661349
  foo  bop  one   0.441652
two  1.211526
  qux  bop  one  0.268520
```

13.1. Splitting an object into groups
In [30]: s.groupby(level=['first','second']).sum()
Out[30]:
fist  second
bar  doo  2.857824
baz  bee  0.636499
foo  bop  0.769873
qux  bop  0.293100
dtype: float64

More on the sum function and aggregation later.

### 13.1.3 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:

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:

In [34]: df['C'].groupby(df['A'])
Out[34]: <pandas.core.groupby.SeriesGroupBy object at 0x5dcea90>

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

### 13.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to itertools.groupby:

In [35]: grouped = df.groupby('A')

In [36]: for name, group in grouped:
.....:     print(name)
.....:     print(group)
.....:
bar
   A    B    C    D
 0 1 bar  one -0.042379 -0.089329
 1 3 bar  three -0.009920 -0.945867
 2 5 bar  two  0.495767  1.956030

[3 rows x 4 columns]
foo
   A    B    C    D
 0 0 foo  one  0.919854 -1.131345
 1 2 foo  two  1.247642  0.337863
 2 4 foo  two  0.290213  0.932132
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')
   A   B   C   D
   1   bar one -0.042379 -0.089329
[1 rows x 4 columns]
('bar', 'three')
   A   B   C   D
   3   bar three -0.00992 -0.945867
[1 rows x 4 columns]
('bar', 'two')
   A   B   C   D
   5   bar two  0.495767  1.95603
[1 rows x 4 columns]
('foo', 'one')
   A   B   C   D
   0   foo one  0.919854  1.131345
   6   foo one  0.362949  0.017587
[2 rows x 4 columns]
('foo', 'three')
   A   B   C   D
   7   foo three  1.548106 -0.016692
[1 rows x 4 columns]
('foo', 'two')
   A   B   C   D
   2   foo two  1.247642  0.337863
   4   foo two  0.290213 -0.932132
[2 rows x 4 columns]
```

It’s standard Python-fu but remember you can unpack the tuple in the for loop statement if you wish: `for (k1, k2), group in grouped:`.

### 13.3 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 [38]: grouped = df.groupby('A')

In [39]: grouped.aggregate(np.sum)
Out[39]:
```
In [40]: grouped = df.groupby(['A', 'B'])

In [41]: grouped.aggregate(np.sum)
Out[41]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>-0.042379 -0.089329</td>
<td></td>
</tr>
<tr>
<td></td>
<td>three -0.009920 -0.945867</td>
<td></td>
</tr>
<tr>
<td></td>
<td>two  0.495767 1.956030</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>-0.556905 -1.113758</td>
<td></td>
</tr>
<tr>
<td></td>
<td>three 1.548106 -0.016692</td>
<td></td>
</tr>
<tr>
<td></td>
<td>two  1.537855 -0.594269</td>
<td></td>
</tr>
</tbody>
</table>

[6 rows x 2 columns]

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 [42]: grouped = df.groupby(['A', 'B'], as_index=False)

In [43]: grouped.aggregate(np.sum)
Out[43]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>one</td>
<td>-0.042379 -0.089329</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>three</td>
<td>-0.009920 -0.945867</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bar</td>
<td>two</td>
<td>0.495767 1.956030</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>one</td>
<td>-0.556905 -1.113758</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>three</td>
<td>1.548106 -0.016692</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>foo</td>
<td>two</td>
<td>1.537855 -0.594269</td>
<td></td>
</tr>
</tbody>
</table>

[6 rows x 4 columns]

In [44]: df.groupby('A', as_index=False).sum()
Out[44]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>0.443469 0.920834</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>foo</td>
<td>2.529056 -1.724719</td>
<td></td>
</tr>
</tbody>
</table>

[2 rows x 3 columns]

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 [45]: df.groupby(['A', 'B']).sum().reset_index()
Out[45]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>one</td>
<td>-0.042379 -0.089329</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>three</td>
<td>-0.009920 -0.945867</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bar</td>
<td>two</td>
<td>0.495767 1.956030</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>one</td>
<td>-0.556905 -1.113758</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>three</td>
<td>1.548106 -0.016692</td>
<td></td>
</tr>
</tbody>
</table>
5  foo  two  1.537855  -0.594269

[6 rows x 4 columns]

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 [46]: grouped.size()
Out[46]:
A   B
bar one  1
     three  1
two   1
foo one  2
     three  1
two   2
dtype: int64
```

### 13.3.1 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 [47]: grouped = df.groupby('A')
In [48]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[48]:
     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 [49]: grouped['D'].agg({'result1' : np.sum, 'result2' : np.mean})
Out[49]:
     result2  result1
   A     
bar  0.306945  0.920834
foo -0.344944 -1.724719
```

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 [50]: grouped.agg([np.sum, np.mean, np.std])
Out[50]:
   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.

### 13.3.2 Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```
In [51]: grouped.agg({'C' : np.sum,
              ....:        'D' : lambda x: np.std(x, ddof=1)})
```

```
Out[51]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.443469</td>
<td>1.490982</td>
</tr>
<tr>
<td>A</td>
<td>2.529056</td>
<td>0.645875</td>
</tr>
</tbody>
</table>
```

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 [52]: grouped.agg({'C' : 'sum', 'D' : 'std'})
```

```
Out[52]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.443469</td>
<td>1.490982</td>
</tr>
<tr>
<td>A</td>
<td>2.529056</td>
<td>0.645875</td>
</tr>
</tbody>
</table>
```

### 13.3.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, and `std`, have optimized Cython implementations:

```
In [53]: df.groupby('A').sum()
```

```
Out[53]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.443469</td>
<td>0.920834</td>
</tr>
<tr>
<td>A</td>
<td>2.529056</td>
<td>1.724719</td>
</tr>
</tbody>
</table>
```

```
In [54]: df.groupby(['A', 'B']).mean()
```

```
Out[54]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>-0.042379</td>
<td>-0.089329</td>
</tr>
<tr>
<td>A B</td>
<td>-0.009920</td>
<td>-0.945867</td>
</tr>
<tr>
<td>A B</td>
<td>0.495767</td>
<td>1.956030</td>
</tr>
<tr>
<td>A B</td>
<td>-0.278452</td>
<td>-0.556879</td>
</tr>
<tr>
<td>A B</td>
<td>1.548106</td>
<td>-0.016692</td>
</tr>
<tr>
<td>A B</td>
<td>0.768928</td>
<td>-0.297134</td>
</tr>
</tbody>
</table>
```

[6 rows x 2 columns]
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).

## 13.4 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
In [55]: index = date_range('10/1/1999', periods=1100)

In [56]: ts = Series(np.random.normal(0.5, 2, 1100), index)

In [57]: ts = rolling_mean(ts, 100, 100).dropna()

In [58]: ts.head()
Out[58]:
2000-01-08 0.779333
2000-01-09 0.778852
2000-01-10 0.786476
2000-01-11 0.782797
2000-01-12 0.798110
Freq: D, dtype: float64

In [59]: ts.tail()
Out[59]:
2002-09-30 0.660294
2002-10-01 0.631095
2002-10-02 0.673601
2002-10-03 0.709213
2002-10-04 0.719369
Freq: D, dtype: float64

In [60]: key = lambda x: x.year

In [61]: zscore = lambda x: (x - x.mean()) / x.std()

In [62]: transformed = ts.groupby(key).transform(zscore)

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

# Original Data
In [63]: grouped = ts.groupby(key)

In [64]: grouped.mean()
Out[64]:
2000 0.442441
2001 0.526246
2002 0.459365
dtype: float64

In [65]: grouped.std()
Out[65]:
2000 0.131752
2001 0.210945
2002 0.128753
```
dtype: float64

# Transformed Data
In [66]: grouped_trans = transformed.groupby(key)

In [67]: grouped_trans.mean()
Out[67]:
2000  1.146560e-15
2001  1.504428e-15
2002  1.675355e-15
dtype: float64

In [68]: grouped_trans.std()
Out[68]:
2000  1.000000e+00
2001  1.000000e+00
2002  1.000000e+00
dtype: float64

We can also visually compare the original and transformed data sets.

In [69]: compare = DataFrame({'Original': ts, 'Transformed': transformed})

In [70]: compare.plot()
Out[70]: <matplotlib.axes.AxesSubplot at 0x7c23950>

```

Another common data transform is to replace missing data with the group mean.

```
In [71]: data_df
Out[71]:
    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  0.367816 -0.469478
6 -0.030651 -1.729858  0.392067
7 -0.347401 -0.429063  1.792958
8 -0.431059  1.605289 -3.302946
9  0.434332 -1.302198  0.756527
10 -0.349926  NaN       0.304228
11  NaN     -0.024779  NaN
12  1.026076 -0.151723 -1.136601
13  0.611215 -0.897508  0.022300
...  ...       ... 
[1000 rows x 3 columns]

In [72]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [73]: key = countries[np.random.randint(0, 4, 1000)]

In [74]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [75]: grouped.count()
Out[75]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>209</td>
<td>217</td>
<td>189</td>
</tr>
<tr>
<td>JP</td>
<td>240</td>
<td>255</td>
<td>217</td>
</tr>
<tr>
<td>UK</td>
<td>216</td>
<td>231</td>
<td>193</td>
</tr>
<tr>
<td>US</td>
<td>239</td>
<td>250</td>
<td>217</td>
</tr>
</tbody>
</table>

[4 rows x 3 columns]

In [76]: f = lambda x: x.fillna(x.mean())

In [77]: 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 [78]: grouped_trans = transformed.groupby(key)

In [79]: grouped.mean() # original group means
Out[79]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>-0.098371</td>
<td>-0.015420</td>
<td>0.068053</td>
</tr>
<tr>
<td>JP</td>
<td>0.069025</td>
<td>0.023100</td>
<td>-0.077324</td>
</tr>
<tr>
<td>UK</td>
<td>0.034069</td>
<td>-0.052580</td>
<td>-0.116525</td>
</tr>
<tr>
<td>US</td>
<td>0.058664</td>
<td>-0.020399</td>
<td>0.028603</td>
</tr>
</tbody>
</table>

[4 rows x 3 columns]

In [80]: grouped_trans.mean() # transformation did not change group means
Out[80]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>-0.098371</td>
<td>-0.015420</td>
<td>0.068053</td>
</tr>
<tr>
<td>JP</td>
<td>0.069025</td>
<td>0.023100</td>
<td>-0.077324</td>
</tr>
</tbody>
</table>

13.4. Transformation
UK  0.034069 -0.052580  -0.116525
US  0.058664  -0.020399   0.028603

[4 rows x 3 columns]

In [81]: grouped.count()  # original has some missing data points
Out[81]:
    A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

[4 rows x 3 columns]

In [82]: grouped_trans.count()  # counts after transformation
Out[82]:
    A    B    C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258

[4 rows x 3 columns]

In [83]: grouped_trans.size()  # Verify non-NA count equals group size
Out[83]:
    GR    JP    UK    US
dtype: int64

13.5 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.

In [84]: sf = Series([1, 1, 2, 3, 3, 3])

In [85]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[85]:
    3
    3
    4
    3
    5
    3
dtype: int64

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

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

In [86]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc'))

In [87]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[87]:
      A  B
    2  2 b
Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [88]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[88]:
         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
[8 rows x 2 columns]
```

For dataframes with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [89]: dff['C'] = np.arange(8)
In [90]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[90]:
          A    B    C
2    NaN    2    2
3    NaN    3    3
4     4    4    4
5     5    5    5
[4 rows x 3 columns]
```

### 13.6 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 [91]: grouped = df.groupby('A')

In [92]: grouped.agg(lambda x: x.std())
Out[92]:
         B    C    D
A
bar  0.301765 1.490982
foo  0.966450 0.645875
[2 rows x 3 columns]
```

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 [93]: grouped = df.groupby('A')

In [94]: grouped.agg(lambda x: x.std())
Out[94]:
         B    C    D
A
bar  0.301765 1.490982
foo  0.966450 0.645875
[2 rows x 3 columns]
```

### 13.6. 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 [91]: grouped = df.groupby('A')

In [92]: grouped.agg(lambda x: x.std())
Out[92]:
         B    C    D
A
bar  0.301765 1.490982
foo  0.966450 0.645875
[2 rows x 3 columns]
```

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 [93]: grouped = df.groupby('A')

In [94]: grouped.agg(lambda x: x.std())
Out[94]:
         B    C    D
A
bar  0.301765 1.490982
foo  0.966450 0.645875
[2 rows x 3 columns]
```

### 13.6. 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 [91]: grouped = df.groupby('A')

In [92]: grouped.agg(lambda x: x.std())
Out[92]:
         B    C    D
A
bar  0.301765 1.490982
foo  0.966450 0.645875
[2 rows x 3 columns]
```

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:
What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```python
In [94]: tsdf = DataFrame(randn(1000, 3),
                   index=date_range('1/1/2000', periods=1000),
                   columns=['A', 'B', 'C'])
In [95]: tsdf ix[::2] = np.nan
In [96]: grouped = tsdf.groupby(lambda x: x.year)
In [97]: grouped.fillna(method='pad')
```

In this example, we chopped the collection of time series into yearly chunks then independently called `fillna` on the groups.

### 13.7 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 [98]: df
Out[98]:
   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

[8 rows x 4 columns]

In [99]: grouped = df.groupby('A')

# could also just call .describe()

In [100]: grouped['C'].apply(lambda x: x.describe())
Out[100]:
   A
bar  count       3.000000
       mean     0.147823
       std      0.301765
       min    -0.042379
       25%    -0.026149
       ...  
foo  std      0.966450
       min   -0.919854
       25%    0.290213
       50%    0.362949
       75%    1.247642
       max    1.548106
Length: 16, dtype: float64

The dimension of the returned result can also change:

In [101]: grouped = df.groupby('A')['C']

In [102]: def f(group):
       ....:     return DataFrame({'original' : group,
       ....:                      'demeaned' : group - group.mean()})
       ....:

In [103]: grouped.apply(f)
Out[103]:
   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

[8 rows x 2 columns]

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

13.7. Flexible apply
In [104]: def f(x):
   .....:     return Series([ x, x**2 ], index = [‘x’, ‘x^s’])
   .....:

In [105]: s
Out[105]:
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 [106]: s.apply(f)
Out[106]:
x    x^s
first second third
bar  doo  one  1.346061  1.811881
     two  1.511763  2.285426
baz  bee  one  1.627081  2.647393
     two  -0.990582  0.981252
foo  bop  one  -0.441652  0.195057
     two   1.211526  1.467795
qux  bop  one   0.268520  0.072103
     two   0.024580  0.000604
[8 rows x 2 columns]

13.8 Other useful features

13.8.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:

In [107]: df
Out[107]:
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>0</td>
<td>foo</td>
<td>one</td>
<td>-0.919854</td>
<td>-1.131345</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>one</td>
<td>-0.042379</td>
<td>-0.089329</td>
</tr>
<tr>
<td>2</td>
<td>foo</td>
<td>two</td>
<td>1.247642</td>
<td>0.337863</td>
</tr>
<tr>
<td>3</td>
<td>bar</td>
<td>three</td>
<td>-0.009920</td>
<td>-0.945867</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>two</td>
<td>0.290213</td>
<td>-0.932132</td>
</tr>
<tr>
<td>5</td>
<td>bar</td>
<td>two</td>
<td>0.495767</td>
<td>1.956030</td>
</tr>
<tr>
<td>6</td>
<td>foo</td>
<td>one</td>
<td>0.362949</td>
<td>0.017587</td>
</tr>
<tr>
<td>7</td>
<td>foo</td>
<td>three</td>
<td>1.548106</td>
<td>-0.016692</td>
</tr>
</tbody>
</table>

[8 rows x 4 columns]

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:
13.8.2 NA group handling

If there are any NaN values in the grouping key, these will be automatically excluded. So there will never be an “NA 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).

13.8.3 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:

```python
In [109]: data = Series(np.random.randn(100))

In [110]: factor = qcut(data, [0, .25, .5, .75, 1.])

In [111]: data.groupby(factor).mean()
```

```bash
Out[111]:

[-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
```

13.8.4 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 [112]: df = pd.DataFrame(list('aaabba'), columns=['A'])

In [113]: df
Out[113]:
          A
0      a
1      a
2      a
3      b
4      b
5      a
[6 rows x 1 columns]

In [114]: df.groupby('A').cumcount()
```

```bash
Out[114]:

0  0
1  1
```
In [115]: df.groupby('A').cumcount(ascending=False)  # kwarg only
Out[115]:
0  3
1  2
2  1
3  1
4  0
5  0
dtype: int64
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.

### 14.1 Concatenating objects

The `concat` function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say “if any” because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```python
In [1]: df = DataFrame(np.random.randn(10, 4))

In [2]: df
Out[2]:
        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  0.407605  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
[10 rows x 4 columns]

# break it into pieces
In [3]: pieces = [df[:3], df[3:7], df[7:]]

In [4]: concatenated = concat(pieces)

In [5]: concatenated
Out[5]:
        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  0.407605  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```

[341]
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”:

```
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]: concatenated = concat(pieces, keys=['first', 'second', 'third'])
```

```
Out[7]:
```

```
 0 1 2 3
first 0 -0.673690 0.113648 -1.478427 0.524988
1 6 0.404705 0.577046 -1.715002 -1.039268
2 7 -0.370647 -1.157892 -1.344312 0.844885
3 8 1.075770 -0.109050 1.643563 -1.469388
4 9 0.357021 -0.674600 -1.776904 -0.968914
5 10 rows x 4 columns
```

Chapter 14. Merge, join, and concatenate
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 [8]: concatenated.ix['second']
Out [8]:
   0   1   2   3
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
```

[4 rows x 4 columns]

It’s not a stretch to see how this can be very useful. More detail on this functionality below.

### 14.1.1 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:

```
In [9]: from pandas.util.testing import rands
   ...
In [10]: df = DataFrame(np.random.randn(10, 4), columns=['a', 'b', 'c', 'd'],
                      index=[rands(5) for _ in range(10)])
   ...
In [11]: df
Out [11]:
   a     b     c     d
0  6.1741  0.294524  0.413738  0.276662
1  0.13960 -0.362543 -0.006154 -0.923061
2  0.895717  0.805244  2.120412 -2.565646
3  1.431256  1.340309  1.170299 -0.226169
4  0.410835  0.813850  0.132003 -0.827317
5 -0.076467 -1.187678  1.130127  1.436737
6 -1.413681  1.607920  1.024180  0.569605
7  0.875906 -2.211372  0.974466  2.006747
8  0.410001 -0.078638  0.545952 -1.219217
9 -1.226825  0.769804 -1.281247 -0.727707
```

[10 rows x 4 columns]

```
In [12]: concat([(df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
                    df.ix[-7:, ['d']]),], axis=1)
   ...
Out [12]:
```

---

14.1. Concatenating objects 343
Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```python
In [13]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
              ...:       df.ix[-7:, ['d']]], axis=1, join='inner')
```

```
Out[13]:
   a     b     c     d
BmVox 1.431256 1.340309 -1.170299 -0.226169
qp7p7 0.410835 0.813850 0.132003 -0.827317
k3K2f -0.076467 -1.187678 1.130127 -1.436737
HGqMS -1.413681 1.607920 1.024180 0.569605

[4 rows x 4 columns]
```

Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:

```python
In [14]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
              ...:       df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
```

```
Out[14]:
   a     b     c     d
6I74i -1.294524 0.413738  NaN  NaN
RP8O8 -0.013960 -0.362543  NaN  NaN
lTKuy 0.895717 0.805244 -1.206412  NaN
BmVox 1.431256 1.340309 -1.170299 -0.226169
qp7p7 0.410835 0.813850 0.132003 -0.827317
k3K2f -0.076467 -1.187678 1.130127 -1.436737
HGqMS -1.413681 1.607920 1.024180 0.569605
Xby44  NaN  NaN  0.974466 -2.006747
PL692  NaN  NaN  NaN  -1.219217
AZAf4  NaN  NaN  NaN  -0.727707

[10 rows x 4 columns]
```

### 14.1.2 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:

```python
In [15]: s = Series(randn(10), index=np.arange(10))
```

```python
In [16]: s1 = s[:5] # note we’re slicing with labels here, so 5 is included
```
In [17]: s2 = s[6:]

In [18]: s1.append(s2)
Out[18]:
0   -0.121306
1    -0.097883
2     0.695775
3     0.341734
4     0.959726
5   -0.619976
6    0.149748
7   -0.732339
8     0.687738
9     dtype: float64

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

In [19]: df = DataFrame(randn(6, 4), index=date_range('1/1/2000', periods=6),
                        columns=['A', 'B', 'C', 'D'])

In [20]: df1 = df.ix[:3]

In [21]: df2 = df.ix[3:, :3]

In [22]: df1
Out[22]:
          A      B      C      D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
[3 rows x 4 columns]

In [23]: df2
Out[23]:
          A      B      C
2000-01-04  0.690579  0.995761  2.396780
2000-01-05  3.357427 -0.317441 -1.236269
2000-01-06 -0.487602 -0.082240 -2.182937
[3 rows x 3 columns]

In [24]: df1.append(df2)
Out[24]:
          A      B      C      D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
2000-01-04  0.690579  0.995761  2.396780  NaN
2000-01-05  3.357427 -0.317441 -1.236269  NaN
2000-01-06 -0.487602 -0.082240 -2.182937  NaN
[6 rows x 4 columns]

append may take multiple objects to concatenate:

In [25]: df1 = df.ix[:2]
In [26]: df2 = df.ix[2:4]

In [27]: df3 = df.ix[4:]

In [28]: df1.append([df2,df3])

Out[28]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.176444</td>
<td>0.403310</td>
<td>-0.154951</td>
<td>0.301624</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>2.179861</td>
<td>-1.369849</td>
<td>-0.954208</td>
<td>1.462696</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>1.743161</td>
<td>-0.826591</td>
<td>-0.345352</td>
<td>1.314232</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.690579</td>
<td>0.995761</td>
<td>2.396780</td>
<td>0.014871</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>3.357427</td>
<td>-0.317441</td>
<td>-1.236269</td>
<td>0.896171</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.487602</td>
<td>-0.082240</td>
<td>-2.182937</td>
<td>0.380396</td>
</tr>
</tbody>
</table>

[6 rows x 4 columns]

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.

### 14.1.3 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:

In [29]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])

In [30]: df2 = DataFrame(randn(3, 4), columns=['A', 'B', 'C', 'D'])

In [31]: df1

Out[31]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.084844</td>
<td>0.432390</td>
<td>1.519970</td>
<td>-0.493662</td>
</tr>
<tr>
<td>1</td>
<td>0.600178</td>
<td>0.274230</td>
<td>0.132885</td>
<td>-0.023688</td>
</tr>
<tr>
<td>2</td>
<td>2.410179</td>
<td>1.450520</td>
<td>0.206053</td>
<td>-0.251905</td>
</tr>
<tr>
<td>3</td>
<td>-2.213588</td>
<td>1.063327</td>
<td>1.266143</td>
<td>0.299368</td>
</tr>
<tr>
<td>4</td>
<td>-0.863838</td>
<td>0.408204</td>
<td>-1.048089</td>
<td>-0.025747</td>
</tr>
<tr>
<td>5</td>
<td>-0.988387</td>
<td>0.094055</td>
<td>1.262731</td>
<td>1.289997</td>
</tr>
</tbody>
</table>

[6 rows x 4 columns]

In [32]: df2

Out[32]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.882423</td>
<td>-0.055758</td>
<td>0.536580</td>
<td>-0.489682</td>
</tr>
<tr>
<td>1</td>
<td>0.369374</td>
<td>-0.034571</td>
<td>-2.484478</td>
<td>-0.281461</td>
</tr>
<tr>
<td>2</td>
<td>0.030711</td>
<td>0.109121</td>
<td>1.126203</td>
<td>-0.977349</td>
</tr>
</tbody>
</table>

[3 rows x 4 columns]

To do this, use the ignore_index argument:

In [33]: concat([df1, df2], ignore_index=True)

Out[33]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.084844</td>
<td>0.432390</td>
<td>1.519970</td>
<td>-0.493662</td>
</tr>
<tr>
<td>1</td>
<td>0.600178</td>
<td>0.274230</td>
<td>0.132885</td>
<td>-0.023688</td>
</tr>
</tbody>
</table>
2  2.410179  1.450520  0.206053 -0.251905
3  -2.213588  1.063327  1.266143  0.299368
4  -0.863838  0.408204  -1.048089  -0.025747
5  -0.988387  0.094055  1.262731  1.289997
6   0.082423  -0.055758  0.536580  -0.489682
7   0.369374  -0.034571  -2.484478  -0.281461
8   0.030711   0.109121  1.126203  -0.977349

[9 rows x 4 columns]

This is also a valid argument to DataFrame.append:

In [34]: df1.append(df2, ignore_index=True)
Out[34]:
    A        B        C        D
0  0.084844  0.432390  1.519970 -0.493662
1  0.600178  0.274230  0.132885  0.023688
2  2.410179  1.450520  0.206053  0.251905
3 -2.213588  1.063327  1.266143  0.299368
4 -0.863838  0.408204 -1.048089  0.025747
5 -0.988387  0.094055  1.262731  1.289997
6  0.082423  0.055758  0.536580  0.489682
7  0.369374  0.034571 -2.484478  0.281461
8  0.030711  0.109121  1.126203  0.977349

[9 rows x 4 columns]

14.1.4 More concatenating with group keys

Let’s consider a variation on the first example presented:

In [35]: df = DataFrame(np.random.randn(10, 4))
In [36]: df
Out[36]:
    0       1       2       3
0  1.474071 -0.064034 -1.282782  0.781836
1  1.071357  0.441153  2.353925  0.583787
2  0.221471  0.744471  0.758527  1.729689
3  0.964980 -0.845696 -1.340896  1.846883
4  1.328865  1.682706  1.717693  0.888782
5  0.228440  0.901805  1.171216  0.520260
6  1.197071 -1.066969  0.303421  0.858447
7  0.306996 -0.028665  0.384316  1.574159
8  1.588931  0.476720  0.473424  0.242861
9  0.014805 -0.284319  0.650776 -1.461665
[10 rows x 4 columns]

# break it into pieces
In [37]: pieces = [df.ix[:, [0, 1]], df.ix[:, [2]], df.ix[:, [3]]]
In [38]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'])
In [39]: result
Out[39]:
    one    two    three
0  1.474071  -0.064034  -1.282782  0.781836
1  1.071357   0.441153   2.353925  0.583787
2  0.221471   0.744471   0.758527  1.729689
3  0.964980  -0.845696  -1.340896  1.846883
4  1.328865   1.682706  -1.717693  0.888782
5  0.228440   0.901805  1.171216  0.520260
6  1.197071  -1.066969  -0.303421  0.858447
7  0.306996  -0.028665   0.384316  1.574159
8  1.588931   0.476720   0.473424  0.242861
9  0.014805  -0.284319   0.650776  1.461665

14.1. Concatenating objects
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 [40]: pieces = {'one': df.ix[:, [0, 1]],
              ....:     'two': df.ix[:, [2]],
              ....:     'three': df.ix[:, [3]]}

In [41]: concat(pieces, axis=1)
Out[41]:
   one  three  two  
 0  1.474071 -1.282782 0.781836  
 1 -1.071357  2.353925 0.583787  
 2  0.221471 -0.744471 1.729689  
 3 -0.964980 -0.845696 1.846883  
 4 -1.328865  1.682706 -1.717693  
 5  0.228440  0.901805 1.171216  
 6 -1.197071 -0.030421 -0.858447  
 7  0.306996 -0.028665 0.384316  
 8  1.588931  0.473424 -0.242861  
 9 -0.014805 -0.284319 0.650776  
[10 rows x 4 columns]
```

```
In [42]: concat(pieces, keys=['three', 'two'])
Out[42]:
   three  two  
 2  NaN  NaN  
 3  NaN  NaN  
 4  NaN  NaN  
 5  NaN  NaN  
 6  NaN  NaN  
 7  NaN  NaN  
 8  NaN  NaN  
 9  NaN  NaN  
```

[10 rows x 4 columns]
The MultiIndex created has levels that are constructed from the passed keys and the columns of the DataFrame pieces:

```
In [43]: result.columns.levels
Out[43]: FrozenList([u'one', u'two', u'three'], [0, 1, 2, 3])
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

```
In [44]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'],
                        levels=[['three', 'two', 'one', 'zero'],
                                names=['group_key'])
```

```
In [45]: result
Out[45]:
<table>
<thead>
<tr>
<th>group_key</th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.474071</td>
<td>-0.064034</td>
<td>-1.282782</td>
</tr>
<tr>
<td>1</td>
<td>-1.071357</td>
<td>0.441153</td>
<td>2.353925</td>
</tr>
<tr>
<td>2</td>
<td>0.221471</td>
<td>-0.744471</td>
<td>0.758527</td>
</tr>
<tr>
<td>3</td>
<td>-0.964980</td>
<td>-0.845696</td>
<td>-1.340896</td>
</tr>
<tr>
<td>4</td>
<td>-1.328865</td>
<td>1.682706</td>
<td>-1.717693</td>
</tr>
<tr>
<td>5</td>
<td>0.228440</td>
<td>0.901805</td>
<td>1.171216</td>
</tr>
<tr>
<td>6</td>
<td>-1.197071</td>
<td>-1.066969</td>
<td>-0.303421</td>
</tr>
<tr>
<td>7</td>
<td>0.306996</td>
<td>-0.028665</td>
<td>0.384316</td>
</tr>
<tr>
<td>8</td>
<td>1.588931</td>
<td>0.476720</td>
<td>0.473424</td>
</tr>
<tr>
<td>9</td>
<td>-0.014805</td>
<td>-0.284319</td>
<td>0.650776</td>
</tr>
</tbody>
</table>

[10 rows x 4 columns]
```

```
In [46]: result.columns.levels
Out[46]: FrozenList([u'three', u'two', u'one', u'zero'], [0, 1, 2, 3])
```

Yes, this is fairly esoteric, but is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

### 14.1.5 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 [47]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])
```

```
In [48]: df
Out[48]:
<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>-0.137707</td>
<td>-0.891060</td>
<td>-0.693921</td>
<td>1.613616</td>
</tr>
<tr>
<td>1</td>
<td>0.464000</td>
<td>0.227371</td>
<td>-0.496922</td>
<td>0.306389</td>
</tr>
<tr>
<td>2</td>
<td>-2.290613</td>
<td>-1.134623</td>
<td>-1.561819</td>
<td>-0.260838</td>
</tr>
<tr>
<td>3</td>
<td>0.281957</td>
<td>1.523962</td>
<td>-0.902937</td>
<td>0.068159</td>
</tr>
<tr>
<td>4</td>
<td>-0.057873</td>
<td>-0.368204</td>
<td>-1.144073</td>
<td>0.861209</td>
</tr>
<tr>
<td>5</td>
<td>0.800193</td>
<td>0.782098</td>
<td>-1.069094</td>
<td>-1.099248</td>
</tr>
<tr>
<td>6</td>
<td>0.255269</td>
<td>0.009750</td>
<td>0.661084</td>
<td>0.379319</td>
</tr>
<tr>
<td>7</td>
<td>-0.008434</td>
<td>1.952541</td>
<td>-1.056652</td>
<td>0.533946</td>
</tr>
</tbody>
</table>
```

[8 rows x 4 columns]
In [49]: s = df.xs(3)

In [50]: df.append(s, ignore_index=True)
Out[50]:
   A   B   C    D
0 -1.137707 -0.891060 -0.693921  1.613616
1  0.464000  0.227371 -0.496922  0.306389
2 -2.290613 -1.134623 -1.561819 -0.260838
3  0.281957  1.523962 -0.902937  0.068159
4 -0.057873 -0.368204 -1.144073  0.861209
5  0.800193  0.782098 -1.069094 -1.099248
6  0.255269  0.009750  0.661084 -0.379319
7 -0.008434  1.952541 -1.056652  0.533946
8  0.281957  1.523962 -0.902937  0.068159

[9 rows x 4 columns]

You should use ignore_index with this method to instruct DataFrame to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed DataFrame and append or concatenate those objects.

You can also pass a list of dicts or Series:

In [51]: df = DataFrame(np.random.randn(5, 4),
                  columns=['foo', 'bar', 'baz', 'qux'])

In [52]: dicts = [{'foo': 1, 'bar': 2, 'baz': 3, 'peekaboo': 4},
             {'foo': 5, 'bar': 6, 'baz': 7, 'peekaboo': 8}]

In [53]: result = df.append(dicts, ignore_index=True)

In [54]: result
Out[54]:
   bar  baz  foo  peekaboo  qux
0 0.040403 -0.507516 -1.226970   NaN  -0.230096
1-1.934370 -1.652499  0.394500   NaN   1.488753
2 0.576897  1.146000 -0.896484   NaN   1.487349
3 2.121453  0.597701  0.604603   NaN   0.563700
4-1.057909  1.375020  0.967861   NaN  -0.928797
5 2.000000  3.000000  1.000000    4   NaN
6 6.000000  7.000000  5.000000    8   NaN

[7 rows x 5 columns]

### 14.2 Database-style DataFrame joining/merging

pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like base::merge.data.frame in R). The reason for this is careful algorithmic design and internal layout of the data in DataFrame.

See the cookbook for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a comparison with SQL.
pandas provides a single function, `merge`, as the entry point for all standard database join operations between DataFrame objects:

```python
merge(left, right, how='left', on=None, left_on=None, right_on=None,
     left_index=False, right_index=False, sort=True,
     suffixes=('_x', '_y'), copy=True)
```

Here’s a description of what each argument is for:

- **left**: A DataFrame object
- **right**: Another DataFrame object
- **on**: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the DataFrames will be inferred to be the join keys
- **left_on**: Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- **right_on**: Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- **left_index**: If `True`, use the index (row labels) from the left DataFrame as its join key(s). In the case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame
- **right_index**: Same usage as `left_index` for the right DataFrame
- **how**: One of 'left', 'right', 'outer', 'inner'. Defaults to 'inner'. See below for more detailed description of each method
- **sort**: Sort the result DataFrame by the join keys in lexicographical order. Defaults to `True`, setting to `False` will improve performance substantially in many cases
- **suffixes**: A tuple of string suffixes to apply to overlapping columns. Defaults to ('_x', '_y')
- **copy**: Always copy data (default `True`) from the passed DataFrame objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.

`merge` is a function in the pandas namespace, and it is also available as a DataFrame instance method, with the calling DataFrame being implicitly considered the left object in the join.

The related `DataFrame.join` method, uses `merge` internally for the index-on-index and index-on-column(s) joins, but `joins on indexes` by default rather than trying to join on common columns (the default behavior for `merge`). If you are joining on index, you may wish to use `DataFrame.join` to save yourself some typing.

### 14.2.1 Brief primer on merge methods (relational algebra)

Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (DataFrame objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two DataFrame objects on their indexes (which must contain unique values)
- **many-to-one** joins: for example when joining an index (unique) to one or more columns in a DataFrame
- **many-to-many** joins: joining columns on columns.
Note: When joining columns on columns (potentially a many-to-many join), any indexes on the passed DataFrame objects **will be discarded**.

It is worth spending some time understanding the result of the **many-to-many** join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the **Cartesian product** of the associated data. Here is a very basic example with one unique key combination:

```
In [55]: left = DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [56]: right = DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [57]: left
Out[57]:
    key lval
   --- ---
   0 foo  1
   1 foo  2

[2 rows x 2 columns]

In [58]: right
Out[58]:
    key rval
   --- ---
   0 foo  4
   1 foo  5

[2 rows x 2 columns]

In [59]: merge(left, right, on='key')
Out[59]:
    key lval rval
   --- --- ---
   0 foo  1  4
   1 foo  1  5
   2 foo  2  4
   3 foo  2  5

[4 rows x 3 columns]
```

Here is a more complicated example with multiple join keys:

```
In [60]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
                     'key2': ['one', 'two', 'one'],
                     'lval': [1, 2, 3]})

In [61]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                       'key2': ['one', 'one', 'one', 'two'],
                       'rval': [4, 5, 6, 7]})

In [62]: merge(left, right, how='outer')
Out[62]:
    key1 key2 lval rval
   --- --- --- ---
   0 foo one  1  4
   1 foo one  1  5
   2 foo two  2 NaN
   3 bar one  3  6
   4 bar two NaN  7
```

Chapter 14. Merge, join, and concatenate
In [63]: merge(left, right, how='inner')
Out[63]:
   key1 key2  lval  rval
0   foo   one    1    4
1   foo   one    1    5
2   bar   one    3    6

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>

14.2.2 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 [64]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [65]: df1 = df.ix[1:, ['A', 'B']]

In [66]: df2 = df.ix[:5, ['C', 'D']]

In [67]: df1
Out[67]:
   A     B
0 -2.461467 -1.553902
1  1.771740 -0.670027
2 -3.201750  0.792716
3 -0.747169 -0.309038
4  0.936527  1.255746
5  0.062297 -0.110388
6  0.077849  0.629498

In [68]: df2
Out[68]:
   C     D
0  0.377953  0.493672
1  2.015523 -1.833722
2  0.049307  0.521493
3  0.936527  1.255746
4 -0.747169  1.255746
5 -2.655452  1.255746

14.2. Database-style DataFrame joining/merging
In [69]: df1.join(df2)
Out[69]:
   A     B      C     D
0 -2.461467 -1.553902  2.015523 -1.833722
1  1.771740  -0.670027  0.049307  -0.521493
2 -3.201750   0.792716  0.146111   1.903247
3 -0.747169  -0.309038  0.393876   1.861468
4  0.936527   1.255746 -2.655452   1.219492
5  0.062297  -0.110388  NaN          NaN
6  0.077849   0.629498  NaN          NaN
7  [7 rows x 4 columns]

In [70]: df1.join(df2, how='outer')
Out[70]:
   A     B      C     D
0  NaN      NaN  0.377953  0.493672
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740  -0.670027  0.049307  -0.521493
3 -3.201750   0.792716  0.146111   1.903247
4 -0.747169  -0.309038  0.393876   1.861468
5  0.936527   1.255746 -2.655452   1.219492
6  0.062297  -0.110388  NaN          NaN
7  0.077849   0.629498  NaN          NaN
8 [8 rows x 4 columns]

In [71]: df1.join(df2, how='inner')
Out[71]:
   A     B      C     D
0  NaN      NaN   0.377953  0.493672
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740  -0.670027  0.049307  -0.521493
3 -3.201750   0.792716  0.146111   1.903247
4 -0.747169  -0.309038  0.393876   1.861468
5  0.936527   1.255746 -2.655452   1.219492
6  0.062297  -0.110388  NaN          NaN
7  0.077849   0.629498  NaN          NaN
5 [5 rows x 4 columns]

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 [72]: merge(df1, df2, left_index=True, right_index=True, how='outer')
Out[72]:
   A     B      C     D
0  NaN      NaN   0.377953  0.493672
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740  -0.670027  0.049307  -0.521493
3 -3.201750   0.792716  0.146111   1.903247
4 -0.747169  -0.309038  0.393876   1.861468
5  0.936527   1.255746 -2.655452   1.219492
6  0.062297  -0.110388  NaN          NaN
7  0.077849   0.629498  NaN          NaN
8 [8 rows x 4 columns]
14.2.3 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:

```python
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:

```python
In [73]: df['key'] = ['foo', 'bar'] * 4

In [74]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
                           columns=['j1', 'j2'])

In [75]: df
Out[75]:
   A    B    C    D  key
0 -0.308853 -0.681087 0.377953 0.493672  foo
1 -2.461467 -1.553902 2.015523 -1.833722  bar
2  1.771740 -0.670027 0.049307 -0.521493  foo
3 -3.201750  0.792716 0.146111  1.903247  bar
4 -0.747169 -0.309038 0.393876  1.861468  foo
5  0.936527  1.255746 -2.655452  1.219492  bar
6  0.062297 -0.110388 -1.184357 -0.558081  foo
7  0.077849  0.629498 -1.035260 -0.438229  bar

[8 rows x 5 columns]

In [76]: to_join
Out[76]:
   j1    j2
bar 0.503703  0.413086
foo -1.139050  0.660342

[2 rows x 2 columns]

In [77]: df.join(to_join, on='key')
Out[77]:
   A    B    C    D  key  j1    j2
0 -0.308853 -0.681087 0.377953 0.493672  foo -1.139050  0.660342
1 -2.461467 -1.553902 2.015523 -1.833722  bar  0.503703  0.413086
2  1.771740 -0.670027 0.049307 -0.521493  foo -1.139050  0.660342
3 -3.201750  0.792716 0.146111  1.903247  bar  0.503703  0.413086
4 -0.747169 -0.309038 0.393876  1.861468  foo -1.139050  0.660342
5  0.936527  1.255746 -2.655452  1.219492  bar  0.503703  0.413086
6  0.062297 -0.110388 -1.184357 -0.558081  foo -1.139050  0.660342
7  0.077849  0.629498 -1.035260 -0.438229  bar  0.503703  0.413086

[8 rows x 7 columns]

In [78]: merge(df, to_join, left_on='key', right_index=True,
                      how='left', sort=False)
       ....:
Out[78]:
```
To join on multiple keys, the passed DataFrame must have a MultiIndex:

```python
In [79]:
index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
                      ['one', 'two', 'three']],
                   labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
                           [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
                   names=['first', 'second'])

In [80]:
to_join = DataFrame(np.random.randn(10, 3), index=index,
                     columns=['j_one', 'j_two', 'j_three'])

# a little relevant example with NAs
In [81]:
key1 = ['bar', 'bar', 'bar', 'foo', 'foo', 'baz', 'baz', 'qux',
        'qux', 'snap']

In [82]:
key2 = ['two', 'one', 'three', 'one', 'two', 'one', 'two', 'two',
        'three', 'one']

In [83]:
data = np.random.randn(len(key1))

In [84]:
data = DataFrame({'key1' : key1, 'key2' : key2,
                   'data' : data})

In [85]:
data
Out[85]:
    data
0   -1.004168   bar    two
1   -1.377627   bar     one
2   0.499281   bar    three
3   1.405256    foo    one
4   0.162565    foo    two
5   -0.067785   baz    one
6   1.260066   baz    two
7   -1.132896   qux    two
8   -2.064811   qux    three
9   0.301016    snap    one

[10 rows x 3 columns]

In [86]:
to_join
Out[86]:
   j_one  j_two  j_three
Now this can be joined by passing the two key column names:

```
In [87]: data.join(to_join, on=['key1', 'key2'])
Out[87]:
          data  key1  key2   j_one   j_two   j_three
0   -1.004168  bar  two  1.397431  1.503874  -0.478905
1    -1.377627  bar  one  1.515707  -0.276487  -0.223762
2     0.499281  bar  three   NaN   NaN   NaN
3    -1.405256   foo  one  0.464794  -0.309337  -0.649593
4     0.162565   foo  two  0.683758  -0.643834   0.421287
5    -0.067785   baz  one   NaN   NaN   NaN
6    -1.260006   baz  two  -0.135950  -0.730327  -0.033277
7    -1.132896   qux  two  -1.537770   0.555759  -2.277282
8    -2.006481   qux  three  -0.390201  1.207122   0.178690
9     0.301016   snap  one   NaN   NaN   NaN

[10 rows x 6 columns]
```

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 [88]: data.join(to_join, on=['key1', 'key2'], how='inner')
Out[88]:
          data  key1  key2   j_one   j_two   j_three
0   -1.004168  bar  two  1.397431  1.503874  -0.478905
1    -1.377627  bar  one  1.515707  -0.276487  -0.223762
3    -1.405256   foo  one  0.464794  -0.309337  -0.649593
4     0.162565   foo  two  0.683758  -0.643834   0.421287
6    -1.260006   baz  one   NaN   NaN   NaN
7    -1.132896   qux  two  -1.537770   0.555759  -2.277282
8    -2.006481   qux  three  -0.390201  1.207122   0.178690

[7 rows x 6 columns]
```

As you can see, this drops any rows where there was no match.

### 14.2.4 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 [89]: left = DataFrame({'key': ['foo', 'foo'], 'value': [1, 2]})

In [90]: right = DataFrame({'key': ['foo', 'foo'], 'value': [4, 5]})

In [91]: merge(left, right, on='key', suffixes=['_left', '_right'])
Out[91]:
   key  value_left  value_right
0   foo          1          4
1   foo          1          5
2   foo          2          4
3   foo          2          5

[4 rows x 3 columns]

DataFrame.join has lsuffix and rsuffix arguments which behave similarly.

14.2.5 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 [92]: A
Out[92]:
   group  key  lvalue
0     a    a     1
1     a    c     2
2     a    e     3
3     b    a     1
4     b    c     2
5     b    e     3

[6 rows x 3 columns]

In [93]: B
Out[93]:
   key  rvalue
0   b     1
1   c     2
2   d     3

[3 rows x 2 columns]

In [94]: ordered_merge(A, B, fill_method='ffill', left_by='group')
Out[94]:
   group  key  lvalue  rvalue
0     a    a     1     NaN
1     a    b     1     1
2     a    c     2     2
3     a    d     2     3
4     a    e     3     3
5     b    a     1     NaN
6     b    b     1     1
7     b    c     2     2
8     b    d     2     3
9     b    e     3     3

[10 rows x 4 columns]
14.2.6 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 [95]: df1 = df.ix[:, ['A', 'B']]  
In [96]: df2 = df.ix[:, ['C', 'D']]  
In [97]: df3 = df.ix[:, ['key']]  
In [98]: df1  
Out[98]:    A   B  
0 -0.308853 -0.681087  
1 -2.461467 -1.553902  
2  1.771740 -0.670027  
3 -3.201750  0.792716  
4  0.936527  1.255746  
5  0.062297 -0.110388  
6  0.077849  0.629498  
[8 rows x 2 columns]  
In [99]: df1.join([df2, df3])  
Out[99]:    A    B    C    D key  
0 -0.308853 -0.681087  0.377953  0.493672 foo  
1 -2.461467 -1.553902  2.015523 -1.833722 bar  
2  1.771740 -0.670027  0.049307 -0.521493 foo  
3 -3.201750  0.792716  0.146111  1.903247 bar  
4  0.936527  1.255746 -2.655452  1.219492 bar  
5  0.062297 -0.110388 -1.184357 -0.558081 foo  
6  0.077849  0.629498 -1.035260 -0.438229 bar  
[8 rows x 5 columns]

14.2.7 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:

In [100]: df1 = DataFrame([[nan, 3., 5.], [-4.6, np.nan, nan],  
......:       [nan, 7., nan]])  
......:  
In [101]: df2 = DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4.],  
......:               index=[1, 2]])  
......:  
For this, use the combine_first method:

In [102]: df1.combine_first(df2)  
Out[102]:  
0   1   2  
NaN  1.2  
0  NaN  3  5.0

14.2. Database-style DataFrame joining/merging
1 -4.6 NaN -8.2
2 -5.0 7 4.0

[3 rows x 3 columns]

Note that this method only takes values from the right DataFrame if they are missing in the left DataFrame. A related method, `update`, alters non-NA values in-place:

In [103]: df1.update(df2)

In [104]: df1
Out[104]:
   0    1    2
0  NaN  3.0  5.0
1 -42.6 NaN -8.2
2  -5.0  1.6  4.0

[3 rows x 3 columns]
RESHAPING AND PIVOT TABLES

15.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

In [1]: df
Out[1]:

<table>
<thead>
<tr>
<th>date</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>A</td>
<td>0.469112</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>A</td>
<td>-0.282863</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>A</td>
<td>-1.509059</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>B</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>B</td>
<td>1.212112</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>B</td>
<td>-0.173215</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>C</td>
<td>0.119209</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>C</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>C</td>
<td>-0.861849</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>D</td>
<td>-2.104569</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>D</td>
<td>-0.494929</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>D</td>
<td>1.071804</td>
</tr>
</tbody>
</table>

[12 rows x 3 columns]

For the curious here is how the above DataFrame was created:

```python
import pandas.util.testing as tm; tm.N = 3

def unpivot(frame):
    N, K = frame.shape
    data = {'value' : frame.values.ravel('F'),
            'variable' : np.asarray(frame.columns).repeat(N),
            'date' : np.tile(np.asarray(frame.index), K)}
    return DataFrame(data, columns=['date', 'variable', 'value'])

df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable A we could do:

In [2]: df[‘variable’] == ‘A’
Out[2]:

<table>
<thead>
<tr>
<th>date</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>A</td>
<td>0.469112</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>A</td>
<td>-0.282863</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>A</td>
<td>-1.509059</td>
</tr>
</tbody>
</table>

[3 rows x 3 columns]
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:

```python
In [3]: df.pivot(index='date', columns='variable', values='value')
```

```
Out[3]:
variable       A    B    C    D
date
2000-01-03  0.469112 -1.135632 0.119209 -2.104569
2000-01-04  -0.282863  1.212112 -1.044236 -0.494929
2000-01-05  -1.509059 -0.173215 -0.861849  1.071804
```

[3 rows x 4 columns]

If the values argument is omitted, and the input DataFrame has more than one column of values which are not used as column or index inputs to pivot, then the resulting “pivoted” DataFrame will have hierarchical columns whose topmost level indicates the respective value column:

```python
In [4]: df['value2'] = df['value'] * 2
In [5]: pivoted = df.pivot('date', 'variable')
In [6]: pivoted
```

```
Out[6]:
variable       A    B    C    D    A    B
value
date
2000-01-03  0.469112 -1.135632 0.119209 -2.104569 0.938225 -2.271265
2000-01-04  -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224
2000-01-05  -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429
```

[3 rows x 8 columns]

You of course can then select subsets from the pivoted DataFrame:

```python
In [7]: pivoted['value2']
```

```
Out[7]:
variable       A    B    C    D
value
date
2000-01-03  0.938225 -2.271265 0.238417 -4.209138
2000-01-04  -0.565727  2.424224 -2.088472 -0.989859
2000-01-05  -3.018117 -0.346429 -1.723698  2.143608
```

[3 rows x 4 columns]

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.
15.2 Reshaping by stacking and unstacking

Closely related to the `pivot` function are the related `stack` and `unstack` functions currently available on Series and DataFrame. These functions are designed to work together with `MultiIndex` objects (see the section on `hierarchical indexing`). Here are essentially what these functions do:

- **stack**: “pivot” a level of the (possibly hierarchical) column labels, returning a DataFrame with an index with a new inner-most level of row labels.

- **unstack**: inverse operation from `stack`: “pivot” a level of the (possibly hierarchical) row index to the column axis, producing a reshaped DataFrame with a new inner-most level of column labels.

The clearest way to explain is by example. Let’s take a prior example data set from the hierarchical indexing section:

```python
In [8]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                        'foo', 'foo', 'qux', 'qux'],
                        ['one', 'two', 'one', 'two',
                        'one', 'two', 'one', 'two']]))

In [9]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [10]: df = DataFrame(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
[4 rows x 2 columns]
```

The `stack` function “compresses” a level in the DataFrame’s columns to produce either:

- A Series, in the case of a simple column Index

- A DataFrame, in the case of a `MultiIndex` in the columns

If the columns have a `MultiIndex`, you can choose which level to stack. The stacked level becomes the new lowest level in a `MultiIndex` on the columns:

```python
In [13]: stacked = df2.stack()

In [14]: stacked
Out[14]:
   first  second  A       B
bar one  bar      0.721555 -0.706771
   baz one  -0.424972  0.567020
   two bar      -1.039575  0.271860
   baz two  0.276232 -1.087401
dtype: float64
```

15.2. Reshaping by stacking and unstacking
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
[4 rows x 2 columns]
```

```
In [16]: stacked.unstack(1)
Out[16]:
         second one      two
    first
bar   A   0.721555 -1.039575
t     B  -0.706771  0.271860
baz   A  -0.424972  0.276232
t     B   0.567020 -1.087401
[4 rows x 2 columns]
```

```
In [17]: stacked.unstack(0)
Out[17]:
          first  bar       baz
   second
one   A   0.721555 -0.424972
     B  -0.706771  0.567020
two   A  -1.039575  0.276232
     B   0.271860 -1.087401
[4 rows x 2 columns]
```

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
t     B  -0.706771  0.271860
baz   A  -0.424972  0.276232
t     B   0.567020 -1.087401
[4 rows x 2 columns]
```

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.

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 [19]: columns = MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
                                      ('B', 'cat'), ('A', 'dog')],
                                      names=['exp', 'animal'])
```
In [20]: df = DataFrame(randn(8, 4), index=index, columns=columns)

In [21]: df2 = df.ix[[0, 1, 2, 4, 5, 7]]

In [22]: df2
Out[22]:
exp  A   B  A
animal cat  dog  cat  dog
first  second
bar   one  -0.370647 -1.157892 -1.344312 0.844885
      two   1.075770 -0.109050 1.643563 -1.469388
baz   one   0.357021 -0.674600 -1.776904 -0.968914
      two   0.895717  0.805244 -1.206412  2.565646
foo   one  -0.013960 -0.362543 -0.006154 -0.923061
      two   0.895717  0.805244 -1.206412  2.565646
qux   two   0.410835  0.813850  0.132003 -0.827317

[6 rows x 4 columns]

As mentioned above, stack can be called with a level argument to select which level in the columns to stack:

In [23]: df2.stack('exp')
Out[23]:
exp  A   B
animal cat  dog
first  second
bar   one  A -0.370647  0.844885
        B -1.344312 -1.157892
      two  A  1.075770 -1.469388
        B  1.643563 -0.109050
baz   one  A  0.357021 -0.968914
        B -1.776904 -0.674600
      two  A  0.895717  2.565646
        B -1.206412  0.805244
foo   one  A -0.013960 -0.923061
        B -0.006154 -0.362543
      two  A  0.895717  2.565646
        B -1.206412  0.805244
qux   two  A  0.410835 -0.827317
        B  0.132003  0.813850

[12 rows x 2 columns]

In [24]: df2.stack('animal')
Out[24]:
exp  A   B
animal cat  dog
first  second
bar   one  A -0.370647 -1.344312
        dog  0.844885 -1.157892
      two  A  1.075770  1.643563
        dog -1.469388 -0.109050
baz   one  A  0.357021 -1.776904
        dog -0.968914 -0.674600
      two  A  0.895717  2.565646
        dog -1.206412  0.805244
foo   one  A -0.013960 -0.006154
        dog -0.923061 -0.362543
      two  A  0.895717  2.565646
        dog -1.206412  0.805244
qux   two  A  0.410835  0.132003
        dog -0.827317  0.813850

[12 rows x 2 columns]
Unstacking when the columns are a MultiIndex is also careful about doing the right thing:

```
In [25]: df[:3].unstack(0)
Out[25]:
exp  A  B  A
    animal  cat  dog  cat  dog
    first   bar  baz  bar  baz  bar
    second
one  -0.370647  0.357021 -1.157892 -0.6746 -1.344312 -1.776904  0.844885
two  1.075770   NaN  -0.109050   NaN  1.643563   NaN   -1.469388

exp
animal
first  baz
second
one   -0.968914
two    NaN

[2 rows x 8 columns]
```

```
In [26]: df2.unstack(1)
Out[26]:
exp  A  B  A
    animal  cat  dog  cat  dog
    second  one  two  one  two  one  two  one
    first
    bar  -0.370647  1.075770 -0.109050 -1.344312  1.643563  0.844885
    baz  0.357021   NaN  -0.674600   NaN  -1.776904   NaN  -0.968914
    foo -0.013960  0.895717 -0.362543  0.805244 -0.006154 -1.206412 -0.923061
    qux  NaN      0.410835  NaN      0.813850  NaN      0.132003  NaN

exp
animal
second  two
first
bar   -1.469388
baz     NaN
foo  2.565646
qux -0.827317

[4 rows x 8 columns]
```

15.3 Reshaping by Melt

The `melt` function found in pandas.core.reshape 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 “pivoted” 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 [28]: cheese
Out[28]:
   first    last  weight
0   John    Doe    130
1  Mary     Bo     150

[2 rows x 4 columns]

In [29]: melt(cheese, id_vars=['first', 'last'])
Out[29]:
   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

[4 rows x 4 columns]

In [30]: melt(cheese, id_vars=['first', 'last'], var_name='quantity')
Out[30]:
   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

[4 rows x 4 columns]

Another way to transform is to use the `wide_to_long` panel data convenience function.

In [31]: 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 : 0.7},
                           'B1980' : {0 : 3.2, 1 : 1.3, 2 : 0.1},
                           'X' : dict(zip(range(3), np.random.randn(3)))})

In [32]: dft['id'] = dft.index

In [33]: dft
Out[33]:
0      a      d     2.5     3.2 -0.076467   0
1      b      e     1.2     1.3 -1.187678   1
2      c      f     0.7     0.1  1.130127   2

[3 rows x 6 columns]

In [34]: pd.wide_to_long(dft, ['A', 'B'], i='id', j='year')
Out[34]:
      X  A  B
id year
0  1970 -0.076467  a  2.5
1  1970 -1.187678  b  1.2
2  1970  1.130127  c  0.7
0  1980 -0.076467  d  3.2
1  1980 -1.187678  e  1.3
2  1980  1.130127  f  0.1

15.3. Reshaping by Melt

367
15.4 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.

```python
In [35]: df
Out[35]:
exp A B A
animal cat dog cat dog
first second
bar one -0.370647 -1.157892 -1.344312 0.844885
two 1.075770 -0.109050 1.643563 -1.469388
baz one 0.357021 -0.674600 -1.776904 -0.968914
two -1.294524 0.413738 0.276662 -0.472035
foo one -0.013960 -0.362543 -0.006154 -0.923061
two 0.895717 0.805244 -1.206412 2.565646
qux one 1.431256 1.340309 -1.170299 -0.226169
two 0.410835 0.813850 0.132003 -0.827317
```

[8 rows x 3 columns]

```python
In [36]: df.stack().mean(1).unstack()
Out[36]:
animal cat dog
first second
bar one -0.857479 -0.156504
two 1.359666 -0.789219
baz one -0.709942 -0.821757
two -0.508931 -0.029148
foo one -0.010057 -0.642802
two -0.155347 1.685445
qux one 0.130479 0.557070
two 0.271419 -0.006733
```

[8 rows x 2 columns]

```python
In [37]: df.groupby(level=1, axis=1).mean()
Out[37]:
animal cat dog
first second
bar one -0.857479 -0.156504
two 1.359666 -0.789219
baz one -0.709942 -0.821757
two -0.508931 -0.029148
foo one -0.010057 -0.642802
two -0.155347 1.685445
qux one 0.130479 0.557070
two 0.271419 -0.006733
```

[8 rows x 2 columns]

```python
# same result, another way
In [38]: df.stack().groupby(level=1).mean()
Out[38]:
animal cat dog
first second
bar one -0.857479 -0.156504
two 1.359666 -0.789219
baz one -0.709942 -0.821757
two -0.508931 -0.029148
foo one -0.010057 -0.642802
two -0.155347 1.685445
qux one 0.130479 0.557070
two 0.271419 -0.006733
```

[8 rows x 2 columns]
In [38]:
exp
   A  B
second
one  0.016301 -0.644049
two  0.110588  0.346200

[2 rows x 2 columns]

In [39]: df.mean().unstack(0)
Out[39]:
exp  A  B
animal
   cat  0.311433 -0.431481
dog  -0.184544  0.133632

[2 rows x 2 columns]

15.5 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
- **rows**: list of columns to group by on the table rows
- **cols**: list of columns to group by on the table columns
- **aggfunc**: function to use for aggregation, defaulting to numpy.mean

Consider a data set like this:

In [40]:
   df = 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))

In [41]: df
Out[41]:
    A  B  C  D     E
 0  one  foo  1.436737 0.149748
 1  one  B    1.413681 -0.732339
 2  two  C    1.607920 0.687738
 3  three A    1.024180 0.176444
 4   one B    0.569605 0.403310
 5   one C    0.875906 -0.154951
 6  two A    2.211372 0.301624
 7  three B    0.974466 -2.179861
 8   one C    2.006747 -1.369849
 9   one A    0.410001 -0.954208
10  two B    -0.078638 1.462696
11  three C    0.545952 -1.743161
We can produce pivot tables from this data very easily:

**In [42]:** `pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])`

**Out[42]:**

```
   C  bar  foo
A
one  A  0.274863 -1.327977
    B -0.079051 -1.320253
    C  0.377300 -0.832506
three A -0.128534 NaN
       B  NaN  0.835120
       C -0.037012 NaN
two   A  NaN -1.154627
       B  NaN  0.835120
       C  NaN  1.188862
```

[9 rows x 2 columns]

**In [43]:** `pivot_table(df, values='D', rows=['B'], cols=['A', 'C'], aggfunc=np.sum)`

**Out[43]:**

```
     A  one  three  two
A  C  bar  foo  bar  foo  bar  foo
  one  0.549725 -2.655954 -0.257067 NaN   NaN   NaN   -2.309255
  B  NaN  NaN  0.835120  1.188974 NaN   NaN   2.377724
  C  0.754600 -1.665013 -0.074024 NaN   NaN   2.377724  2.241830
```

[3 rows x 6 columns]

**In [44]:** `pivot_table(df, values=['D','E'], rows=['B'], cols=['A', 'C'], aggfunc=np.sum)`

**Out[44]:**

```
  D          
  E  A  one  three  two
C  bar  foo  bar  foo  bar  foo
B  A  0.549725 -2.655954 -0.257067 NaN   NaN   NaN   -2.309255 -2.190477
  B  NaN  NaN  0.835120  1.188974 NaN   NaN   1.399070
  C  0.754600 -1.665013 -0.074024 NaN   NaN   2.377724  2.241830
```

```
  A  one  three  two
C  C  foo  bar  foo  bar  foo
B  A  NaN  NaN  0.867024 NaN   NaN   0.316495
  B  NaN  NaN  1.177566  2.358867 NaN   NaN
  C  -1.687290 -2.230762 NaN   NaN   2.001971
```

[3 rows x 12 columns]

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 [45]: pivot_table(df, rows=['A', 'B'], cols=['C'])
Out[45]:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bar</td>
<td>foo</td>
<td>bar</td>
<td>foo</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-0.079051</td>
<td>-1.320253</td>
<td>0.699535</td>
<td>-0.538846</td>
</tr>
<tr>
<td></td>
<td>0.377300</td>
<td>-0.832506</td>
<td>1.120915</td>
<td>-0.843645</td>
</tr>
<tr>
<td></td>
<td>-0.128534</td>
<td>NaN</td>
<td>0.433512</td>
<td>NaN</td>
</tr>
<tr>
<td>three</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.835120</td>
<td>NaN</td>
<td>0.588783</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>-0.037012</td>
<td>NaN</td>
<td>-1.115381</td>
<td>NaN</td>
</tr>
<tr>
<td>two</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-0.594487</td>
<td>NaN</td>
<td>1.179433</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>1.188862</td>
<td>NaN</td>
<td>1.000985</td>
<td>NaN</td>
</tr>
</tbody>
</table>

[9 rows x 4 columns]

You can render a nice output of the table omitting the missing values by calling to_string if you wish:

In [46]: table = pivot_table(df, rows=['A', 'B'], cols=['C'])

In [47]: print(table.to_string(na_rep=''))

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bar</td>
<td>foo</td>
<td>bar</td>
<td>foo</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.274863</td>
<td>-1.327977</td>
<td>-1.095238</td>
<td>-0.338421</td>
</tr>
<tr>
<td></td>
<td>-0.079051</td>
<td>-1.320253</td>
<td>0.699535</td>
<td>-0.538846</td>
</tr>
<tr>
<td></td>
<td>0.377300</td>
<td>-0.832506</td>
<td>1.120915</td>
<td>-0.843645</td>
</tr>
<tr>
<td></td>
<td>-0.128534</td>
<td>NaN</td>
<td>0.433512</td>
<td>NaN</td>
</tr>
<tr>
<td>three</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.835120</td>
<td>NaN</td>
<td>0.588783</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>-0.037012</td>
<td>NaN</td>
<td>-1.115381</td>
<td>NaN</td>
</tr>
<tr>
<td>two</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-0.594487</td>
<td>NaN</td>
<td>1.179433</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>1.188862</td>
<td>NaN</td>
<td>1.000985</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Note that pivot_table is also available as an instance method on DataFrame.

15.5.1 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

- **rows**: array-like, values to group by in the rows
- **cols**: 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 [48]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'
In [49]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)
In [50]: b = np.array([one, one, two, one, two, one], dtype=object)
In [51]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)
In [52]: crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
Out[52]:

   b    one    two
  c   dull   shiny
da
   bar 1 0 0 1
   foo 2 1 1 0
```

### 15.5.2 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:

```python
In [53]: df.pivot_table(rows=['A', 'B'], cols='C', margins=True, aggfunc=np.std)
Out[53]:

<table>
<thead>
<tr>
<th>C</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>All</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>one</td>
<td>0.968543</td>
<td>0.153810</td>
<td>1.084870</td>
<td>0.199447</td>
<td>0.690376</td>
</tr>
<tr>
<td></td>
<td>0.917338</td>
<td>0.132127</td>
<td>0.894343</td>
<td>0.418926</td>
<td>0.273641</td>
</tr>
<tr>
<td></td>
<td>0.705136</td>
<td>1.660627</td>
<td>1.254131</td>
<td>1.804346</td>
<td>0.744165</td>
</tr>
<tr>
<td>three</td>
<td>1.630183</td>
<td>1.630183</td>
<td>0.363548</td>
<td>NaN</td>
<td>0.363548</td>
</tr>
<tr>
<td></td>
<td>0.917338</td>
<td>0.132127</td>
<td>0.894343</td>
<td>0.418926</td>
<td>0.273641</td>
</tr>
<tr>
<td></td>
<td>0.705136</td>
<td>1.660627</td>
<td>1.254131</td>
<td>1.804346</td>
<td>0.744165</td>
</tr>
<tr>
<td></td>
<td>0.824435</td>
<td>0.824435</td>
<td>0.887815</td>
<td>NaN</td>
<td>0.887815</td>
</tr>
<tr>
<td>two</td>
<td>1.494463</td>
<td>1.494463</td>
<td>NaN</td>
<td>0.197065</td>
<td>0.197065</td>
</tr>
<tr>
<td></td>
<td>0.729521</td>
<td>0.729521</td>
<td>0.400594</td>
<td>NaN</td>
<td>0.400594</td>
</tr>
<tr>
<td></td>
<td>0.592638</td>
<td>0.592638</td>
<td>NaN</td>
<td>0.442998</td>
<td>0.442998</td>
</tr>
<tr>
<td></td>
<td>0.816058</td>
<td>1.294620</td>
<td>1.055572</td>
<td>1.190502</td>
<td>1.403041</td>
</tr>
<tr>
<td>All</td>
<td>0.602542</td>
<td>0.602542</td>
<td>0.602542</td>
<td>0.602542</td>
<td>0.602542</td>
</tr>
</tbody>
</table>
```

### 15.6 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:

```python
In [54]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])
In [55]: cut(ages, bins=3)
Out[55]:

(9.95, 26.667]
(9.95, 26.667]
(9.95, 26.667]
```
If the `bins` keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [56]: cut(ages, bins=[0, 18, 35, 70])
Out[56]:
   (0, 18)  (18, 35)  (35, 70)
0          0         0
1          0         0
2          0         0
3          0         0
4          0         0
5          0         0

Levels (3): Index([’(0, 18)’, ’(18, 35)’, ’(35, 70)’], dtype=object)
```

### 15.7 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 [57]: df = DataFrame(’key’: list(’bbacab’), ’data’: range(6))
In [58]: get_dummies(df[’key’])
Out[58]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0

[6 rows x 3 columns]
```

Sometimes it’s useful to prefix the column names, for example when merging the result with the original DataFrame:

```
In [59]: dummies = get_dummies(df[’key’], prefix=’key’)  
In [60]: dummies
Out[60]:
   key_a  key_b  key_c
0  0  1  0
1  1  0  0
2  1  0  0
3  0  1  0
4  1  0  0
5  0  1  0

[6 rows x 3 columns]
```
In [61]: df[['data1']].join(dummies)
Out[61]:
   data1  key_a  key_b  key_c
0      0       0       1       0
1      1       0       1       0
2      2       1       0       0
3      3       0       0       1
4      4       1       0       0
5      5       0       1       0

[6 rows x 4 columns]

This function is often used along with discretization functions like `cut`:

In [62]: values = randn(10)
In [63]: values
Out[63]:
array([-0.0822, -2.1829, 0.3804, 0.0848, 0.4324, 1.52 , -0.4937,
       0.6002, 0.2742, 0.1329])
In [64]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
In [65]: get_dummies(cut(values, bins))
Out[65]:
   (0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]
0         0         0         0         0
1         0         0         0         0
2         0         0         1         0
3         0         0         0         0
4         0         0         1         0
5         0         0         0         0
6         0         0         0         0
7         0         0         0         1
8         0         0         1         0
9         1         0         0         0

[10 rows x 4 columns]

See also `get_dummies()`.

15.8 Factorizing values

To encode 1-d values as an enumerated type use `factorize`:

In [66]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])
In [67]: x
Out[67]:
0    A
1    A
2  NaN
3    B
4  3.14
5  inf
dtype: object
In [68]: labels, uniques = pd.factorize(x)

In [69]: labels
Out[69]: array([ 0,  0, -1,  1,  2,  3])

In [70]: uniques
Out[70]: array(['A', 'B', 3.14, inf], dtype=object)

Note that `factorize` is similar to `numpy.unique`, but differs in its handling of NaN:

In [71]: pd.factorize(x, sort=True)
Out[71]: (array([2, 2, -1, 3, 0, 1]), array([3.14, inf, 'A', 'B'], dtype=object))

In [72]: np.unique(x, return_inverse=True)[::-1]
Out[72]: (array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 04:00:00]
Length: 5, Freq: H, Timezone: None
```

Index pandas objects with dates:

```python
In [3]: ts = Series(randn(len(rng)), index=rng)
In [4]: ts.head()
Out[4]:
2011-01-01 00:00:00  0.469112
2011-01-01 01:00:00  -0.282863
2011-01-01 02:00:00  -1.509059
2011-01-01 03:00:00  -1.135632
2011-01-01 04:00:00   1.212112
Freq: H, dtype: float64
```

Change frequency and fill gaps:

```python
# to 45 minute frequency and forward fill
In [5]: converted = ts.asfreq('45Min', method='pad')
```
In [6]: converted.head()
Out[6]:
2011-01-01 00:00:00  0.469112
2011-01-01 00:45:00  0.469112
2011-01-01 01:30:00 -0.282863
2011-01-01 02:15:00 -1.509059
2011-01-01 03:00:00 -1.135632
Freq: 45T, dtype: float64

Resample:

# Daily means
In [7]: ts.resample('D', how='mean')
Out[7]:
2011-01-01 -0.319569
2011-01-02 -0.337703
2011-01-03  0.117258
Freq: D, dtype: float64

16.1 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

378  Chapter 16. Time Series / Date functionality
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 forth-coming in future releases.

## 16.2 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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2005-11-23, 2010-12-31]
Length: 2, Freq: None, Timezone: 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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-04 10:00:00]
Length: 1, Freq: None, Timezone: None
```

```python
In [19]: to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
Out[19]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-14, 2012-01-14]
Length: 2, Freq: None, Timezone: 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 then 0.13.0 explicitly specifying a format string of `%Y%m%d` takes a faster path still.

### 16.2.1 Invalid Data

Pass `coerce=True` to convert invalid data to `NaT` (not a time):

```python
In [20]: to_datetime(['2009-07-31', 'asd'])
Out[20]: array(['2009-07-31', 'asd'], dtype=object)
```

```python
In [21]: to_datetime(['2009-07-31', 'asd'], coerce=True)
```
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)
```

### 16.2.2 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]: <class 'pandas.tseries.index.DatetimeIndex'>
[2012-10-08 18:15:05, ..., 2012-10-12 18:15:05]
Length: 5, Freq: None, Timezone: None
```

```python
In [24]: to_datetime([1349720105100, 1349720105200, 1349720105300, ...
           : 1349720105400, 1349720105500], unit='ms')
Out[24]: <class 'pandas.tseries.index.DatetimeIndex'>
[2012-10-08 18:15:05.100000, ..., 2012-10-08 18:15:05.500000]
Length: 5, Freq: None, Timezone: None
```

These work, but the results may be unexpected.

```python
In [25]: to_datetime([1])
Out[25]: <class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:00.000000001]
Length: 1, Freq: None, Timezone: None
```

```python
In [26]: to_datetime([1, 3.14], unit='s')
Out[26]: <class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:01, 1970-01-01 00:00:03.140000]
Length: 2, Freq: None, Timezone: None
```

**Note:** Epoch times will be rounded to the nearest nanosecond.

### 16.3 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:
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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01, ..., 2012-05-03]
Length: 3, Freq: None, Timezone: None

In [30]: index = Index(dates)

In [31]: index # Automatically converted to DatetimeIndex
Out[31]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01, ..., 2012-05-03]
Length: 3, Freq: None, Timezone: None

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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31, ..., 2083-04-30]
Length: 1000, Freq: M, Timezone: None

In [34]: index = bdate_range('2012-1-1', periods=250)

In [35]: index
Out[35]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-02, ..., 2012-12-14]
Length: 250, Freq: B, Timezone: 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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01, ..., 2012-01-01]
Length: 366, Freq: D, Timezone: None

In [40]: rng = bdate_range(start, end)

In [41]: rng
Out[41]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03, ..., 2011-12-30]
The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

### 16.4 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

DatetimeIndex 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.
DatetimeIndex can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

In [46]: rng = date_range(start, end, freq='BM')

In [47]: ts = Series(randn(len(rng)), index=rng)

In [48]: ts.index
Out[48]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31, ..., 2011-12-30]
Length: 12, Freq: BM, Timezone: None

In [49]: ts[:5].index
Out[49]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31, ..., 2011-05-31]
Length: 5, Freq: BM, Timezone: None

In [50]: ts[::2].index
Out[50]:
<class 'pandas.tseries.index.DatetimeIndex'>
Length: 6, Freq: 2BM, Timezone: None

### 16.4.1 Partial String Indexing

You can pass in dates and strings that parse to dates as indexing parameters:

In [51]: ts['1/31/2011']
Out[51]:
2011-01-31  -1.281247
Freq: BM, dtype: float64

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
2011-11-30  -0.732339
2011-12-30  0.687738
Freq: BM, dtype: float64

To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

In [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
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
2013-01-01 00:07:00 1.462696
2013-01-01 00:08:00 -1.743161
2013-01-01 00:09:00 -0.826591
2013-01-01 00:10:00 -0.345352
2013-01-01 00:11:00 1.314232
2013-01-01 00:12:00 0.690579
2013-01-01 00:13:00 0.995761
2013-01-01 00:14:00 2.396780
...
[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
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
2013-01-01 00:07:00 1.462696
2013-01-01 00:08:00 -1.743161
2013-01-01 00:09:00 -0.826591
2013-01-01 00:10:00 -0.345352
2013-01-01 00:11:00 1.314232
2013-01-01 00:12:00 0.690579
2013-01-01 00:13:00 0.995761
2013-01-01 00:14:00 2.396780
...
[100000 rows x 1 columns]
This starts on the very first time in the month, and includes the last date & time for the month

```
In [59]: dft['2013-1':'2013-2']
Out[59]:
   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
2013-01-01  00:07:00    1.462696
2013-01-01  00:08:00   -1.743161
2013-01-01  00:09:00   -0.826591
2013-01-01  00:10:00   -0.345352
2013-01-01  00:11:00    1.314232
2013-01-01  00:12:00    0.690579
2013-01-01  00:13:00    0.995761
2013-01-01  00:14:00    2.396780
...[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]:
   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
2013-01-01  00:07:00    1.462696
2013-01-01  00:08:00   -1.743161
2013-01-01  00:09:00   -0.826591
2013-01-01  00:10:00   -0.345352
2013-01-01  00:11:00    1.314232
2013-01-01  00:12:00    0.690579
2013-01-01  00:13:00    0.995761
2013-01-01  00:14:00    2.396780
...[84960 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
2013-01-01  00:07:00    1.462696
2013-01-01  00:08:00   -1.743161
2013-01-01  00:09:00   -0.826591
2013-01-01  00:10:00   -0.345352
2013-01-01  00:11:00    1.314232
2013-01-01  00:12:00    0.690579
2013-01-01  00:13:00    0.995761
2013-01-01  00:14:00    2.396780
```
2013-01-01 00:07:00  1.462696
2013-01-01 00:08:00 -1.743161
2013-01-01 00:09:00 -0.826591
2013-01-01 00:10:00 -0.345352
2013-01-01 00:11:00  1.314232
2013-01-01 00:12:00  0.690579
2013-01-01 00:13:00  0.995761
2013-01-01 00:14:00  2.396780
...
[83521 rows x 1 columns]

We are stopping on the included end-point as its 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 00:07:00  0.251617
2013-01-15 00:08:00 -0.775668
2013-01-15 00:09:00  0.521086
2013-01-15 00:10:00  2.030114
2013-01-15 00:11:00 -0.250333
2013-01-15 00:12:00 -1.158353
2013-01-15 00:13:00  0.685205
2013-01-15 00:14:00 -0.089428
...
[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
Name: 2013-01-15 12:30:00, dtype: float64

### 16.4.2 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
2013-01-01 00:00:00  0.176444
2013-01-01 00:01:00 -0.154951
2013-01-01 00:02:00  0.301624
2013-01-01 00:03:00 -2.179861
2013-01-01 00:04:00 -1.369849
2013-01-01 00:05:00 -0.954208
2013-01-01 00:06:00  1.462696
2013-01-01 00:07:00 -1.743161
2013-01-01 00:08:00 -0.826591
2013-01-01 00:09:00 -0.345352
2013-01-01 00:10:00  1.314232
2013-01-01 00:11:00  0.690579
2013-01-01 00:12:00  0.995761
2013-01-01 00:13:00  2.396780
... (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
2013-01-01 10:12:00 -0.246733
2013-01-01 10:13:00 -1.429225
2013-01-01 10:14:00 -1.265339
2013-01-01 10:15:00  0.710986
2013-01-01 10:16:00 -0.818200
2013-01-01 10:17:00  0.543542
2013-01-01 10:18:00  1.577713
2013-01-01 10:19:00  0.316630
2013-01-01 10:20:00 -0.773194
2013-01-01 10:21:00 -1.615112
2013-01-01 10:22:00  0.965363
2013-01-01 10:23:00 -0.882845
2013-01-01 10:24:00 -1.861244
2013-01-01 10:25:00 -0.742435
2013-01-01 10:26:00 -1.937111
... (83521 rows x 1 columns)

### 16.4.3 Truncating & Fancy Indexing

A `truncate` convenience function is provided that is equivalent to slicing:

In [66]: ts.truncate(before='10/31/2011', after='12/31/2011')
Out[66]:
   A
2011-10-31  0.149848
2011-11-30 -0.732339
2011-12-30  0.687738
Freq: BM, dtype: float64

16.4. DatetimeIndex
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]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31, ..., 2011-07-29]
Length: 3, Freq: None, Timezone: None
```

### 16.5 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>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>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>
</tbody>
</table>

The basic DateOffset takes the same arguments as dateutil.relativedelta, which works like:

```python
In [68]: d = datetime(2008, 8, 18)
In [69]: d + relativedelta(months=4, days=5)
Out[69]: datetime.datetime(2008, 12, 23, 0, 0)
```

We could have done the same thing with DateOffset:

```python
In [70]: from pandas.tseries.offsets import *
```
In [71]: d + DateOffset(months=4, days=5)
Out[71]: Timestamp('2008-12-23 00:00:00', tz=None)

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 00:00:00', tz=None)

In [73]: d + BMonthEnd()
Out[73]: Timestamp('2008-08-29 00:00:00', tz=None)

The `rollforward` and `rollback` methods do exactly what you would expect:

In [74]: d
Out[74]: datetime.datetime(2008, 8, 18, 0, 0)

In [75]: offset = BMonthEnd()

In [76]: offset.rollforward(d)
Out[76]: Timestamp('2008-08-29 00:00:00', tz=None)

In [77]: offset.rollback(d)
Out[77]: datetime.datetime(2008, 7, 31, 0, 0)

It’s definitely worth exploring the `pandas.tseries.offsets` module and the various docstrings for the classes.

### 16.5.1 Parametric offsets

Some of the offsets can be “parameterized” when created to result in different behavior. For example, the `Week` offset for generating weekly data accepts a `weekday` parameter which results in the generated dates always lying on a particular day of the week:

In [78]: d + Week()
Out[78]: datetime.datetime(2008, 8, 25, 0, 0)

In [79]: d + Week(weekday=4)
Out[79]: Timestamp('2008-08-22 00:00:00', tz=None)

In [80]: (d + Week(weekday=4)).weekday()
Out[80]: 4

Another example is parameterizing `YearEnd` with the specific ending month:
16.5.2 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 [83]: from pandas.tseries.offsets import CustomBusinessDay

# As an interesting example, let’s look at Egypt where
# a Friday-Saturday weekend is observed.
In [84]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers’ Day so let’s
# add that for a couple of years
In [85]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]

In [86]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [87]: dt = datetime(2013, 4, 30)

In [88]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00

In [89]: dts = date_range(dt, periods=5, freq=bday_egypt).to_series()

In [90]: print(dts)
2013-04-30
2013-04-30
2013-05-02
2013-05-02
2013-05-05
2013-05-05
2013-05-06
2013-05-06
2013-05-07
2013-05-07
Freq: C, dtype: datetime64[ns]

In [91]: 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
dtype: object
```

**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.

### 16.5.3 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</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>MS</td>
<td>month start frequency</td>
</tr>
<tr>
<td>BMS</td>
<td>business month start frequency</td>
</tr>
<tr>
<td>Q</td>
<td>quarter end frequency</td>
</tr>
<tr>
<td>BQ</td>
<td>business quarter end frequency</td>
</tr>
<tr>
<td>QS</td>
<td>quarter start frequency</td>
</tr>
<tr>
<td>BQS</td>
<td>business quarter start frequency</td>
</tr>
<tr>
<td>A</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>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>milliseconds</td>
</tr>
<tr>
<td>U</td>
<td>microseconds</td>
</tr>
</tbody>
</table>

### 16.5.4 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```python
In [92]: date_range(start, periods=5, freq='B')
Out[92]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03, ..., 2011-01-07]
Length: 5, Freq: B, Timezone: None
```

```python
In [93]: date_range(start, periods=5, freq=BDay())
Out[93]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03, ..., 2011-01-07]
Length: 5, Freq: B, Timezone: None
```

You can combine together day and intraday offsets:
In [94]: date_range(start, periods=10, freq='2h20min')
Out[94]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 21:00:00]
Length: 10, Freq: 140T, Timezone: None

In [95]: date_range(start, periods=10, freq='1D10U')
Out[95]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-10 00:00:00.000090]
Length: 10, Freq: 86400000010U, Timezone: None

16.5.5 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W-SUN</td>
<td>weekly frequency (sundays). Same as ‘W’</td>
</tr>
<tr>
<td>W-MON</td>
<td>weekly frequency (mondays)</td>
</tr>
<tr>
<td>W-TUE</td>
<td>weekly frequency (tuesdays)</td>
</tr>
<tr>
<td>W-WED</td>
<td>weekly frequency (wednesdays)</td>
</tr>
<tr>
<td>W-THU</td>
<td>weekly frequency (thursdays)</td>
</tr>
<tr>
<td>W-FRI</td>
<td>weekly frequency (fridays)</td>
</tr>
<tr>
<td>W-SAT</td>
<td>weekly frequency (saturdays)</td>
</tr>
<tr>
<td>(B)Q(S)-DEC</td>
<td>quarterly frequency, year ends in December. Same as ‘Q’</td>
</tr>
<tr>
<td>(B)Q(S)-JAN</td>
<td>quarterly frequency, year ends in January</td>
</tr>
<tr>
<td>(B)Q(S)-FEB</td>
<td>quarterly frequency, year ends in February</td>
</tr>
<tr>
<td>(B)Q(S)-MAR</td>
<td>quarterly frequency, year ends in March</td>
</tr>
<tr>
<td>(B)Q(S)-APR</td>
<td>quarterly frequency, year ends in April</td>
</tr>
<tr>
<td>(B)Q(S)-MAY</td>
<td>quarterly frequency, year ends in May</td>
</tr>
<tr>
<td>(B)Q(S)-JUN</td>
<td>quarterly frequency, year ends in June</td>
</tr>
<tr>
<td>(B)Q(S)-JUL</td>
<td>quarterly frequency, year ends in July</td>
</tr>
<tr>
<td>(B)Q(S)-AUG</td>
<td>quarterly frequency, year ends in August</td>
</tr>
<tr>
<td>(B)Q(S)-SEP</td>
<td>quarterly frequency, year ends in September</td>
</tr>
<tr>
<td>(B)Q(S)-OCT</td>
<td>quarterly frequency, year ends in October</td>
</tr>
<tr>
<td>(B)Q(S)-NOV</td>
<td>quarterly frequency, year ends in November</td>
</tr>
<tr>
<td>(B)A(S)-DEC</td>
<td>annual frequency, anchored end of December. Same as ‘A’</td>
</tr>
<tr>
<td>(B)A(S)-JAN</td>
<td>annual frequency, anchored end of January</td>
</tr>
<tr>
<td>(B)A(S)-FEB</td>
<td>annual frequency, anchored end of February</td>
</tr>
<tr>
<td>(B)A(S)-MAR</td>
<td>annual frequency, anchored end of March</td>
</tr>
<tr>
<td>(B)A(S)-APR</td>
<td>annual frequency, anchored end of April</td>
</tr>
<tr>
<td>(B)A(S)-MAY</td>
<td>annual frequency, anchored end of May</td>
</tr>
<tr>
<td>(B)A(S)-JUN</td>
<td>annual frequency, anchored end of June</td>
</tr>
<tr>
<td>(B)A(S)-JUL</td>
<td>annual frequency, anchored end of July</td>
</tr>
<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.
16.5.6 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 quarter 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.

16.6 Time series-related instance methods

16.6.1 Shifting / lagging

One may want to shift or lag the values in a TimeSeries back and forward in time. The method for this is shift, which is available on all of the pandas objects. In DataFrame, shift will currently only shift along the index and in Panel along the major_axis.

```
In [96]: ts = ts[:5]
In [97]: ts.shift(1)
Out[97]:
2011-01-31   NaN
2011-02-28 -1.281247
2011-03-31 -0.727707
```

16.6. Time series-related instance methods
The shift method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object or also a `offset alias`:

```python
In [98]: ts.shift(5, freq=datetools.bday)
Out[98]:
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
```

```python
In [99]: ts.shift(5, freq='BM')
Out[99]:
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 TimeSeries objects also have a `tshift` convenience method that changes all the dates in the index by a specified number of offsets:

```python
In [100]: ts.tshift(5, freq='D')
Out[100]:
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.

### 16.6.2 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`.

```python
In [101]: dr = date_range('1/1/2010', periods=3, freq=3 * datetools.bday)
In [102]: ts = Series(randn(3), index=dr)
In [103]: ts
Out[103]:
2010-01-01  -0.659574
2010-01-06   1.494522
2010-01-11  -0.778425
Freq: 3B, dtype: float64
```

```python
In [104]: ts.asfreq(BDay())
Out[104]:
2010-01-01  -0.659574
```

---

**Chapter 16. Time Series / Date functionality**
asfreq provides a further convenience so you can specify an interpolation method for any gaps that may appear after
the frequency conversion.

```
In [105]: ts.asfreq(BDay(), method='pad')
Out[105]:
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
```

### 16.6.3 Filling forward / backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the *missing data section*.

### 16.6.4 Converting to Python datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.

### 16.7 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 [106]: rng = date_range('1/1/2012', periods=100, freq='S')

In [107]: ts = Series(randint(0, 500, len(rng)), index=rng)

In [108]: ts.resample('5Min', how='sum')
Out[108]:
2012-01-01   25103
Freq: 5T, dtype: int64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency
conversion and resampling operation.

The `how` parameter can be a function name or numpy array function that takes an array and produces aggregated
values:

```
In [109]: ts.resample('5Min', how='sum')
Out[109]:
2012-01-01   25103
Freq: 5T, dtype: int64
```

```
In [110]: ts.resample('5Min', how=np.sum)
Out[110]:
2012-01-01   25103
Freq: 5T, dtype: int64
```
In [109]: ts.resample('5Min') # default is mean
Out[109]:
2012-01-01  251.03
Freq: 5T, dtype: float64

In [110]: ts.resample('5Min', how='ohlc')
Out[110]:
<table>
<thead>
<tr>
<th></th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>308</td>
<td>460</td>
<td>9</td>
<td>205</td>
</tr>
</tbody>
</table>

[1 rows x 4 columns]

In [111]: ts.resample('5Min', how=np.max)
Out[111]:
2012-01-01  460
Freq: 5T, dtype: int64

Any function available via dispatching can be given to the how parameter by name, including sum, mean, std, 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 [112]: ts.resample('5Min', closed='right')
Out[112]:
2011-12-31 23:55:00  308.000000
2012-01-01  00:00:00  250.454545
Freq: 5T, dtype: float64

In [113]: ts.resample('5Min', closed='left')
Out[113]:
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 [114]: ts[:2].resample('250L')
Out[114]:
2012-01-01  00:00:00  308
2012-01-01  00:00:00.250000  NaN
2012-01-01  00:00:00.500000  NaN
2012-01-01  00:00:00.750000  NaN
2012-01-01  00:00:01  204
Freq: 250L, dtype: float64

In [115]: ts[:2].resample('250L', fill_method='pad')
Out[115]:
2012-01-01  00:00:00  308
2012-01-01  00:00:00.250000  308
2012-01-01  00:00:00.500000  308
2012-01-01  00:00:00.750000  308
2012-01-01  00:00:01  204
Freq: 250L, dtype: int64

In [116]: ts[:2].resample('250L', fill_method='pad', limit=2)
Out[116]:
2012-01-01  00:00:00  308
2012-01-01  00:00:00.250000  308
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 [117]: ts.resample('5Min') # by default label='right'
Out[117]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

```
In [118]: ts.resample('5Min', label='left')
Out[118]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

```
In [119]: ts.resample('5Min', label='left', loffset='1s')
Out[119]:
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 convenient or performant as the new pandas timeseries API.

### 16.8 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`.

#### 16.8.1 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 [120]: Period('2012', freq='A-DEC')
Out[120]: Period('2012', 'A-DEC')
```

```
In [121]: Period('2012-1-1', freq='D')
Out[121]: Period('2012-01-01', 'D')
```

```
In [122]: Period('2012-1-1 19:00', freq='H')
Out[122]: 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 [123]: p = Period('2012', freq='A-DEC')

In [124]: p + 1
Out[124]: Period('2013', 'A-DEC')

In [125]: p - 3
Out[125]: Period('2009', 'A-DEC')

Taking the difference of `Period` instances with the same frequency will return the number of frequency units between them:

In [126]: Period('2012', freq='A-DEC') - Period('2002', freq='A-DEC')
Out[126]: 10

### 16.8.2 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 [127]: prng = period_range('1/1/2011', '1/1/2012', freq='M')

In [128]: prng
Out[128]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2012-01]
length: 13

The `PeriodIndex` constructor can also be used directly:

In [129]: PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[129]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2011-03]
length: 3

Just like `DatetimeIndex`, a `PeriodIndex` can also be used to index pandas objects:

In [130]: Series(randn(len(prng)), prng)
Out[130]:
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
16.8.3 Frequency Conversion and Resampling with PeriodIndex

The frequency of Periods and PeriodIndex can be converted via the `asfreq` method. Let’s start with the fiscal year 2011, ending in December:

```python
In [131]: p = Period('2011', freq='A-DEC')
In [132]: p
Out[132]: Period('2011', 'A-DEC')
```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```python
In [133]: p.asfreq('M', how='start')
Out[133]: Period('2011-01', 'M')
In [134]: p.asfreq('M', how='end')
Out[134]: Period('2011-12', 'M')
```

The shorthands ‘s’ and ‘e’ are provided for convenience:

```python
In [135]: p.asfreq('M', 's')
Out[135]: Period('2011-01', 'M')
In [136]: p.asfreq('M', 'e')
Out[136]: Period('2011-12', 'M')
```

Converting to a “super-period” (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```python
In [137]: p = Period('2011-12', freq='M')
In [138]: p.asfreq('A-NOV')
Out[138]: 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 start and ends. Thus, first quarter of 2011 could start in 2010 or a few months into 2011. Via anchored frequencies, pandas works all quarterly frequencies Q-JAN through Q-DEC.

Q-DEC define regular calendar quarters:

```python
In [139]: p = Period('2012Q1', freq='Q-DEC')
In [140]: p.asfreq('D', 's')
Out[140]: Period('2012-01-01', 'D')
In [141]: p.asfreq('D', 'e')
Out[141]: Period('2012-03-31', 'D')
```

Q-MAR defines fiscal year end in March:

```python
In [142]: p = Period('2011Q4', freq='Q-MAR')
In [143]: p.asfreq('D', 's')
Out[143]: Period('2011-01-01', 'D')
In [144]: p.asfreq('D', 'e')
Out[144]: Period('2011-03-31', 'D')
```
16.9 Converting between Representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```
In [145]: rng = date_range('1/1/2012', periods=5, freq='M')
In [146]: ts = Series(randn(len(rng)), index=rng)
In [147]: ts
Out[147]:
2012-01-31  0.020601
2012-02-29 -0.411719
2012-03-31  2.079413
2012-04-30 -1.077911
2012-05-31  0.099258
Freq: M, dtype: float64
```

```
In [148]: ps = ts.to_period()
In [149]: ps
Out[149]:
2012-01  0.020601
2012-02 -0.411719
2012-03  2.079413
2012-04 -1.077911
2012-05  0.099258
Freq: M, dtype: float64
```

```
In [150]: ps.to_timestamp()
Out[150]:
2012-01-01  0.020601
2012-02-01 -0.411719
2012-03-01  2.079413
2012-04-01 -1.077911
2012-05-01  0.099258
Freq: MS, dtype: float64
```

Remember that 's' and 'e' can be used to return the timestamps at the start or end of the period:

```
In [151]: ps.to_timestamp('D', how='s')
Out[151]:
2012-01-01  0.020601
2012-02-01 -0.411719
2012-03-01  2.079413
2012-04-01 -1.077911
2012-05-01  0.099258
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 [152]: prng = period_range('1990Q1', '2000Q4', freq='Q-NOV')
In [153]: ts = Series(randn(len(prng)), prng)
In [154]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9
```
16.10 Time Zone Handling

Using pytz, pandas provides rich support for working with timestamps in different time zones. By default, pandas objects are time zone unaware:

```
In [156]: rng = date_range(’3/6/2012 00:00’, periods=15, freq=’D’)

In [157]: print(rng.tz)
None
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions:

```
In [158]: rng_utc = date_range(’3/6/2012 00:00’, periods=10, freq=’D’, tz=’UTC’)

In [159]: print(rng_utc.tz)
UTC
```

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 [160]: ts = Series(randn(len(rng)), rng)

In [161]: ts_utc = ts.tz_localize(’UTC’)

In [162]: ts_utc
```

```
Out[162]:
2012-03-06 00:00:00+00:00 -2.189293
2012-03-07 00:00:00+00:00 -1.819506
2012-03-08 00:00:00+00:00  0.229798
2012-03-09 00:00:00+00:00  0.119425
2012-03-10 00:00:00+00:00  1.808966
2012-03-11 00:00:00+00:00  1.015841
2012-03-12 00:00:00+00:00 -1.651784
2012-03-13 00:00:00+00:00  0.347674
2012-03-14 00:00:00+00:00  0.773688
2012-03-15 00:00:00+00:00  0.425863
2012-03-16 00:00:00+00:00  0.579486
2012-03-17 00:00:00+00:00 -0.745396
2012-03-18 00:00:00+00:00  0.141880
2012-03-19 00:00:00+00:00 -1.077754
2012-03-20 00:00:00+00:00 -1.301174
Freq: D, dtype: float64
```

You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```
In [163]: ts_utc.tz_convert(’US/Eastern’)

Out[163]:
```

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:

```
In [164]: rng_eastern = rng_utc.tz_convert('US/Eastern')

In [165]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [166]: rng_eastern[5]
Out[166]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [167]: rng_berlin[5]
Out[167]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

Out[168]: True
```

Like Series, DataFrame, and DatetimeIndex, Timestamps can be converted to other time zones using `tz_convert`:

```
In [169]: rng_eastern[5]
Out[169]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [170]: rng_berlin[5]
Out[170]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

In [171]: rng_eastern[5].tz_convert('Europe/Berlin')
Out[171]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of Timestamps functions just like DatetimeIndex and TimeSeries:

```
In [172]: rng[5]
Out[172]: Timestamp('2012-03-11 00:00:00', tz=None)

In [173]: rng[5].tz_localize('Asia/Shanghai')
Out[173]: Timestamp('2012-03-11 08:00:00+0800', tz='Asia/Shanghai')
```

Operations between TimeSeries in different time zones will yield UTC TimeSeries, aligning the data on the UTC timestamps:

```
In [174]: eastern = ts_utc.tz_convert('US/Eastern')

In [175]: berlin = ts_utc.tz_convert('Europe/Berlin')
```
In [176]: result = eastern + berlin

In [177]: result
Out[177]:
2012-03-06 00:00:00+00:00 -4.378586
2012-03-07 00:00:00+00:00 -3.639011
2012-03-08 00:00:00+00:00 0.459596
2012-03-09 00:00:00+00:00 0.238849
2012-03-10 00:00:00+00:00 3.617932
2012-03-11 00:00:00+00:00 2.031683
2012-03-12 00:00:00+00:00 -3.303568
2012-03-13 00:00:00+00:00 0.695349
2012-03-14 00:00:00+00:00 -1.547376
2012-03-15 00:00:00+00:00 0.851726
2012-03-16 00:00:00+00:00 1.158971
2012-03-17 00:00:00+00:00 -1.490793
2012-03-18 00:00:00+00:00 0.283760
2012-03-19 00:00:00+00:00 -2.155508
2012-03-20 00:00:00+00:00 -2.602348
Freq: D, dtype: float64

In [178]: result.index
Out[178]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-06, ..., 2012-03-20]
Length: 15, Freq: D, Timezone: UTC

In some cases, localize cannot determine the DST and non-DST hours when there are duplicates. This often happens when reading files that simply duplicate the hours. The infer_dst argument in tz_localize will attempt to determine the right offset.

In [179]: rng_hourly = DatetimeIndex([‘11/06/2011 00:00’, ‘11/06/2011 01:00’,
......:
‘11/06/2011 03:00’])

In [180]: rng_hourly.tz_localize(‘US/Eastern’)  
--------------------------------------------------------------------------- 
AmbiguousTimeError Traceback (most recent call last)
<ipython-input-180-8c5fa6a37f5b> in <module>()
----> 1 rng_hourly.tz_localize(’US/Eastern’)

/home/user1/src/pandas/pandas/tseries/index.pyc in tz_localize(self, tz, infer_dst)
 1606 # Convert to UTC
 1607 -> new_dates = tslib.tz_localize_to_utc(self.asi8, tz, infer_dst=infer_dst)
 1609 new_dates = new_dates.view(_NS_DTYPE)
 1610
/home/user1/src/pandas/pandas/tslib.so in pandas.tslib.tz_localize_to_utc (pandas/tslib.c:30832)()

AmbiguousTimeError: Cannot infer dst time from Timestamp(‘2011-11-06 01:00:00’, tz=None), try using the ‘infer_dst’ argument

In [181]: rng_hourly_eastern = rng_hourly.tz_localize(‘US/Eastern’, infer_dst=True)

In [182]: rng_hourly_eastern.values
Out[182]:
array([‘2011-11-06T06:00:00.000000000+0200’,

16.10. Time Zone Handling 403
16.11 Time Deltas

Timedeltas are differences in times, expressed in difference units, e.g. days, hours, minutes, seconds. They can be both positive and negative. DateOffsets that are absolute in nature (Day, Hour, Minute, Second, Milli, Micro, Nano) can be used as timedeltas.

```
In [183]: from datetime import datetime, timedelta

In [184]: s = Series(date_range('2012-1-1', periods=3, freq='D'))

In [185]: td = Series([timedelta(days=i) for i in range(3)])

In [186]: df = DataFrame(dict(A = s, B = td))

In [187]: df
Out[187]:
     A      B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

[3 rows x 2 columns]

In [188]: df['C'] = df['A'] + df['B']

In [189]: df
Out[189]:
     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

[3 rows x 3 columns]

In [190]: df.dtypes
Out[190]:
A    datetime64[ns]
B    timedelta64[ns]
C    datetime64[ns]
dtype: object

In [191]: s - s.max()
Out[191]:
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [192]: s - datetime(2011,1,1,3,5)
Out[192]:
0  364 days, 20:55:00
```


1 365 days, 20:55:00
2 366 days, 20:55:00
dtype: timedelta64[ns]

In [193]: s + timedelta(minutes=5)
Out[193]:
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 [194]: s + Minute(5)
Out[194]:
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 [195]: s + Minute(5) + Milli(5)
Out[195]:
0  2012-01-01 00:05:00.005000
1  2012-01-02 00:05:00.005000
2  2012-01-03 00:05:00.005000
dtype: datetime64[ns]

Getting scalar results from a timedelta64[ns] series

In [196]: y = s - s[0]

In [197]: y
Out[197]:
0   0 days
1   1 days
2   2 days
dtype: timedelta64[ns]

Series of timedeltas with NaT values are supported

In [198]: y = s - s.shift()

In [199]: y
Out[199]:
0   NaT
1   1 days
2   1 days
dtype: timedelta64[ns]

Elements can be set to NaT using np.nan analagously to datetimes

In [200]: y[1] = np.nan

In [201]: y
Out[201]:
0   NaT
1   NaT
2   1 days
dtype: timedelta64[ns]

Operands can also appear in a reversed order (a singluar object operated with a Series)
In [202]: s.max() - s
Out[202]:
0  2 days
1  1 days
2  0 days
dtype: timedelta64[ns]

In [203]: datetime(2011,1,1,3,5) - s
Out[203]:
0 -364 days, 20:55:00
1 -365 days, 20:55:00
2 -366 days, 20:55:00
dtype: timedelta64[ns]

In [204]: timedelta(minutes=5) + s
Out[204]:
0  2012-01-01 00:05:00
1  2012-01-02 00:05:00
2  2012-01-03 00:05:00
dtype: datetime64[ns]

Some timedelta numeric like operations are supported.

In [205]: td - timedelta(minutes=5, seconds=5, microseconds=5)
Out[205]:
0 -0 days, 00:05:05.000005
1  0 days, 23:54:54.999995
2  1 days, 23:54:54.999995
dtype: timedelta64[ns]

min, max and the corresponding idxmin, idxmax operations are supported on frames

In [206]: A = s - Timestamp('20120101') - timedelta(minutes=5, seconds=5)

In [207]: B = s - Series(date_range('2012-1-2', periods=3, freq='D'))

In [208]: df = DataFrame(dict(A=A, B=B))

In [209]: df
Out[209]:
     A          B
0 -0 days, 00:05:05 -1 days
1  0 days, 23:54:55 -1 days
2  1 days, 23:54:55 -1 days

[3 rows x 2 columns]

In [210]: df.min()
Out[210]:
A  -0 days, 00:05:05
B  -1 days, 00:00:00
dtype: timedelta64[ns]

In [211]: df.min(axis=1)
Out[211]:
0 -1 days
1 -1 days
2 -1 days
dtype: timedelta64[ns]
In [212]: df.idxmin()
Out[212]:
A  0
B  0
dtype: int64

In [213]: df.idxmax()
Out[213]:
A  2
B  0
dtype: int64

min, max operations are supported on series; these return a single element timedelta64[ns] Series (this avoids having to deal with numpy timedelta64 issues). idxmin, idxmax are supported as well.

In [214]: df.min().max()
Out[214]:
0 -00:05:05
dtype: timedelta64[ns]

In [215]: df.min(axis=1).min()
Out[215]:
0 -1 days
dtype: timedelta64[ns]

In [216]: df.min().idxmax()
Out[216]: 'A'

In [217]: df.min(axis=1).idxmin()
Out[217]: 0

You can fillna on timedeltas. Integers will be interpreted as seconds. You can pass a timedelta to get a particular value.

In [218]: y.fillna(0)
Out[218]:
0  0 days
1  0 days
2  1 days
dtype: timedelta64[ns]

In [219]: y.fillna(10)
Out[219]:
0  0 days, 00:00:10
1  0 days, 00:00:10
2  1 days, 00:00:00
dtype: timedelta64[ns]

In [220]: y.fillna(timedelta(days=-1,seconds=5))
Out[220]:
0  -0 days, 23:59:55
1  -0 days, 23:59:55
2  1 days, 00:00:00
dtype: timedelta64[ns]
16.12 Time Deltas & Reductions

**Warning:** A numeric reduction operation for `timedelta64[ns]` can return a single-element `Series` of `dtype timedelta64[ns]`.

You can do numeric reduction operations on `timedeltas`.

In [221]: y2 = y.fillna(timedelta(days=-1,seconds=5))

In [222]: y2
Out[222]:
0    -0 days, 23:59:55
1    -0 days, 23:59:55
2     1 days, 00:00:00
dtype: timedelta64[ns]

In [223]: y2.mean()
Out[223]:
0    -07:59:56.666667
dtype: timedelta64[ns]

In [224]: y2.quantile(.1)
Out[224]: numpy.timedelta64(-86395000000000,'ns')

16.13 Time Deltas & Conversions

New in version 0.13. **string/integer conversion**

Using the 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). It can also construct `Series`.

**Warning:** This requires `numpy` >= 1.7

In [225]: to_timedelta('1 days 06:05:01.00003')
Out[225]: numpy.timedelta64(108301000030000,'ns')

In [226]: to_timedelta('15.5us')
Out[226]: numpy.timedelta64(15500,'ns')

In [227]: to_timedelta([1 days 06:05:01.00003',15.5us','nan'])
Out[227]:
0   1 days, 06:05:01.000030
1    0 days, 00:00:00.000016
2        NaT
dtype: timedelta64[ns]

In [228]: to_timedelta(np.arange(5),unit='s')
Out[228]:
0    00:00:00
1    00:00:01
2    00:00:02
3    00:00:03
4    00:00:04
dtype: timedelta64[ns]
In [229]: to_timedelta(np.arange(5), unit='d')
Out[229]:
0   0 days
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]

frequency conversion

Timedeltas can be converted to other ‘frequencies’ by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield float64 dtyped Series.

In [230]: td = Series(date_range('20130101', periods=4)) - Series(date_range('20121201', periods=4))
In [231]: td[2] += np.timedelta64(timedelta(minutes=5, seconds=3))
In [232]: td[3] = np.nan
In [233]: td
Out[233]:
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 [234]: td / np.timedelta64(1, 'D')
Out[234]:
0   31.000000
1   31.000000
2   31.003507
3      NaN
dtype: float64

In [235]: td.astype('timedelta64[D]')
Out[235]:
0   31
1   31
2   31
3      NaN
dtype: float64

# to seconds
In [236]: td / np.timedelta64(1, 's')
Out[236]:
0  2678400
1  2678400
2  2678703
3      NaN
dtype: float64

In [237]: td.astype('timedelta64[s]')
Out[237]:
0  2678400
1  2678400

16.13. Time Deltas & Conversions
Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series yields another `timedelta64[ns]` dtypes Series.

In [238]: td * -1
Out[238]:
0  -31 days, 00:00:00
1   -31 days, 00:00:00
2  -31 days, 00:05:03
3    NaT
dtype: timedelta64[ns]

In [239]: td * Series([1, 2, 3, 4])
Out[239]:
0    31 days, 00:00:00
1    62 days, 00:00:00
2    93 days, 00:15:09
3     NaT
dtype: timedelta64[ns]

### 16.13.1 Numpy < 1.7 Compatibility

Numpy < 1.7 has a broken `timedelta64` type that does not correctly work for arithmetic. Pandas bypasses this, but for frequency conversion as above, you need to create the divisor yourself. The `np.timedelta64` type only has 1 argument, the number of `micro` seconds.

The following are equivalent statements in the two versions of numpy.

```python
from distutils.version import LooseVersion
if LooseVersion(np.__version__) <= '1.6.2':
    y / np.timedelta(86400*int(1e6))
    y / np.timedelta(int(1e6))
else:
    y / np.timedelta64(1,'D')
    y / np.timedelta64(1,'s')
```
Note: We intend to build more plotting integration with matplotlib as time goes on.

We use the standard convention for referencing the matplotlib API:

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

### 17.1 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 = Series(randn(1000), index=date_range('1/1/2000', periods=1000))
In [3]: ts = ts.cumsum()
In [4]: ts.plot()
Out[4]: <matplotlib.axes.AxesSubplot at 0x5dfb2d0>
```
If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above. The method takes a number of arguments for controlling the look of the plot:

```
In [5]: plt.figure(); ts.plot(style='k--', label='Series'); plt.legend()
Out[5]: <matplotlib.legend.Legend at 0xfd43210>
```

On DataFrame, `plot` is a convenience to plot all of the columns with labels:
In [6]: df = DataFrame(randn(1000, 4), index=ts.index, columns=list('ABCD'))

In [7]: df = df.cumsum()

In [8]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[8]: <matplotlib.legend.Legend at 0x12f2b590>

You may set the legend argument to False to hide the legend, which is shown by default.

In [9]: df.plot(legend=False)
Out[9]: <matplotlib.axes.AxesSubplot at 0x12f2b590>
Some other options are available, like plotting each Series on a different axis:

```python
In [10]: df.plot(subplots=True, figsize=(6, 6)); plt.legend(loc='best')
Out[10]: <matplotlib.legend.Legend at 0x13024d50>
```
You may pass `logy` to get a log-scale Y axis.

In [11]: plt.figure();

In [12]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [13]: ts = np.exp(ts.cumsum())

In [14]: ts.plot(logy=True)
Out[14]: <matplotlib.axes.AxesSubplot at 0x12fe8490>
You can plot one column versus another using the `x` and `y` keywords in `DataFrame.plot`:

```python
In [15]: plt.figure()
Out[15]: <matplotlib.figure.Figure at 0x144bf810>

In [16]: df3 = DataFrame(randn(1000, 2), columns=['B', 'C']).cumsum()

In [17]: df3['A'] = Series(list(range(len(df))))

In [18]: df3.plot(x='A', y='B')
Out[18]: <matplotlib.axes.AxesSubplot at 0x14f20e50>
```
17.1.1 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```python
In [19]: plt.figure()
Out[19]: <matplotlib.figure.Figure at 0x144f2450>

In [20]: df.A.plot()
Out[20]: <matplotlib.axes.AxesSubplot at 0x14f20d90>

In [21]: df.B.plot(secondary_y=True, style='g')
Out[21]: <matplotlib.axes.AxesSubplot at 0x98edf90>
```
17.1.2 Selective Plotting on Secondary Y-axis

To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

In [22]: plt.figure()
Out[22]: <matplotlib.figure.Figure at 0x144caa10>

In [23]: ax = df.plot(secondary_y=['A', 'B'])

In [24]: ax.set_ylabel('CD scale')
Out[24]: <matplotlib.text.Text at 0x148c4910>

In [25]: ax.right_ax.set_ylabel('AB scale')
Out[25]: <matplotlib.text.Text at 0x151ac950>
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:

In [26]: plt.figure()
Out[26]: <matplotlib.figure.Figure at 0x151b3050>

In [27]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[27]: <matplotlib.axes.AxesSubplot at 0x148f0290>
17.1.3 Suppressing tick resolution adjustment

Pandas includes automatically 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 [28]: plt.figure()
Out[28]: <matplotlib.figure.Figure at 0x13057ad0>

In [29]: df.A.plot()
Out[29]: <matplotlib.axes.AxesSubplot at 0x130572d0>
```
Using the `x_compat` parameter, you can suppress this behavior:

```
In [30]: plt.figure()
Out[30]: <matplotlib.figure.Figure at 0x13057d50>

In [31]: df.A.plot(x_compat=True)
Out[31]: <matplotlib.axes.AxesSubplot at 0x73208d0>
```

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:

```python
In [32]: import pandas as pd
```

```python
In [33]: plt.figure()
Out[33]: <matplotlib.figure.Figure at 0x12f7fad0>
```

```python
In [34]: with pd.plot_params.use('x_compat', True):
   ....:     df.A.plot(color='r')
   ....:     df.B.plot(color='g')
   ....:     df.C.plot(color='b')
   ....:
```

17.1.4 Targeting different subplots

You can pass an ax argument to Series.plot to plot on a particular axis:

```python
In [35]: fig, axes = plt.subplots(nrows=2, ncols=2)
```

```python
In [36]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A')
Out[36]: <matplotlib.text.Text at 0x148f5c90>
```

```python
In [37]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B')
Out[37]: <matplotlib.text.Text at 0x98ec550>
```

```python
In [38]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C')
Out[38]: <matplotlib.text.Text at 0x146b6e90>
```

```python
In [39]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D')
Out[39]: <matplotlib.text.Text at 0x146cb690>
```
17.2 Other plotting features

17.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

In [40]: plt.figure();

In [41]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')

Out[41]: <matplotlib.lines.Line2D at 0x97c8b50>
Calling a DataFrame’s `plot` method with `kind='bar'` produces a multiple bar plot:

```
In [42]: df2 = DataFrame(rand(10, 4), columns=['a', 'b', 'c', 'd'])
In [43]: df2.plot(kind='bar');
```

To produce a stacked bar plot, pass `stacked=True`:
In [44]: df2.plot(kind='bar', stacked=True);

To get horizontal bar plots, pass kind='barh':

In [45]: df2.plot(kind='barh', stacked=True);
17.2.2 Histograms

In [46]: plt.figure();

In [47]: df['A'].diff().hist()
Out[47]: <matplotlib.axes.AxesSubplot at 0x16376b10>

For a DataFrame, hist plots the histograms of the columns on multiple subplots:

In [48]: plt.figure()
Out[48]: <matplotlib.figure.Figure at 0x16afefd0>

In [49]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[49]:
array([[<matplotlib.axes.AxesSubplot object at 0x16afef0>,
       <matplotlib.axes.AxesSubplot object at 0x169a6a50>],
       [<matplotlib.axes.AxesSubplot object at 0x16c37750>,
       <matplotlib.axes.AxesSubplot object at 0x16c4a250>]], dtype=object)
New since 0.10.0, the `by` keyword can be specified to plot grouped histograms:

In [50]: data = Series(randn(1000))

In [51]: data.hist(by=randint(0, 4, 1000), figsize=(6, 4))
Out[51]:
array([[[<matplotlib.axes.AxesSubplot object at 0x171c73d0>,
          <matplotlib.axes.AxesSubplot object at 0x179df8d0>],
        [<matplotlib.axes.AxesSubplot object at 0x179fac10>,
          <matplotlib.axes.AxesSubplot object at 0x17b1f610>]],
       dtype=object)
17.2.3 Box-Plotting

DataFrame has a boxplot method which allows you to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

In [52]: df = DataFrame(rand(10,5))

In [53]: plt.figure();

In [54]: bp = df.boxplot()
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [55]: df = DataFrame(rand(10,2), columns=['Col1', 'Col2'] )
```

```
In [56]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])
```

```
In [57]: plt.figure();
```

```
In [58]: bp = df.boxplot(by='X')
```
You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [59]: df = DataFrame(rand(10,3), columns=['Col1', 'Col2', 'Col3'])

In [60]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])

In [61]: df['Y'] = Series(['A','B','A','B','A','B','A','B','A','B'])

In [62]: plt.figure();

In [63]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```
17.2.4 Scatter plot matrix

New in 0.7.3. You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.tools.plotting`:

In [64]: from pandas.tools.plotting import scatter_matrix

In [65]: df = DataFrame(randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [66]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')

Out [66]:
array([[<matplotlib.axes.AxesSubplot object at 0x169ae4d0>,
        <matplotlib.axes.AxesSubplot object at 0x17df0350>,
        <matplotlib.axes.AxesSubplot object at 0x17e07f90>,
        <matplotlib.axes.AxesSubplot object at 0x1818d2d0>],
       [<matplotlib.axes.AxesSubplot object at 0x17f70e90>,
        <matplotlib.axes.AxesSubplot object at 0x1699b410>,
        <matplotlib.axes.AxesSubplot object at 0x1302a790>,
        <matplotlib.axes.AxesSubplot object at 0x15cb42d0>],
       [<matplotlib.axes.AxesSubplot object at 0x1637d190>,
        <matplotlib.axes.AxesSubplot object at 0x169ae4d0>,
        <matplotlib.axes.AxesSubplot object at 0x17df0350>,
        <matplotlib.axes.AxesSubplot object at 0x1818d2d0>],
       [<matplotlib.axes.AxesSubplot object at 0x1816ef90>,
        <matplotlib.axes.AxesSubplot object at 0x16376590>,
        <matplotlib.axes.AxesSubplot object at 0x164eece10>,
        <matplotlib.axes.AxesSubplot object at 0x18421790>]], dtype=object)
can create density plots using the Series/DataFrame.plot and setting `kind='kde'`:

```
In [67]: ser = Series(randn(1000))

In [68]: ser.plot(kind='kde')
Out[68]: <matplotlib.axes.AxesSubplot at 0x18a9ee90>
```
17.2.5 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 [69]: from pandas import read_csv

In [70]: from pandas.tools.plotting import andrews_curves

In [71]: data = read_csv('data/iris.data')

In [72]: plt.figure()
Out[72]: <matplotlib.figure.Figure at 0x18ff5390>

In [73]: andrews_curves(data, 'Name')
Out[73]: <matplotlib.axes.AxesSubplot at 0x18ff5710>
17.2.6 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 [74]: from pandas import read_csv

In [75]: from pandas.tools.plotting import parallel_coordinates

In [76]: data = read_csv('data/iris.data')

In [77]: plt.figure()
Out[77]: <matplotlib.figure.Figure at 0x19722450>

In [78]: parallel_coordinates(data, 'Name')
Out[78]: <matplotlib.axes.AxesSubplot at 0x19722550>
```

17.2.7 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 [79]: from pandas.plotting import lag_plot

In [80]: plt.figure()
Out[80]: <matplotlib.figure.Figure at 0x19db5790>

In [81]: data = Series(0.1 * rand(1000) +
....: 0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
....:

In [82]: lag_plot(data)
Out[82]: <matplotlib.axes.AxesSubplot at 0x19d857d0>
17.2.8 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 [83]: from pandas.tools.plotting import autocorrelation_plot

In [84]: plt.figure()
Out[84]: <matplotlib.figure.Figure at 0x19d9dd0>

In [85]: data = Series(0.7 * rand(1000) +
....: 0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
....:

In [86]: autocorrelation_plot(data)
Out[86]: <matplotlib.axes.AxesSubplot at 0x19d9dd0>
17.2.9 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 [87]: from pandas.tools.plotting import bootstrap_plot
In [88]: data = Series(rand(1000))
In [89]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[89]: <matplotlib.figure.Figure at 0x199eb910>
17.2.10 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 [90]: from pandas import read_csv

In [91]: from pandas.tools.plotting import radviz

In [92]: data = read_csv('data/iris.data')

In [93]: plt.figure()
Out[93]: <matplotlib.figure.Figure at 0x1957b450>

In [94]: radviz(data, 'Name')
Out[94]: <matplotlib.axes.AxesSubplot at 0x1956f3d0>
17.2.11 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 jet colormap, we can simply pass ‘jet’ to colormap=

```
In [95]: df = DataFrame(randn(1000, 10), index=ts.index)

In [96]: df = df.cumsum()

In [97]: plt.figure()
Out[97]: <matplotlib.figure.Figure at 0x1a64b990>

In [98]: df.plot(colormap='jet')
Out[98]: <matplotlib.axes.AxesSubplot at 0x19709a10>
```
or we can pass the colormap itself

```
In [99]: from matplotlib import cm

In [100]: plt.figure()
Out[100]: <matplotlib.figure.Figure at 0x179ec350>

In [101]: df.plot(colormap=cm.jet)
Out[101]: <matplotlib.axes.AxesSubplot at 0x19722b90>
```
Colormaps can also be used other plot types, like bar charts:

```
In [102]: dd = DataFrame(randn(10, 10)).applymap(abs)
In [103]: dd = dd.cumsum()
In [104]: plt.figure()
Out[104]: <matplotlib.figure.Figure at 0x15cba590>
In [105]: dd.plot(kind='bar', colormap='Greens')
Out[105]: <matplotlib.axes.AxesSubplot at 0x15cc6d50>
```
Parallel coordinates charts:

In [106]: plt.figure()
Out[106]: <matplotlib.figure.Figure at 0x19c09a50>

In [107]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
Out[107]: <matplotlib.axes.AxesSubplot at 0x19826410>

Andrews curves charts:
In [108]: plt.figure()
Out[108]: <matplotlib.figure.Figure at 0x192f4310>

In [109]: andrews_curves(data, 'Name', colormap='winter')
Out[109]: <matplotlib.axes.AxesSubplot at 0x19006290>
Note: The tips data set can be downloaded here. Once you download it execute

```python
from pandas import read_csv
tips_data = read_csv('tips.csv')
```

from the directory where you downloaded the file.

We import the rplot API:

```python
In [1]: import pandas.tools.rplot as rplot
```

## 18.1 Examples

RPlot is a flexible API for producing Trellis plots. These plots allow you to arrange data in a rectangular grid by values of certain attributes.

```python
In [2]: plt.figure()
Out[2]: <matplotlib.figure.Figure at 0x103f7f50>

In [3]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [4]: plot.add(rplot.TrellisGrid([‘sex’, ‘smoker’]))

In [5]: plot.add(rplot.GeomHistogram())

In [6]: plot.render(plt.gcf())
Out[6]: <matplotlib.figure.Figure at 0x103f7f50>
```
In the example above, 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 [7]: plt.figure()
Out[7]: <matplotlib.figure.Figure at 0x88a3d90>

In [8]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [9]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [10]: plot.add(rplot.GeomDensity())

In [11]: plot.render(plt.gcf())
Out[11]: <matplotlib.figure.Figure at 0x88a3d90>
```
Example above is the same as previous except the plot is set to kernel density estimation. This shows how easy it is to have different plots for the same Trellis structure.

In [12]: plt.figure()
Out[12]: <matplotlib.figure.Figure at 0xea38cd0>

In [13]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [14]: plot.add(rplot.TrellisGrid(["sex", 'smoker']))

In [15]: plot.add(rplot.GeomScatter())

In [16]: plot.add(rplot.GeomPolyFit(degree=2))

In [17]: plot.render(plt.gcf())
Out[17]: <matplotlib.figure.Figure at 0xea38cd0>
The plot above shows that it is possible to have two or more plots for the same data displayed on the same Trellis grid cell.

In [18]: plt.figure()
Out[18]: <matplotlib.figure.Figure at 0x1302aed0>

In [19]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [20]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [21]: plot.add(rplot.GeomScatter())

In [22]: plot.add(rplot.GeomDensity2D())

In [23]: plot.render(plt.gcf())
Out[23]: <matplotlib.figure.Figure at 0x1302aed0>
Above is a similar plot but with 2D kernel desntity estimation plot superimposed.

```python
In [24]: plt.figure()
Out[24]: <matplotlib.figure.Figure at 0x12f25c10>

In [25]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [26]: plot.add(rplot.TrellisGrid([‘sex’, ’.’]))

In [27]: plot.add(rplot.GeomHistogram())

In [28]: plot.render(plt.gcf())
Out[28]: <matplotlib.figure.Figure at 0x12f25c10>
```
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 [29]: plt.figure()
Out[29]: <matplotlib.figure.Figure at 0x12fb6d90>

In [30]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [31]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [32]: plot.add(rplot.GeomHistogram())

In [33]: plot.render(plt.gcf())
Out[33]: <matplotlib.figure.Figure at 0x12fb6d90>
If the first grouping attribute is not specified the plots will be arranged in a row.

In [34]: plt.figure()
Out [34]: <matplotlib.figure.Figure at 0x12fa55d0>

In [35]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [36]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [37]: plot.add(rplot.GeomHistogram())

In [38]: plot = rplot.RPlot(tips_data, x='tip', y='total_bill')

In [39]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [40]: plot.add(rplot.GeomPoint(size=80.0, colour=rplot.ScaleRandomColour('day'), shape=rplot.ScaleShape('size'), alpha=1.0))

In [41]: plot.render(plt.gcf())
Out [41]: <matplotlib.figure.Figure at 0x12fa55d0>
As shown above, scatter plots are also possible. Scatter plots allow you to map various data attributes to graphical properties of the plot. In the example above 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.

### 18.2 Scales

ScaleGradient(column, colour1, colour2)

This one allows you to map an attribute (specified by parameter column) value to the colour of a graphical object. The larger the value of the attribute the closer the colour will be to colour2, the smaller the value, the closer it will be to colour1.

ScaleGradient2(column, colour1, colour2, colour3)

The same as ScaleGradient but interpolates linearly between three colours instead of two.

ScaleSize(column, min_size, max_size, transform)
Map attribute value to size of the graphical object. Parameter min_size (default 5.0) is the minimum size of the graphical object, max_size (default 100.0) is the maximum size and transform is a one argument function that will be used to transform the attribute value (defaults to lambda x: x).

ScaleShape(column)

Map the shape of the object to attribute value. The attribute has to be categorical.

ScaleRandomColour(column)

Assign a random colour to a value of categorical attribute specified by column.
IO TOOLS (TEXT, CSV, HDF5, ...)

The Pandas I/O api is a set of top level reader functions accessed like `pd.read_csv()` that generally return a pandas object.

- `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`
19.1 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. `read_csv` is capable of inferring 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.
- **dialect**: string or `pandas.core.dtypes.dtypes.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.
- **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)
- **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 `pandas.core.iotype.dateutil.parser`. Specifying this implicitly sets `parse_dates` as `True`. You can also use functions from community supported date converters from `pandas.core.iotype.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**: denotes the start of a comment and ignores the rest of the line. Currently line commenting is not supported.

• **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)

• **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’.

• **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

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
```
In the case of indexed data, you can pass the column number or column name you wish to use as the index:

```
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

[3 rows x 3 columns]
```

```
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

[3 rows x 3 columns]
```

You can also use a list of columns to create a hierarchical index:

```
In [5]: pd.read_csv('foo.csv', index_col=[0, 'A'])
Out[5]:
   B  C
date  A
20090101 a 1 2
20090102 b 3 4
20090103 c 4 5

[3 rows x 2 columns]
```

The `dialect` keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a `csv.Dialect` instance.

Suppose you had data with unenclosed quotes:

```
In [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]:
```

458 Chapter 19. IO Tools (Text, CSV, HDF5, ...)
All of the dialect options can be specified separately by keyword arguments:

```python
In [10]: data = 'a,b,c~1,2,3~4,5,6'
In [11]: pd.read_csv(StringIO(data), lineterminator='~')
```

Another common dialect option is `skipinitialspace`, to skip any whitespace after a delimiter:

```python
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)
```

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.

### 19.1.1 Specifying column data types

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

```python
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
```

19.1. CSV & Text files
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

19.1.2 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]: from StringIO import StringIO

In [23]: data = 'a,b,c
1,2,3
4,5,6
7,8,9'

In [24]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [25]: pd.read_csv(StringIO(data))
Out[25]:
   a  b  c
  0  1  2  3
  1  4  5  6
  2  7  8  9

[3 rows x 3 columns]

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 [26]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [27]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[27]:
   foo  bar  baz
  0   1   2   3
  1   4   5   6
  2   7   8   9

[3 rows x 3 columns]
In [28]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[28]:
    foo  bar  baz
0    a    b    c
1    1    2    3
2    4    5    6
3    7    8    9

[4 rows x 3 columns]

If the header is in a row other than the first, pass the row number to header. This will skip the preceding rows:

In [29]: data = 'skip this skip it

  a,b,c

1,2,3
4,5,6
7,8,9'

In [30]: pd.read_csv(StringIO(data), header=1)
Out[30]:
    a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

[3 rows x 3 columns]

19.1.3 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:

In [31]: data = 'a,b,c,d

    1,2,3,foo
4,5,6,bar
7,8,9,baz'

In [32]: pd.read_csv(StringIO(data))
Out[32]:
    a  b  c  d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

[3 rows x 4 columns]

In [33]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[33]:
    b  d
0  2  foo
1  5  bar
2  8  baz

[3 rows x 2 columns]

In [34]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[34]:
    a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz

[3 rows x 3 columns]
19.1.4 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 [35]: data = b'word,length
       Träumen,7
       Grüße,5'.decode('utf8').encode('latin-1')
In [36]: df = pd.read_csv(StringIO(data), encoding='latin-1')
In [37]: df
Out[37]:
       word  length
0  Träumen     7
1    Grüße     5
[2 rows x 2 columns]
In [38]: df['word'][1]
Out[38]: u'Grüße'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won’t parse correctly at all without specifying the encoding.

19.1.5 Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the DataFrame’s row names:

```python
In [39]: data = 'a,b,c
       4,apple,bat,5.7
       8,orange,cow,10'
In [40]: pd.read_csv(StringIO(data))
Out[40]:
     a  b  c
4 apple bat 5.7
8 orange cow 10.0
[2 rows x 3 columns]
In [41]: data = 'index,a,b,c
       4,apple,bat,5.7
       8,orange,cow,10'
In [42]: pd.read_csv(StringIO(data), index_col=0)
Out[42]:
    a   b   c
index
4 apple bat 5.7
8 orange cow 10.0
[2 rows x 3 columns]
```

Ordinarily, you can achieve this behavior using the `index_col` option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass `index_col=False`:

```python
In [43]: data = 'a,b,c\n       4,apple,\n       8,orange,'
In [44]: print(data)
a,b,c
```
4, apple, bat,
8, orange, cow,

In [45]: pd.read_csv(StringIO(data))
Out[45]:
   a  b  c
0  4  apple  bat  NaN
1  8  orange  cow  NaN
[2 rows x 3 columns]

In [46]: pd.read_csv(StringIO(data), index_col=False)
Out[46]:
   a  b  c
0  4  apple  bat
1  8  orange  cow
[2 rows x 3 columns]

19.1.6 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 [47]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [48]: df
Out[48]:
   A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
[3 rows x 3 columns]

# These are python datetime objects
In [49]: df.index
Out[49]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01, ..., 2009-01-03]
Length: 3, Freq: None, Timezone: 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 [50]: 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
In [51]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [52]: df
Out[52]:
   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

[6 rows x 4 columns]

By default the parser removes the component date columns, but you can choose to retain them via the
keep_date_col keyword:

In [53]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   : keep_date_col=True)
   :

In [54]: df
Out[54]:
   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

[6 rows x 7 columns]

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 [55]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [56]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [57]: df
Out[57]:
   nominal actual  0  4
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:

```python
In [58]: date_spec = {‘nominal’: [1, 2], ‘actual’: [1, 3]}

In [59]: df = pd.read_csv(‘tmp.csv’, header=None, parse_dates=date_spec,
        index_col=0) #index is the nominal column

In [60]: df
Out[60]:
          actual   0   4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00 KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00 KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59

[6 rows x 3 columns]
```

**Note:** `read_csv` has a fast_path for parsing datetime strings in iso8601 format, e.g. “2000-01-01T00:01:02+00:00” and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

**Note:** When passing a dict as the `parse_dates` argument, the order of the columns prepended is not guaranteed, because `dict` objects do not impose an ordering on their keys. On Python 2.7+ you may use `collections.OrderedDict` instead of a regular `dict` if this matters to you. Because of this, when using a dict for ‘parse_dates’ in conjunction with the `index_col` argument, it’s best to specify `index_col` as a column label rather then as an index on the resulting frame.

### 19.1.7 Date Parsing Functions

Finally, the parser allows you can specify a custom `date_parser` function to take full advantage of the flexiblity of the date parsing API:

```python
In [61]: import pandas.io.date_converters as conv

In [62]: df = pd.read_csv(‘tmp.csv’, header=None, parse_dates=date_spec,
        date_parser=conv.parse_date_time)

In [63]: df
Out[63]:
          actual   0   4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00 KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00 KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59

[6 rows x 3 columns]
```

19.1. CSV & Text files 465
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

[6 rows x 4 columns]

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.

### 19.1.8 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 [64]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
                  ....:       infer_datetime_format=True)

In [65]: df
Out[65]:
    A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5

[3 rows x 3 columns]
```
19.1.9 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 [66]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c
```

```python
In [67]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[67]:
   date  value  cat
0 2000-01-06   5   a
1 2000-02-06  10   b
2 2000-03-06  15   c
[3 rows x 3 columns]
```

```python
In [68]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[68]:
   date  value  cat
0 2000-06-01   5   a
1 2000-06-02  10   b
2 2000-06-03  15   c
[3 rows x 3 columns]
```

19.1.10 Thousand Separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings

```python
In [69]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z
```

```python
In [70]: df = pd.read_csv('tmp.csv', sep='|')
```

```python
In [71]: df
Out[71]:
   ID   level  category
0 Patient1  123,000       x
1 Patient2    23,000       y
2 Patient3  1,234,018     z
[3 rows x 3 columns]
```

```python
In [72]: df.level.dtype
Out[72]: dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly
In [73]: `print(open('tmp.csv').read())`
ID|level|category
---|-----|-----
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [74]: `df = pd.read_csv('tmp.csv', sep='|', thousands=',')`
In [75]: `df`
Out[75]:
   ID   level  category
0  Patient1  123000      x
1  Patient2   23000      y
2  Patient3  1234018     z

[3 rows x 3 columns]

In [76]: `df.level.dtype`
Out[76]: dtype('int64')

19.1.11 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

19.1.12 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`.

19.1.13 Comments

Sometimes comments or meta data may be included in a file:
In [77]: `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 [78]: df = pd.read_csv('tmp.csv')
In [79]: df
Out[79]:
     ID      level category
0  Patient1   123000      x  # really unpleasant
1  Patient2   23000      y  # wouldn't take his medicine
2  Patient3  1234018      z  # awesome

[3 rows x 3 columns]

We can suppress the comments using the `comment` keyword:

In [80]: df = pd.read_csv('tmp.csv', comment='#')
In [81]: df
Out[81]:
     ID      level category
0  Patient1   123000      x
1  Patient2   23000      y
2  Patient3  1234018      z

[3 rows x 3 columns]

### 19.1.14 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a `Series`:

In [82]: `print(open('tmp.csv').read())`
level  
Patient1,123000  
Patient2,23000  
Patient3,1234018

In [83]: output = pd.read_csv('tmp.csv', squeeze=True)
In [84]: output
Out[84]:
     Patient1   123000  
 Patient2   23000  
 Patient3  1234018  
Name: level, dtype: int64

In [85]: `type(output)`
Out[85]: pandas.core.series.Series
19.1.15 Boolean values

The common values True, False, TRUE, and FALSE are all recognized as boolean. Sometime you would want to recognize some other values as being boolean. To do this use the true_values and false_values options:

In [86]: data= 'a,b,c
1,Yes,2
3,No,4'

In [87]: print(data)
a,b,c
1,Yes,2
3,No,4

In [88]: pd.read_csv(StringIO(data))
Out[88]:
    a  b  c
0  1  Yes  2
1  3  No   4
[2 rows x 3 columns]

In [89]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[89]:
    a  b  c
0  1  True  2
1  3  False 4
[2 rows x 3 columns]

19.1.16 Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many will cause an error by default:

In [27]: data = 'a,b,c\n1,2,3
4,5,6,7\n8,9,10'

In [28]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
CParserError Traceback (most recent call last)
CParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4

You can elect to skip bad lines:

In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
Skipping line 3: expected 3 fields, saw 4

Out[29]:
    a  b  c
0  1  2  3
1  8  9 10

19.1.17 Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the escapechar option:
In [90]: data = ‘a,b
   \n   "hello, \"Bob\", nice to see you",5’

In [91]: print(data)
   a,b
   "hello, "Bob\", nice to see you",5

In [92]: pd.read_csv(StringIO(data), escapechar='\')
Out[92]:
   a   b
  0 hello, "Bob", nice to see you   5

[1 rows x 2 columns]

19.1.18 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:

In [93]: print(open(‘bar.csv’).read())
id8141 360.242940 149.910199 11950.7
id1594 444.953632 166.985655 11788.4
id1849 364.136849 183.628767 11806.2
id1230 413.836124 184.375703 11916.8
id1948 502.953953 173.237159 12468.3

In order to parse this file into a DataFrame, we simply need to supply the column specifications to the `read_fwf` function along with the file name:

#Column specifications are a list of half-intervals
In [94]: cols = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [95]: df = pd.read_fwf(‘bar.csv’, cols=colspecs, header=None, index_col=0)

In [96]: df
Out[96]:
   1     2     3
  0  id8141  360.242940  149.910199  11950.7
    id1594  444.953632  166.985655  11788.4
    id1849  364.136849  183.628767  11806.2
    id1230  413.836124  184.375703  11916.8
    id1948  502.953953  173.237159  12468.3

[5 rows x 3 columns]

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

#Widths are a list of integers
In [97]: widths = [6, 14, 13, 10]
In [98]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [99]: df
Out[99]:
   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
[5 rows x 4 columns]

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).

In [100]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [101]: df
Out[101]:
    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
[5 rows x 3 columns]

19.1.19 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

In [102]: 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 [103]: pd.read_csv('foo.csv')
Out[103]:
   A  B  C
20090101 a 1 2
20090102 b 3 4
20090103 c 4 5
[3 rows x 3 columns]

Note that the dates weren’t automatically parsed. In that case you would need to do as before:

In [104]: df = pd.read_csv('foo.csv', parse_dates=True)
In [105]: df.index
Out[105]:
<class ‘pandas.tseries.index.DatetimeIndex’>
[2009-01-01, ..., 2009-01-03]
Length: 3, Freq: None, Timezone: None

19.1.20 Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

In [106]: 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 [107]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])

In [108]: df
Out[108]:
   zit  xit
year indiv
1977 A  1.20  0.60
      B  1.50  0.50
      C  1.70  0.80
1978 A  0.20  0.06
      B  0.70  0.20
      C  0.80  0.30
      D  0.90  0.50
      E  1.40  0.90
1979 C  0.20  0.15
      D  0.14  0.05
      E  0.50  0.15
      F  1.20  0.50
      G  3.40  1.90
      H  5.40  2.70
      I  6.40  1.20

[15 rows x 2 columns]

In [109]: df.ix[1978]
Out[109]:
19.1.21 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 interveaining rows. In order to have the pre-0.13 behavior of tupleizing columns, specify tupleize_cols=True.

In [110]: from pandas.util.testing import makeCustomDataframe as mkdf

In [111]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)

In [112]: df.to_csv('mi.csv')

In [113]: print(open('mi.csv').read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2

In [114]: pd.read_csv('mi.csv',header=[0,1,2,3],index_col=[0,1])

Out[114]:
C0        C_l0_g0  C_l0_g1  C_l0_g2
C1        C_l1_g0  C_l1_g1  C_l1_g2
C2        C_l2_g0  C_l2_g1  C_l2_g2
C3        C_l3_g0  C_l3_g1  C_l3_g2
R0        R1
R_l0_g0   R_l1_g0  R0C0    R0C1    R0C2
R_l0_g1   R_l1_g1  R1C0    R1C1    R1C2
R_l0_g2   R_l1_g2  R2C0    R2C1    R2C2
R_l0_g3   R_l1_g3  R3C0    R3C1    R3C2
R_l0_g4   R_l1_g4  R4C0    R4C1    R4C2

[5 rows x 3 columns]

Starting in 0.13.0, read_csv will be able to interpret a more common format of multi-columns indices.

In [115]: 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 [116]: pd.read_csv('mi2.csv', header=[0,1], index_col=0)
Out[116]:
   a   b   c
  q  r  s  t  u  v
one 1  2  3  4  5  6
two 7  8  9 10 11 12

[2 rows x 6 columns]

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.

19.1.22 Automatically “sniffing” the delimiter

read_csv is capable of inferring delimited (not necessarily comma-separated) files. YMMV, as pandas uses the csv.Sniffer class of the csv module.

In [117]: print(open('tmp2.sv').read())
0:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-0.1042359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.721551622443669:-0.7067711336300845:-1.0395749851146963:0.2718598854282986
4:-0.4249723297883753:0.56702349793672:0.27623201927711873:-1.08400691259915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.715002316146375:-0.139268835147725
7:-0.370468582364464:1.1578922506419993:-1.344311812631667:0.8448851414248841
8:0.757697831715533:-0.10904997528022223:1.6435630703622064:-1.469387955399115
9:0.35702056413309086::0.6746001037299882:-1.7769037169718767:-0.9689138124473498

In [118]: pd.read_csv('tmp2.sv')
Out[118]:
   :0:1:2:3
0: 0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-0.1042359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.721551622443669:-0.7067711336300845:-1.0395749851146963:0.2718598854282986
4:-0.4249723297883753:0.56702349793672:0.27623201927711873:-1.08400691259915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.715002316146375:-0.139268835147725
7:-0.370468582364464:1.1578922506419993:-1.344311812631667:0.8448851414248841
8:0.757697831715533:-0.10904997528022223:1.6435630703622064:-1.469387955399115
9:0.35702056413309086::0.6746001037299882:-1.7769037169718767:-0.9689138124473498

[10 rows x 1 columns]

19.1.23 Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

In [119]: print(open('tmp.sv').read())
|0|1|2|3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934

pandas: powerful Python data analysis toolkit, Release 0.13.1

1|1.2121120250208506| -0.1732146905330858| 0.11920871129693428| -1.0442359662799567
2| -0.8618489634377999| 2.1045692188948086| -0.494292740687813| 1.0718038070373388
3| 0.7215551622443669| -0.7067711336300845| -1.0395749851146963| 0.27185988554282986
4| -0.4249723297883753| 0.567020349793672| 0.27623201927771873| -1.087406912859915
5| -0.6736897080883706| 0.1136484096888855| -1.4784265524372235| 0.5249876671147047
6| 0.4047052186802365| 0.5770458959204836| -1.715020161146375| -1.0392684835147725
7| -0.3706468582364464| 1.1578922506419993| -1.344311812731667| 0.844851414248841
8| 1.0757697837155533| -0.10904997528022223| 1.6435630703622064| -1.46938995399115
9| 0.35702056413309086| -0.6746001037299882| -1.776903716971867| -0.9689138124473498

In [120]: table = pd.read_table('tmp.sv', sep='|')

In [121]: table

Out[121]:

<table>
<thead>
<tr>
<th>Unnamed: 0 0 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0.469112 -0.282863 -1.509059 -1.135632</td>
</tr>
<tr>
<td>1 2 1.212112 -0.173215 0.119209 -1.044236</td>
</tr>
<tr>
<td>2 2 -0.861849 -2.104569 -0.494929 1.071804</td>
</tr>
<tr>
<td>3 3 0.721555 -0.706771 -1.039575 0.271860</td>
</tr>
<tr>
<td>4 4 -0.424972 0.567020 0.276232 -1.087401</td>
</tr>
<tr>
<td>5 5 -0.673690 0.113648 -1.478427 0.524988</td>
</tr>
<tr>
<td>6 6 0.404705 0.577046 -1.715002 -1.039268</td>
</tr>
<tr>
<td>7 7 -0.370647 -1.157892 -1.344312 0.844885</td>
</tr>
<tr>
<td>8 8 1.075770 -0.109050 1.643563 -1.469388</td>
</tr>
<tr>
<td>9 9 0.357021 -0.674600 -1.776904 -0.968914</td>
</tr>
</tbody>
</table>

[10 rows x 5 columns]

By specifying a chunksize to read_csv or read_table, the return value will be an iterable object of type TextFileReader:

In [122]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)

In [123]: reader

Out[123]: <pandas.io.parsers.TextFileReader at 0xb5de090>

In [124]: for chunk in reader:
   print(chunk)

.....:  print(chunk)
       Unnamed: 0 0 1 2 3
       [4 rows x 5 columns]

       Unnamed: 0 0 1 2 3
       [4 rows x 5 columns]

       Unnamed: 0 0 1 2 3
       [4 rows x 5 columns]
Specifying `iterator=True` will also return the `TextFileReader` object:

```python
In [125]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)
```

```python
In [126]: reader.get_chunk(5)
```

```
   Unnamed  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  0.276232 -1.087401
```

19.1.24 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`: A string path to the file to write
- `na_rep`: A string representation of a missing value (default ‘’)
- `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’
- `sep`: Field delimiter for the output file (default ”,”)
- `encoding`: a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3
- `tupleize_cols`: boolean, default False, if False, write as a list of tuples, otherwise write in an expanded line format suitable for `read_csv`

19.1.25 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
- `formaters` 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.

19.2 JSON

Read and write JSON format files and strings.

19.2.1 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 serialisable 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.

```python
In [127]: dfj = DataFrame(randn(5, 2), columns=list('AB'))
In [128]: json = dfj.to_json()
Out[129]: '{"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:

```python
In [130]: dfjo = DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
.....:     columns=list('ABC'), index=list('xyz'))
.....:
In [131]: dfjo
Out[131]:
   A  B  C
  x  1  4  7
  y  2  5  8
  z  3  6  9
[3 rows x 3 columns]
In [132]: sjo = Series(dict(x=15, y=16, z=17), name='D')
In [133]: sjo
Out[133]:
   x    y    z
Name: D, dtype: int64
```

**Column oriented** (the default for DataFrame) serialises the data as nested JSON objects with column labels acting as the primary index:

```python
In [134]: dfjo.to_json(orient="columns")
Out[134]: '{"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:

```python
In [135]: dfjo.to_json(orient="index")
In [136]: sjo.to_json(orient="index")
Out[136]: '{"x":15,"y":16,"z":17}''
```

**Record oriented** serialises the data to a JSON array of column -> value records, index labels are not included. This is useful for passing DataFrame data to plotting libraries, for example the JavaScript library d3.js:

```python
In [137]: dfjo.to_json(orient="records")
Out[137]: '[["A":1,"B":4,"C":7],"B":5,"C":8],{"A":3,"B":6,"C":9}]''
```

19.2. JSON 479
Value oriented is a bare-bones option which serialises to nested JSON arrays of values only, column and index labels are not included:

```python
In [139]: dfjo.to_json(orient="values")
Out[139]: '[[1,4,7],[2,5,8],[3,6,9]]'
```

Split oriented serialises to a JSON object containing separate entries for values, index and columns. Name is also included for Series:

```python
In [140]: dfjo.to_json(orient="split")
Out[140]: '{"columns":["A","B","C"],"index":"x","y","z"},"data":[[1,4,7],[2,5,8],[3,6,9]]'

In [141]: sjo.to_json(orient="split")
Out[141]: '{"name":"D","index":"x","y","z"},"data":[15,16,17]'
```

Note: Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialisation. If you wish to preserve label ordering use the split option as it uses ordered containers.

**Date Handling**

Writing in iso date format

```python
In [142]: dfd = DataFrame(randn(5, 2), columns=list('AB'))
In [143]: dfd['date'] = Timestamp('20130101')
In [144]: dfd = dfd.sort_index(1, ascending=False)
In [145]: json = dfd.to_json(date_format='iso')
In [146]: json
Out[146]: ...
```

Writing in iso date format, with microseconds

```python
In [147]: json = dfd.to_json(date_format='iso', date_unit='us')
In [148]: json
Out[148]: ...
```

Epoch timestamps, in seconds

```python
In [149]: json = dfd.to_json(date_format='epoch', date_unit='s')
In [150]: json
Out[150]: ...
```

Writing to a file, with a date index and a date column

```python
In [151]: dfj2 = dfj.copy()
In [152]: dfj2['date'] = Timestamp('20130101')
In [153]: dfj2['ints'] = list(range(5))
```
In [154]: dfj2[‘bools’] = True

In [155]: dfj2.index = date_range(‘20130101’, periods=5)

In [156]: dfj2.to_json(‘test.json’)

In [157]: open(‘test.json’).read()
Out [157]: ‘{"A":1356984000000,"1357084800000":0.2766617129,"1357171200000":-0.013955593999999999}’

**Fallback Behavior**

If the JSON serialiser 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 serialised.
- 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 serialisation 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: Maximum recursion level reached
```

which can be dealt with by specifying a simple `default_handler`:

In [158]: dftd.to_json(default_handler=str)
Out [158]: ‘{"0":"23 days, 0:00:00","1":0:00:05","2":42}’

In [159]: def my_handler(obj):
   ....:     return obj.total_seconds()
   ....:

**19.2.2 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`:  
  - `DataFrame`:  

pandas: powerful Python data analysis toolkit, Release 0.13.1

- default is index
- allowed values are {split, records, index}

**DataFrame**
- default is columns
- allowed values are {split, records, index, columns, values}

The format of the JSON string

<table>
<thead>
<tr>
<th>split</th>
<th>dict like {index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</th>
</tr>
</thead>
<tbody>
<tr>
<td>records</td>
<td>list like [[column -&gt; value], ..., [column -&gt; value]]</td>
</tr>
<tr>
<td>index</td>
<td>dict like {index -&gt; {column -&gt; value}}</td>
</tr>
<tr>
<td>columns</td>
<td>dict like {column -&gt; {index -&gt; value}}</td>
</tr>
<tr>
<td>values</td>
<td>just the values array</td>
</tr>
</tbody>
</table>

- **dtype**: if True, infer dtypes, if a dict of column to dtype, then use those, if False, then don’t infer dtypes at all, default is True, apply only to the data

- **convert_axes**: boolean, try to convert the axes to the proper dtypes, default is True

- **convert_dates**: a list of columns to parse for dates; If True, then try to parse datelike columns, default is True

- **keep_default_dates**: boolean, default True. If parsing dates, then parse the default datelike 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 parsable.

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 [160]: pd.read_json(json)
Out[160]:
   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

[5 rows x 3 columns]

Reading from a file:

In [161]: pd.read_json('test.json')
Out[161]:
   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

[5 rows x 5 columns]

Don’t convert any data (but still convert axes and dates):

In [162]: pd.read_json('test.json', dtype=object).dtypes
Out[162]:
A    object
B    object
bools  object
date   object
ints   object
dtype: object

Specify dtypes for conversion:

In [163]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[163]:
A       float32
B       float64
bools   int8
date    datetime64[ns]
ints    int64
dtype: object

Preserve string indicies:

In [164]: si = DataFrame(np.zeros((4, 4)),
         columns=list(range(4)),
         index=[str(i) for i in range(4)])

In [165]: si
Out[165]:
   0  1  2  3
0  0  0  0  0
1  0  0  0  0
In [166]: si.index
Out[166]: Index([u'0', u'1', u'2', u'3'], dtype='object')

In [167]: si.columns
Out[167]: Int64Index([0, 1, 2, 3], dtype='int64')

In [168]: json = si.to_json()

In [169]: sij = pd.read_json(json, convert_axes=False)

In [170]: sij
Out[170]:
   0   1   2   3
0  0   0   0   0
1  0   0   0   0
2  0   0   0   0
3  0   0   0   0

[4 rows x 4 columns]

In [171]: sij.index
Out[171]: Index([u'0', u'1', u'2', u'3'], dtype='object')

In [172]: sij.columns
Out[172]: Index([u'0', u'1', u'2', u'3'], dtype='object')

Dates written in nanoseconds need to be read back in nanoseconds:

In [173]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as milliseconds -> Won't Work

In [174]: dfju = pd.read_json(json, date_unit='ms')

In [175]: dfju
Out[175]:
            A      B     bools  date          ints
0  1356998400000000000  -1.294524  0.413738  2013-01-01   0
1  1357084800000000000   0.276662 -0.472035  2013-01-01   1
2  1357171200000000000  -0.013960 -0.362543  2013-01-01   2
3  1357257600000000000  -0.006154 -0.923061  2013-01-01   3
4  1357344000000000000   0.895717  0.805244  2013-01-01   4

[5 rows x 5 columns]

# Let Pandas detect the correct precision

In [176]: dfju = pd.read_json(json)

In [177]: dfju
Out[177]:
            A      B     bools  date          ints
0  1356998400000000000  -1.294524  0.413738  2013-01-01   0
1  1357084800000000000   0.276662 -0.472035  2013-01-01   1
2  1357171200000000000  -0.013960 -0.362543  2013-01-01   2
3  1357257600000000000  -0.006154 -0.923061  2013-01-01   3
4  1357344000000000000   0.895717  0.805244  2013-01-01   4
```python
2013-01-04 -0.006154 -0.923061 True 2013-01-01 3
2013-01-05 0.895717 0.805244 True 2013-01-01 4

[5 rows x 5 columns]

# Or specify that all timestamps are in nanoseconds
In [178]: dfju = pd.read_json(json, date_unit='ns')

In [179]: dfju
Out[179]:
   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

[5 rows x 5 columns]

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 deserialisation 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 [180]: randfloats = np.random.uniform(-100, 1000, 10000)

In [181]: randfloats.shape = (1000, 10)

In [182]: dffloats = DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [183]: jsonfloats = dffloats.to_json()

In [184]: timeit read_json(jsonfloats)
100 loops, best of 3: 12.1 ms per loop

In [185]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 6 ms per loop

The speedup is less noticable for smaller datasets:

In [186]: jsonfloats = dffloats.head(100).to_json()

In [187]: timeit read_json(jsonfloats)
100 loops, best of 3: 3.96 ms per loop

In [188]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 3.22 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.

19.2.3 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.

In [189]: from pandas.io.json import json_normalize

In [190]: 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 [191]: json_normalize(data, ‘counties’, [‘state’, ‘shortname’, [‘info’, ‘governor’]])

Out[191]:

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>info.governor</th>
<th>state</th>
<th>shortname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dade</td>
<td>12345</td>
<td>Rick Scott</td>
<td>Florida</td>
<td>FL</td>
</tr>
<tr>
<td>Broward</td>
<td>40000</td>
<td>Rick Scott</td>
<td>Florida</td>
<td>FL</td>
</tr>
<tr>
<td>Palm Beach</td>
<td>60000</td>
<td>Rick Scott</td>
<td>Florida</td>
<td>FL</td>
</tr>
<tr>
<td>Summit</td>
<td>1234</td>
<td>John Kasich</td>
<td>Ohio</td>
<td>OH</td>
</tr>
<tr>
<td>Cuyahoga</td>
<td>1337</td>
<td>John Kasich</td>
<td>Ohio</td>
<td>OH</td>
</tr>
</tbody>
</table>

[5 rows x 5 columns]

19.3 HTML

19.3.1 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 [192]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'

In [193]: dfs = read_html(url)

In [194]: dfs
Out[194]:

<table>
<thead>
<tr>
<th>Bank Name</th>
<th>City</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syringa Bank</td>
<td>Boise</td>
<td>ID</td>
</tr>
<tr>
<td>The Bank of Union</td>
<td>El Reno</td>
<td>OK</td>
</tr>
<tr>
<td>DuPage National Bank</td>
<td>West Chicago</td>
<td>IL</td>
</tr>
<tr>
<td>Texas Community Bank, National Association</td>
<td>The Woodlands</td>
<td>TX</td>
</tr>
<tr>
<td>Bank of Jackson County</td>
<td>Graceville</td>
<td>FL</td>
</tr>
<tr>
<td>First National Bank also operating as The Nat...</td>
<td>Edinburg</td>
<td>TX</td>
</tr>
<tr>
<td>The Community’s Bank</td>
<td>Bridgeport</td>
<td>CT</td>
</tr>
<tr>
<td>Sunrise Bank of Arizona</td>
<td>Phoenix</td>
<td>AZ</td>
</tr>
<tr>
<td>Community South Bank</td>
<td>Parsons</td>
<td>TN</td>
</tr>
<tr>
<td>Bank of Wausau</td>
<td>Wausau</td>
<td>WI</td>
</tr>
<tr>
<td>First Community Bank of Southwest Florida (als...</td>
<td>Fort Myers</td>
<td>FL</td>
</tr>
<tr>
<td>Mountain National Bank</td>
<td>Sevierville</td>
<td>TN</td>
</tr>
<tr>
<td>1st Commerce Bank</td>
<td>North Las Vegas</td>
<td>NV</td>
</tr>
<tr>
<td>Banks of Wisconsin d/b/a Bank of Kenosha</td>
<td>Kenosha</td>
<td>WI</td>
</tr>
<tr>
<td>Central Arizona Bank</td>
<td>Scottsdale</td>
<td>AZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CERT</th>
<th>Acquiring Institution</th>
<th>Closing Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sunwest Bank</td>
<td>2014-01-31</td>
</tr>
<tr>
<td>1</td>
<td>BancFirst</td>
<td>2014-01-24</td>
</tr>
<tr>
<td>2</td>
<td>Republic Bank of Chicago</td>
<td>2014-01-17</td>
</tr>
<tr>
<td>3</td>
<td>Spirit of Texas Bank, SSB</td>
<td>2013-12-13</td>
</tr>
<tr>
<td>4</td>
<td>First Federal Bank of Florida</td>
<td>2013-10-30</td>
</tr>
<tr>
<td>5</td>
<td>PlainsCapital Bank</td>
<td>2013-09-13</td>
</tr>
<tr>
<td>6</td>
<td>No Acquirer</td>
<td>2013-09-13</td>
</tr>
<tr>
<td>7</td>
<td>First Fidelity Bank, National Association</td>
<td>2013-08-23</td>
</tr>
<tr>
<td>8</td>
<td>CB&amp;S Bank, Inc.</td>
<td>2013-08-23</td>
</tr>
<tr>
<td>9</td>
<td>Nicolet National Bank</td>
<td>2013-08-09</td>
</tr>
<tr>
<td>10</td>
<td>C1 Bank</td>
<td>2013-08-02</td>
</tr>
<tr>
<td>11</td>
<td>First Tennessee Bank, National Association</td>
<td>2013-06-07</td>
</tr>
<tr>
<td>12</td>
<td>Plaza Bank</td>
<td>2013-06-06</td>
</tr>
<tr>
<td>13</td>
<td>North Shore Bank, FSB</td>
<td>2013-05-31</td>
</tr>
<tr>
<td>14</td>
<td>Western State Bank</td>
<td>2013-05-14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Updated Date</th>
<th>Loss</th>
<th>Share Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-01-31</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2014-01-28</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2014-01-27</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2014-01-13</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2013-12-09</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>2013-11-01</td>
<td>SFR/NSF</td>
<td></td>
</tr>
<tr>
<td>2013-12-20</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>2013-11-01</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>2013-11-01</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string

```
In [195]: with open(file_path, 'r') as f:
    .....
    dfs = read_html(f.read())
    .....

In [196]: dfs
Out[196]:

[ 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
7 Heritage Bank of North Florida Orange Park FL 26680
8 First Federal Bank Lexington KY 29594
9 Gold Canyon Bank Gold Canyon AZ 58066
10 Frontier Bank LaGrange GA 16431
11 Covenant Bank Chicago IL 22476
12 1st Regents Bank Andover MN 57157
13 Westside Community Bank University Place WA 33997
14 Community Bank of the Ozarks Sunrise Beach MO 27331
... ... ... ...

   Acquiring Institution Closing Date Updated Date
0 North Shore Bank, FSB 2013-05-31 2013-05-31
1 Western State Bank 2013-05-14 2013-05-20
2 Synovus Bank 2013-05-10 2013-05-21
3 Capital Bank, N.A. 2013-05-10 2013-05-14
4 Hamilton State Bank 2013-04-26 2013-05-16
5 CertusBank, National Association 2013-04-26 2013-05-17
7 FirstAtlantic Bank 2013-04-19 2013-05-16
8 Your Community Bank 2013-04-19 2013-04-23
9 First Scottsdale Bank, National Association 2013-04-05 2013-04-09
10 HeritageBank of the South 2013-03-08 2013-03-26
11 Liberty Bank and Trust Company 2013-02-15 2013-03-04
12 First Minnesota Bank 2013-01-18 2013-02-28
13 Sunwest Bank 2013-01-11 2013-01-24
14 Bank of Sullivan 2012-12-14 2013-01-24
... ... ... ...
```
You can even pass in an instance of `StringIO` if you so desire

```python
In [197]: from cStringIO import StringIO

In [198]: with open(file_path, 'r') as f:
   ....:     sio = StringIO(f.read())
   ....:

In [199]: dfs = read_html(sio)

In [200]: dfs
Out[200]:

<table>
<thead>
<tr>
<th>Bank Name</th>
<th>City</th>
<th>ST</th>
<th>CERT</th>
</tr>
</thead>
</table>
| Banks of Wisconsin d/b/a Bank of Kenosha | Kenosha      | WI | 35386
| Central Arizona Bank      | Scottsdale   | AZ | 34527
| Sunrise Bank              | Valdosta     | GA | 58185
| Pisgah Community Bank     | Asheville    | NC | 58701
| Douglas County Bank       | Douglasville | GA | 21649
| Parkway Bank              | Lenoir       | NC | 57158
| Chipola Community Bank    | Marianna     | FL | 58034
| Heritage Bank of North Florida | Orange Park | FL | 26680
| First Federal Bank        | Lexington    | KY | 29594
| Gold Canyon Bank          | Gold Canyon  | AZ | 58066
| Frontier Bank             | LaGrange     | GA | 16431
| Covenant Bank             | Chicago      | IL | 22476
| 1st Regents Bank          | Andover      | MN | 57157
| Westside Community Bank   | University Place | WA | 33997
| Community Bank of the Ozarks | Sunrise Beach | MO | 27331

[506 rows x 7 columns]
```

<table>
<thead>
<tr>
<th>Acquiring Institution</th>
<th>Closing Date</th>
<th>Updated Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>North Shore Bank, FSB</td>
<td>2013-05-31</td>
<td>2013-05-31</td>
</tr>
<tr>
<td>Western State Bank</td>
<td>2013-05-14</td>
<td>2013-05-20</td>
</tr>
<tr>
<td>Synovus Bank</td>
<td>2013-05-10</td>
<td>2013-05-21</td>
</tr>
<tr>
<td>Capital Bank, N.A.</td>
<td>2013-05-10</td>
<td>2013-05-14</td>
</tr>
<tr>
<td>Hamilton State Bank</td>
<td>2013-04-26</td>
<td>2013-05-16</td>
</tr>
<tr>
<td>CertusBank, National Association</td>
<td>2013-04-26</td>
<td>2013-05-17</td>
</tr>
<tr>
<td>FirstAtlantic Bank</td>
<td>2013-04-19</td>
<td>2013-05-16</td>
</tr>
<tr>
<td>Your Community Bank</td>
<td>2013-04-19</td>
<td>2013-04-23</td>
</tr>
<tr>
<td>First Scottsdale Bank, National Association</td>
<td>2013-04-05</td>
<td>2013-04-09</td>
</tr>
<tr>
<td>HeritageBank of the South</td>
<td>2013-03-08</td>
<td>2013-03-26</td>
</tr>
<tr>
<td>Liberty Bank and Trust Company</td>
<td>2013-02-15</td>
<td>2013-03-04</td>
</tr>
<tr>
<td>First Minnesota Bank</td>
<td>2013-01-18</td>
<td>2013-02-28</td>
</tr>
<tr>
<td>Sunwest Bank</td>
<td>2013-01-11</td>
<td>2013-01-24</td>
</tr>
<tr>
<td>Bank of Sullivan</td>
<td>2012-12-14</td>
<td>2013-01-24</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'])

19.3.2 Writing to HTML files

DataFrames 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 [201]: df = DataFrame(randn(2, 2))

In [202]: df
Out [202]:
   0       1
0 -0.184744  0.496971
1 -0.856240  1.857977
[2 rows x 2 columns]

In [203]: 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 [204]: 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 [205]:** `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>
<th>0</th>
<td>-0.1847438576</td>
<td>0.4969711327</td>
</tr>
<tr>
<th>1</th>
<td>-0.8562396763</td>
<td>1.8579766508</td>
</tr>
</tbody>
</table>
```

**HTML:**

`bold_rows` will make the row labels bold by default, but you can turn that off

**In [206]:** `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 [207]:** `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>
<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>
```
Finally, the `escape` argument allows you to control whether the “<”, “>” and “&” characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`.

```python
In [208]: df = DataFrame({'a': list('&<>'), 'b': randn(3)})
```

Escaped:

```python
In [209]: print(df.to_html())
```

```html
table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">  
<th></th>  
<th>a</th>  
<th>b</th>  
</tr>
</thead>
<tbody>
<tr>
<th>0</th>  
<td>&amp;</td>  
<td>-0.474063</td>  
</tr>
<tr>
<th>1</th>  
<td>&lt;</td>  
<td>-0.230305</td>  
</tr>
<tr>
<th>2</th>  
<td>&gt;</td>  
<td>-0.400654</td>  
</tr>
</tbody>
</table>
```

Not escaped:

```python
In [210]: print(df.to_html(escape=False))
```

```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>-0.184744</td>  
<td> 0.496971</td>  
</tr>
<tr>
<th>1</th>  
<td>&lt; </td>  
<td>-0.856240</td>  
<td> 1.857977</td>  
</tr>
</tbody>
</table>
```
19.4 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.

Besides `read_excel` you can also read Excel files using the `ExcelFile` class. The following two command are equivalent:

```python
# using the ExcelFile class
xls = pd.ExcelFile(‘path_to_file.xls’)  
xls.parse(‘Sheet1’, index_col=None, na_values=[‘NA’])

# using the read_excel function
read_excel(‘path_to_file.xls’, ‘Sheet1’, index_col=None, na_values=[‘NA’])
```

The class based approach can be used to read multiple sheets or to introspect the sheet names using the `sheet_names` attribute.

Note: The prior method of accessing `ExcelFile` has been moved from `pandas.io.parsers` to the top level namespace starting from pandas 0.12.0.

New in version 0.13. There are now two ways to read in sheets from an Excel file. You can provide either the index of a sheet or its name. If the value provided is an integer then it is assumed that the integer refers to the index of a sheet, otherwise if a string is passed then it is assumed that the string refers to the name of a particular sheet in the file.
Using the sheet name:

```python
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```python
read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

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.

```python
read_excel('path_to_file.xls', 'Sheet1', parse_cols=2, index_col=None, na_values=['NA'])
```

If `parse_cols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```python
read_excel('path_to_file.xls', 'Sheet1', parse_cols=[0, 2, 3], index_col=None, na_values=['NA'])
```

To write a DataFrame object to a sheet of an Excel file, you can use the `to_excel` instance method. The arguments are largely the same as `to_csv` described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the DataFrame should be written. For example:

```python
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Files with a `.xls` extension will be written using `xlwt` and those with a `.xlsx` extension will be written using `xlsxwriter` (if available) or `openpyxl`.

The 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.

### 19.4.1 Excel writer engines

New in version 0.13. pandas chooses an Excel writer via two methods:

1. the `engine` keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the `XlsxWriter` for `.xlsx` and `openpyxl` for `.xlsm` files and `xlwt` for `.xls` files. If you have multiple engines installed, you can set the default engine through `setting the config options` `io.excel.xlsx.writer` and `io.excel.xls.writer`. pandas will fall back on `openpyxl` for `.xlsx` files if `Xlsxwriter` is not available.

To specify which writer you want to use, you can pass an engine keyword argument to `to_excel` and to `ExcelWriter`.  

---

19.4. Excel files 495
# 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')

19.5 Clipboard

A handy way to grab data is to use the `read_clipboard` method, which takes the contents of the clipboard buffer and passes them to the `read_table` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a DataFrame by calling:

```python
clipdf = pd.read_clipboard()
```

In [211]: clipdf
Out[211]:
    A  B  C
   x 1  4  p
   y 2  5  q
   z 3  6  r

[3 rows x 3 columns]

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.

In [212]: df = pd.DataFrame(randn(5, 3))

In [213]: df
Out[213]:
   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

[5 rows x 3 columns]

In [214]: df.to_clipboard()

In [215]: pd.read_clipboard()
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.

### 19.6 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.

```
In [216]: df
Out[216]:
   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
```

```
In [217]: df.to_pickle('foo.pkl')
```

The `read_pickle` function in the pandas namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [218]: read_pickle('foo.pkl')
Out[218]:
   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:** In 0.13, pickle preserves compatibility with pickles created prior to 0.13. These must be read with `pd.read_pickle`, rather than the default python `pickle.load`. See [this question](http://docs.python.org/2.7/library/pickle.html) for a detailed explanation.

**Note:** These methods were previously `pd.save` and `pd.load`, prior to 0.12.0, and are now deprecated.
19.7 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 [219]: df = DataFrame(np.random.rand(5,2),columns=list('AB'))
In [220]: df.to_msgpack('foo.msg')
In [221]: pd.read_msgpack('foo.msg')
Out[221]:
   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

[5 rows x 2 columns]
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [222]: s = Series(np.random.rand(5),index=date_range('20130101',periods=5))
In [223]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1,2,3]), s)
In [224]: pd.read_msgpack('foo.msg')
Out[224]:
[ 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

[5 rows x 2 columns], u'foo', array([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 `iterator=True` to iterate over the unpacked results

```
In [225]: for o in pd.read_msgpack('foo.msg',iterator=True):
.....:     print o
.....:
   A    B
0 0.154336 0.710999
```

498 Chapter 19. IO Tools (Text, CSV, HDF5, ...
1 0.398096 0.765220
2 0.586749 0.293052
3 0.290293 0.710783
4 0.988593 0.062106

[5 rows x 2 columns]

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 [226]: df.to_msgpack('foo.msg', append=True)

In [227]: pd.read_msgpack('foo.msg')
Out[227]:
   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

[5 rows x 2 columns], u'foo', array([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

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 [228]: pd.to_msgpack('foo2.msg', { 'dict' : [ { 'df' : df }, { 'string' : 'foo' }, { 'scalar' : 1. }, { 's' : s } ] })

In [229]: pd.read_msgpack('foo2.msg')
Out[229]:

{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

[5 rows x 2 columns]},
{u'string': u'foo'},
{u'scalar': 1.0},
{u's': u'19.7. msgpack (experimental) 499'}}
19.7.1 Read/Write API

Msgpacks can also be read from and written to strings.

In [230]: df.to_msgpack()
Out[230]:
'b8a6blocks\xa5item\xa5dt\xa5\xa5index\xa5klass\xa5Index\xa5data'

Furthermore you can concatenate the strings to produce a list of the original objects.

In [231]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[231]:
[ 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
[5 rows x 2 columns], 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]

19.8 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

Note: PyTables 3.0.0 was recently released to enable support for Python 3. Pandas should be fully compatible (and previously written stores should be backwards compatible) with all PyTables >= 2.3. For python >= 3.2, pandas >= 0.12.0 is required for compatibility.

In [232]: store = HDFStore('store.h5')

In [233]: 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 [234]: np.random.seed(1234)
pandas: powerful Python data analysis toolkit, Release 0.13.1

In [235]: index = date_range('1/1/2000', periods=8)

In [236]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [237]: df = DataFrame(randn(8, 3), index=index,
   .....: columns=['A', 'B', 'C'])
   .....:

In [238]: 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 [239]: store['s'] = s

In [240]: store['df'] = df

In [241]: store['wp'] = wp

# the type of stored data
In [242]: store.root.wp._v_attrs.pandas_type
Out[242]: 'wide'

In [243]: store
Out[243]:
<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 [244]: store['df']
Out[244]:
     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
19.8. HDF5 (PyTables)
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

[8 rows x 3 columns]

Deletion of the object specified by the key

```python
# store.remove('wp') is an equivalent method
In [246]: del store['wp']
```

```python
In [247]: store
Out[247]:<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 [248]: store.close()
In [249]: store
Out[249]:<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
File is CLOSED
```

```python
In [250]: store.is_open
Out[250]: False
```

# Working with, and automatically closing the store with the context manager
```python
In [251]: with get_store('store.h5') as store:
.....:    store.keys()
.....:
```

19.8.1 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 [252]: df_tl = DataFrame(dict(A=list(range(5)), B=list(range(5))))
In [253]: df_tl.to_hdf('store_tl.h5','table',append=True)
In [254]: read_hdf('store_tl.h5', 'table', where = ['index>2'])
Out[254]:
       A  B
0    3  3
1    4  4
[2 rows x 2 columns]
```

19.8.2 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. These 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')
```

TypeError: cannot pass a where specification when reading a fixed format. this store must be selected in its entirety

19.8.3 Table Format

HDFStore supports another PyTables format on disk, the table format. Conceptually a table is shaped very much like a DataFrame, with rows and columns. A table may be appended to in the same or other sessions. In addition, delete & query type operations are supported. This format is specified by format='table' or format='t' to append or put or to_hdf.

This format can be set as an option as well pd.set_option('io.hdf.default_format','table') to enable put/append/to_hdf to by default store in the table format.

```python
In [255]: store = HDFStore('store.h5')

In [256]: df1 = df[0:4]

In [257]: df2 = df[4:]

# append data (creates a table automatically)
In [258]: store.append('df', df1)

In [259]: store.append('df', df2)

In [260]: store
Out[260]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df         frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])

# select the entire object
In [261]: store.select('df')
Out[261]:
    A      B      C
0 0.887163 0.859588 -0.636524
1 0.015696 -2.242685  1.150036
2 0.991946  0.953324 -2.021255
3 -0.334077  0.002118  0.405453
4  0.289092  1.321158 -1.546906
5  0.202646 -0.655969  0.193421
6  0.553439  1.318152 -0.469305
7  0.675554 -1.817027 -0.183109

19.8. HDF5 (PyTables)  503
# the type of stored data
In [262]: store.root.df._v_attrs.pandas_type
Out[262]: 'frame_table'

Note: You can also create a table by passing format='table' or format='t' to a put operation.

## 19.8.4 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 [263]: store.put('foo/bar/bah', df)
In [264]: store.append('food/orange', df)
In [265]: store.append('food/apple', df)
In [266]: store
Out[266]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/apple frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])

# a list of keys are returned
In [267]: store.keys()
Out[267]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [268]: store.remove('food')
In [269]: store
Out[269]: <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])

## 19.8.5 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 [270]: 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 [271]: df_mixed.ix[3:5, ['A', 'B', 'string', 'datetime64']] = np.nan

In [272]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})

In [273]: df_mixed1 = store.select('df_mixed')

In [274]: df_mixed1
Out[274]:
     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           NaN  True   NaN      1   NaN
4         NaN   NaN           NaN  True   NaN      1   NaN
5         NaN   NaN           NaN  True   NaN      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

[8 rows x 7 columns]

In [275]: df_mixed1.get_dtype_counts()
Out[275]:
bool    1
datetime64[ns]    1
float32         2
int64           1
object          1
dtype: int64

# we have provided a minimum string column size

In [276]: store.root.df_mixed.table
Out[276]:
/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}
19.8.6 Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.

```
In [277]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
                           ['one', 'two', 'three']],
                           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
                                   [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
                           names=['foo', 'bar'])
                   
In [278]: df_mi = DataFrame(np.random.randn(10, 3), index=index,
                           columns=['A', 'B', 'C'])
                   
In [279]: df_mi
Out[279]:
          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  0.680566
    two -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

[10 rows x 3 columns]
```

```
In [280]: store.append('df_mi',df_mi)

In [281]: store.select('df_mi')
Out[281]:
          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  0.680566
    two -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

[10 rows x 3 columns]
```

```
# the levels are automatically included as data columns
In [282]: store.select('df_mi', 'foo=bar')
Out[282]:
          A     B     C
    foo bar
    one   -0.584718 0.816594 -0.081947
    two   -0.344766 0.528288 -1.068989
    three -0.511881 0.291205  0.566534
    bar one  0.503592 0.285296  0.484288
    two  1.363482 -0.781105  0.468018
    baz two  1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
    qux one -0.203933 -0.182175  0.680566
    two -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

[10 rows x 3 columns]
```
19.8.7 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 it's 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

Here are some examples:

In [283]: dfq = DataFrame(randn(10,4),columns=list('ABCD'),index=date_range('20130101',periods=10))

In [284]: store.append('dfq',dfq,format='table',data_columns=True)

Use boolean expressions, with in-line function evaluation.

In [285]: store.select('dfq','index>Timestamp('20130104') & columns=['A', 'B'])

Out[285]:
      A       B
2013-01-05  1.21  0.797
2013-01-06 -0.85  1.176
2013-01-07  0.98 -1.21
2013-01-08  0.79 -0.47
2013-01-09 -0.80  2.12
2013-01-10  0.33  0.53

[6 rows x 2 columns]

Use and inline column reference

In [286]: store.select('dfq',where="A>0 or C>0")

Out[286]:
      A       B       C       D
2013-01-01  0.43 -1.70  0.39 -0.48
2013-01-02 -0.29  0.69  0.76  0.24
2013-01-03  0.15  0.81  1.99  0.64
2013-01-04 -0.96  2.08  1.93 -1.73
2013-01-05  1.21  0.79 -0.38  0.70
2013-01-07  0.98 -0.12  2.36  0.49
2013-01-08  0.79 -0.47 -0.06  1.36
2013-01-10  0.33  0.53 -0.74 -0.32

[8 rows x 4 columns]

Works with a Panel as well.

In [287]: store.append('wp',wp)

In [288]: store

Out[288]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df_mi frame_table (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->[bar,foo])
/df_mixed frame_table (typ->appendable,nrows->8,ncols->7,indexers->[index])
/dfq frame_table (typ->appendable,nrows->10,ncols->4,indexers->[index],dc->[A,B,C,D])
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':

```python
In [290]: store.select('df', "columns=['A', 'B']")
```

```
<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>0.288092</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>
```

[8 rows x 2 columns]

`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 [291]: wp.to_frame()
```

```
<table>
<thead>
<tr>
<th>major</th>
<th>minor</th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>A</td>
<td>1.058969</td>
<td>0.215269</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.397840</td>
<td>0.841009</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.337438</td>
<td>-1.445810</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>1.047579</td>
<td>-1.401973</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>A</td>
<td>1.045938</td>
<td>-0.100918</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.863717</td>
<td>-0.548242</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.122092</td>
<td>-0.144620</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.124713</td>
<td>0.354020</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>A</td>
<td>-0.322795</td>
<td>-0.035513</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.841675</td>
<td>0.565738</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2.390961</td>
<td>1.545659</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.076200</td>
<td>-0.974236</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>A</td>
<td>-0.566446</td>
<td>-0.070345</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.036142</td>
<td>0.307969</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-2.074978</td>
<td>-0.208499</td>
</tr>
</tbody>
</table>
```

[20 rows x 2 columns]

```
# limiting the search
In [292]: store.select('wp','major_axis>20000102 & minor_axis=['A','B']", 

<table>
<thead>
<tr>
<th>major</th>
<th>minor</th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>A</td>
<td>1.058969</td>
<td>0.215269</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.397840</td>
<td>0.841009</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.337438</td>
<td>-1.445810</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>1.047579</td>
<td>-1.401973</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>A</td>
<td>1.045938</td>
<td>-0.100918</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.863717</td>
<td>-0.548242</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.122092</td>
<td>-0.144620</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.124713</td>
<td>0.354020</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>A</td>
<td>-0.322795</td>
<td>-0.035513</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.841675</td>
<td>0.565738</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2.390961</td>
<td>1.545659</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0.076200</td>
<td>-0.974236</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>A</td>
<td>-0.566446</td>
<td>-0.070345</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0.036142</td>
<td>0.307969</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-2.074978</td>
<td>-0.208499</td>
</tr>
</tbody>
</table>
```

[20 rows x 2 columns]
Out[292]:
<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 [293]: from datetime import timedelta

In [294]: dftd = DataFrame(dict(A = Timestamp('20130101'), B = [ Timestamp('20130101') + timedelta(days=i,seconds=10) for i in range(10) ]))

In [295]: dftd['C'] = dftd['A']-dftd['B']

In [296]: dftd

Out[296]:
   A                  B                  C
0 2013-01-01 00:00:10 -0 days 00:00:10
1 2013-01-01 00:00:10 -1 days 00:00:10
2 2013-01-01 00:00:10 -2 days 00:00:10
3 2013-01-01 00:00:10 -3 days 00:00:10
4 2013-01-01 00:00:10 -4 days 00:00:10
5 2013-01-01 00:00:10 -5 days 00:00:10
6 2013-01-01 00:00:10 -6 days 00:00:10
7 2013-01-01 00:00:10 -7 days 00:00:10
8 2013-01-01 00:00:10 -8 days 00:00:10
9 2013-01-01 00:00:10 -9 days 00:00:10

[10 rows x 3 columns]

In [297]: store.append('dftd',dftd,data_columns=True)

In [298]: store.select('dftd','C<3.5D')

Out[298]:
   A                  B                  C
4 2013-01-01 00:00:10 -4 days 00:00:10
5 2013-01-01 00:00:10 -5 days 00:00:10
6 2013-01-01 00:00:10 -6 days 00:00:10
7 2013-01-01 00:00:10 -7 days 00:00:10
8 2013-01-01 00:00:10 -8 days 00:00:10
9 2013-01-01 00:00:10 -9 days 00:00:10

[6 rows x 3 columns]
19.8.8 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`.

```
# we have automagically already created an index (in the first section)
In [299]: i = store.root.df.table.cols.index.index

In [300]: i.optlevel, i.kind
Out[300]: (6, 'medium')

# change an index by passing new parameters
In [301]: store.create_table_index('df', optlevel=9, kind='full')

In [302]: i = store.root.df.table.cols.index.index

In [303]: i.optlevel, i.kind
Out[303]: (9, 'full')
```

See here for how to create a completely-sorted-index (CSI) on an existing store.

19.8.9 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`

```
In [304]: df_dc = df.copy()

In [305]: df_dc['string'] = 'foo'

In [306]: df_dc.ix[4:6,'string'] = np.nan

In [307]: df_dc.ix[7:9,'string'] = 'bar'

In [308]: df_dc['string2'] = 'cool'

In [309]: df_dc.ix[1:3,['B','C']] = 1.0

In [310]: df_dc
Out[310]:
A   B   C  string  string2
0  2000-01-01 0.887163 0.859588 0.363524 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 0.0334077 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
```
[8 rows x 5 columns]

# on-disk operations

In [311]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])

In [312]: store.select('df_dc', [ Term('B>0') ])

Out[312]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>string</td>
<td>string2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>--------</td>
<td>--------</td>
</tr>
<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>

[6 rows x 5 columns]

# getting creative

In [313]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')

Out[313]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>string</td>
<td>string2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>--------</td>
<td>--------</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>
</tbody>
</table>

[3 rows x 5 columns]

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

In [314]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]

Out[314]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>string</td>
<td>string2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>--------</td>
<td>--------</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>
</tbody>
</table>

[3 rows x 5 columns]

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as "PyTables" columns

In [315]: store.root.df_dc.table

Out[315]:
<table>
<thead>
<tr>
<th>df_dc/table (Table(8,))</th>
<th>''</th>
</tr>
</thead>
<tbody>
<tr>
<td>description :=</td>
<td></td>
</tr>
<tr>
<td>&quot;index&quot;: Int64Col(shape=(), dflt=0, pos=0),</td>
<td></td>
</tr>
<tr>
<td>&quot;values_block_0&quot;: Float64Col(shape=(1,), dflt=0.0, pos=1),</td>
<td></td>
</tr>
<tr>
<td>&quot;B&quot;: Float64Col(shape=(), dflt=0.0, pos=2),</td>
<td></td>
</tr>
<tr>
<td>&quot;C&quot;: Float64Col(shape=(), dflt=0.0, pos=3),</td>
<td></td>
</tr>
<tr>
<td>&quot;string&quot;: StringCol(itemsize=3, shape=(), dflt='', pos=4),</td>
<td></td>
</tr>
<tr>
<td>&quot;string2&quot;: StringCol(itemsize=4, shape=(), dflt='', pos=5)</td>
<td></td>
</tr>
<tr>
<td>byteorder := 'little'</td>
<td></td>
</tr>
<tr>
<td>chunkshape := (1680,)</td>
<td></td>
</tr>
<tr>
<td>autoindex := True</td>
<td></td>
</tr>
</tbody>
</table>
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!)

### 19.8.10 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.

```python
In [316]: 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

[3 rows x 3 columns]
```

```python
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

[3 rows x 3 columns]
```

```python
A   B   C
2000-01-07 0.553439 1.318152 -0.469305
2000-01-08 0.675554 -1.817027 -0.183109

[2 rows x 3 columns]
```

**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.

```python
for df in read_hdf('store.h5','df', chunsize=3):
    print(df)
```

Note, that the chunksize keyword applies to the source rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```python
In [317]: dfeq = DataFrame({'number': np.arange(1,11)})
```

```python
In [318]: dfeq
Out[318]:
    number
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10
```

19.8. HDF5 (PyTables)
In [319]: store.append('dfeq', dfeq, data_columns=['number'])

In [320]: def chunks(l, n):
    ....:     return [l[i:i+n] for i in xrange(0, len(l), n)]
    ....:

In [321]: evens = [2,4,6,8,10]

In [322]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [323]: for c in chunks(coordinates, 2):
    ....:     print store.select('dfeq', where=c)
    ....:     number
    1  2
    3  4

[2 rows x 1 columns]
number
5  6
7  8

[2 rows x 1 columns]
number
9  10

[1 rows x 1 columns]

19.8.11 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 [324]: store.select_column('df_dc', 'index')
Out[324]:
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 [325]: store.select_column('df_dc', 'string')
Out[325]:
0    foo
1    foo
2    NaN
3    foo
4    NaN
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 [326]: df_coord = DataFrame(np.random.randn(1000,2),index=date_range('20000101',periods=1000))

In [327]: store.append('df_coord',df_coord)

In [328]: c = store.select_as_coordinates('df_coord','index>20020101')

In [329]: c.summary()
Out[329]: u'Int64Index: 268 entries, 732 to 999'

In [330]: store.select('df_coord',where=c)

Out[330]:
      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
2002-01-09 -0.383978  1.209025
2002-01-10  0.213923 -0.113980
2002-01-11  0.944945 -0.183393
2002-01-12  1.714323  0.024600
2002-01-13  0.454133  0.272278
2002-01-14  0.305823 -0.390413
2002-01-15  0.424165  0.208513
2002-01-16  0.429386  1.357697
      ...    ...
[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 [331]: df_mask = DataFrame(np.random.randn(1000,2),index=date_range('20000101',periods=1000))

In [332]: store.append('df_mask',df_mask)

In [333]: c = store.select_column('df_mask','index')

In [334]: where = c[DatetimeIndex(c).month==5].index

In [335]: store.select('df_mask',where=where)
Out[335]:
      0     1
2000-05-01 -0.098554 -0.280782
2000-05-02  0.739851  1.627182
Storer Object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [336]: store.get_storer('df_dc').nrows
Out[336]: 8
```

19.8.12 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.

```
In [337]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
                   columns=['A', 'B', 'C', 'D', 'E', 'F'])
```

```
In [338]: df_mt['foo'] = 'bar'
```

```
In [339]: df_mt.ix[1, ('A', 'B')] = np.nan
```

# you can also create the tables individually
```
In [340]: store.append_to_multiple([{'df1_mt': ['A', 'B'], 'df2_mt': None} ],
               df_mt, selector='df1_mt')
```
In [341]: store
Out[341]:
<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->[bar,foo])
/df_mixed frame_table (typ->appendable,nrows->8,ncols->7,indexers->[index])
/df_eq 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])
/dftd frame_table (typ->appendable,nrows->10,ncols->3,indexers->[index],dc->[A,B,C])
/wp   wide_table (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah frame (shape->[8,3])

# individual tables were created
In [342]: store.select('df1_mt')
Out[342]:
          A     B
2000-01-01 -0.816310 1.282296
2000-01-03  0.684353 -1.755306
2000-01-04 -1.315814  1.455079
2000-01-05 -0.027564  0.046757
2000-01-06 -0.416244 -0.821168
2000-01-07  0.665090  1.084344
2000-01-08  0.607460  0.790907

[7 rows x 2 columns]

In [343]: store.select('df2_mt')
Out[343]:
          C     D     E     F    foo
2000-01-01 -1.521825 -0.428670 -1.550209  0.826839   bar
2000-01-03  0.684353 -1.755306  1.455079  1.894976   bar
2000-01-04 -1.315814  1.455079 -0.027564  0.467577   bar
2000-01-05 -0.416244 -0.821168  0.665090  1.084344   bar
2000-01-06 -1.945328  2.115021  0.148762  1.073931   bar
2000-01-07  0.709897 -2.022441  0.714697  0.318215   bar
2000-01-08  0.852225  0.966963 -0.379903  0.929313   bar

[7 rows x 5 columns]

# as a multiple
In [344]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
....:               selector = 'df1_mt')
....:
Out[344]:
          A     B     C     D     E     F    foo
2000-01-07  0.66509  1.084344 -0.709897 -2.022441  0.714697  0.318215   bar
2000-01-08  0.60746  0.790907  0.852225  0.966963 -0.379903  0.929313   bar

[2 rows x 7 columns]
19.8.13 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 [345]: store.remove('wp', 'major_axis>20000102')
Out[345]: 12
In [346]: store.select('wp')
Out[346]:
<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).

19.8.14 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

- \texttt{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 \texttt{complevel=0}

- \texttt{store.append('df', df, complib='zlib', complevel=5)}

\texttt{ptrepack}

\texttt{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 \texttt{PyTables} utility \texttt{ptrepack}. In addition, \texttt{ptrepack} can change compression levels after the fact.

- \texttt{ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5}

Furthermore \texttt{ptrepack in.h5 out.h5} will \texttt{repack} the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the \texttt{copy} method.

### 19.8.15 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 \texttt{np.nan} for EVERY COLUMN (having a \texttt{nan} in a string, or a \texttt{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.

- \texttt{HDFStore} is not-threadsafe for writing. The underlying \texttt{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 \texttt{fsync()} before releasing write locks. For convenience you can use \texttt{store.flush(fsync=True)} to do this for you.

- \texttt{PyTables} only supports fixed-width string columns in tables. The sizes of a string based indexing column (e.g. \texttt{columns} or \texttt{minor\_axis}) are determined as the maximum size of the elements in that axis or by passing the parameter

  \begin{verbatim}
  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.
  \end{verbatim}

### 19.8.16 DataTypes

\texttt{HDFStore} will map an object dtype to the \texttt{PyTables} underlying dtype. This means the following types are known to work:

- \texttt{floating : float64, float32, float16} (using \texttt{np.nan} to represent invalid values)

- \texttt{integer : int64, int32, int8, uint64, uint32, uint8}

- \texttt{bool}

- \texttt{datetime64[ns]} (using \texttt{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)

```
In [347]: import datetime

In [348]: df = DataFrame(dict(datelike=Series([datetime.datetime(2001, 1, 1),
                                             ....:
                                             datetime.datetime(2001, 1, 2), np.nan]))

In [349]: df
Out[349]:
          datelike
0 2001-01-01
1 2001-01-02
2 NaT
[3 rows x 1 columns]

In [350]: df.dtypes
Out[350]:
datelike    datetime64[ns]
dtype: object

# to convert
In [351]: df[‘datelike’] = Series(df[‘datelike’].values, dtype=’M8[ns]’)

In [352]: df
Out[352]:
          datelike
0 2001-01-01
1 2001-01-02
2 NaT
[3 rows x 1 columns]

In [353]: df.dtypes
Out[353]:
datelike    datetime64[ns]
dtype: object
```

### 19.8.17 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 [354]: dfs = DataFrame(dict(A = 'foo', B = 'bar'),index=list(range(5)))

In [355]: dfs
Out[355]:
   A   B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
4  foo  bar

[5 rows x 2 columns]

# A and B have a size of 30
In [356]: store.append('dfs', dfs, min_itemsize = 30)

In [357]: store.get_storer('dfs').table
Out[357]:
/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 size is calculated
In [358]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })

In [359]: store.get_storer('dfs2').table
Out[359]:
/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 inadvertantly turn an actual `nan` value into a missing value.
In [360]: dfss = DataFrame(dict(A = ['foo','bar','nan']))

In [361]: dfss
Out[361]:
   A
0  foo
1  bar
2  nan
[3 rows x 1 columns]

In [362]: store.append('dfss', dfss)

In [363]: store.select('dfss')
Out[363]:
   A
0  foo
1  bar
2  NaN
[3 rows x 1 columns]

# here you need to specify a different nan rep
In [364]: store.append('dfss2', dfss, nan_rep='_nan_')

In [365]: store.select('dfss2')
Out[365]:
   A
0  foo
1  bar
2  nan
[3 rows x 1 columns]

19.8.18 External Compatibility

HDFStore write table format objects in specific formats suitable for producing loss-less roundtrips 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. Create a table format store like this:

In [366]: store_export = HDFStore('export.h5')

In [367]: store_export.append('df_dc', df_dc, data_columns=df_dc.columns)

In [368]: store_export
Out[368]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
/df_dc frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[A,B,C,string,string2])

19.8.19 Backwards Compatibility

0.10.1 of HDFStore can read tables created in a prior version of pandas, however query terms using the prior (un-documented) 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
# a legacy store
In [369]: legacy_store = HDFStore(legacy_file_path,'r')

In [370]: legacy_store
Out[370]:
<class 'pandas.io.pytables.HDFStore'>
File path: /home/user1/src/pandas/doc/source/_static/legacy_0.10.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/dfl Mixed frame_table [0.10.0] (typ->appendable,nrows->30,ncols->11,indexers->[index])
/pl Mixed wide_table [0.10.0] (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,minor_axis])
/p4d Mixed wide_table [0.10.0] (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis,minor_axis])
/.. Wide (shape->[3,30,4])

# copy (and return the new handle)
In [371]: new_store = legacy_store.copy('store_new.h5')

In [372]: new_store
Out[372]:
<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/dfl Mixed frame_table (typ->appendable,nrows->30,ncols->11,indexers->[index])
/pl Mixed wide_table (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,minor_axis])
/p4d Mixed wide_table (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis,minor_axis])
/.. Wide (shape->[3,30,4])

In [373]: new_store.close()

19.8.20 Performance

• **Tables** come with a writing performance penalty as compared to regular 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.

19.8.21 Experimental

HDFStore supports Panel4D storage.

In [374]: p4d = Panel4D({ '11' : wp })
In [375]: `p4d`  
Out[375]:  
<class 'pandas.core.panel.Panel'>  
Dimensions: 1 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)  
Labels axis: 11 to 11  
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 [376]: `store.append('p4d', p4d)`  

In [377]: `store`  
Out[377]:  
<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->2,indexers->[index])  
/df_dc frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,str])  
/df_mask frame_table (typ->appendable,nrows->1000,ncols->2,indexers->[index])  
/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->2,indexers->[index])  
/dfs2 frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])  
/dffs frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index])  
/dffs2 frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index])  
/dftd frame_table (typ->appendable,nrows->10,ncols->3,indexers->[index])  
/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,minor_axis])  
/foo/bar/bah frame (shape->[8,3])

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 [378]: `store.append('p4d2', p4d, axes=['labels', 'major_axis', 'minor_axis'])`  

In [379]: `store`  
Out[379]:  
<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->2,indexers->[index])  
/df_dc frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,str])  
/df_mask frame_table (typ->appendable,nrows->1000,ncols->2,indexers->[index])  
/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->2,indexers->[index])  
/dfs2 frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])  
/dffs frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index])  
/dftd frame_table (typ->appendable,nrows->10,ncols->3,indexers->[index])  
/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,minor_axis])  
/foo/bar/bah frame (shape->[8,3])
In [380]: store.select('p4d2', [Term('labels=l1'), Term('items=Item1'), Term('minor_axis=A_big_strings')])
Out[380]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 0 (labels) x 1 (items) x 0 (major_axis) x 0 (minor_axis)
Labels axis: None
Items axis: Item1 to Item1
Major_axis axis: None
Minor_axis axis: None

19.9 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. These wrappers only support the Python database adapters which respect the Python DB-API. See some cookbook examples for some advanced strategies

For example, suppose you want to query some data with different types from a table such as:

<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>

Functions from pandas.io.sql can extract some data into a DataFrame. In the following example, we use the SQLite SQL database engine. You can use a temporary SQLite database where data are stored in "memory". Just do:

```python
import sqlite3
from pandas.io import sql
# Create your connection.
cnx = sqlite3.connect(':memory:')

# Let data be the name of your SQL table. With a query and your database connection, just use the read_sql() function to get the query results into a DataFrame:

In [381]: sql.read_sql("SELECT * FROM data;", cnx)
Out[381]:

<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>2010-10-18</td>
<td>X</td>
<td>27.50</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2010-10-19</td>
<td>Y</td>
<td>-12.50</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>2010-10-20</td>
<td>Z</td>
<td>5.73</td>
<td>1</td>
</tr>
</tbody>
</table>
[3 rows x 5 columns]
```

You can also specify the name of the column as the DataFrame index:

```python
In [382]: sql.read_sql("SELECT * FROM data;", cnx, index_col='id')
Out[382]:

<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>2010-10-18</td>
<td>X</td>
<td>27.50</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2010-10-19</td>
<td>Y</td>
<td>-12.50</td>
<td>0</td>
</tr>
</tbody>
</table>
```
63  2010-10-20  00:00:00  Z  5.73  1

[3 rows x 4 columns]

In [383]: sql.read_sql("SELECT * FROM data;", cnx, index_col='date')
Out[383]:
   id  Col_1  Col_2  Col_3
date
2010-10-18 00:00:00   26  X  27.50  1
2010-10-19 00:00:00   42  Y -12.50  0
2010-10-20 00:00:00   63  Z  5.73  1

[3 rows x 4 columns]

Of course, you can specify a more “complex” query.

In [384]: sql.read_sql("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", cnx)
Out[384]:
   id  Col_1  Col_2
0   42  Y  -12.5

[1 rows x 3 columns]

There are a few other available functions:

- tquery returns a list of tuples corresponding to each row.
- uquery does the same thing as tquery, but instead of returning results it returns the number of related rows.
- write_frame writes records stored in a DataFrame into the SQL table.
- has_table checks if a given SQLite table exists.

Note: For now, writing your DataFrame into a database works only with SQLite. Moreover, the index will currently be dropped.

19.10 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 derived from the source table. Additionally, DataFrames can be uploaded into BigQuery datasets as tables if the source datatypes are compatible with BigQuery ones.

For specifics on the service itself, see here

As an example, suppose you want to load all data from an existing table : test_dataset.test_table into BigQuery and pull it into a DataFrame.

from pandas.io import gbq

# Insert your BigQuery Project ID Here
# Can be found in the web console, or
# using the command line tool 'bq ls'
projectid = "xxxxxxxx"

data_frame = gbq.read_gbq('SELECT * FROM test_dataset.test_table', project_id = projectid)
The user will then be authenticated by the `bq` command line client - this usually involves the default browser opening to a login page, though the process can be done entirely from command line if necessary. Datasets and additional parameters can be either configured with `bq`, passed in as options to `read_gbq`, or set using Google’s `gflags` (this is not officially supported by this module, though care was taken to ensure that they should be followed regardless of how you call the method).

Additionally, you can define which column to use as an index as well as a preferred column order as follows:

```python
data_frame = gbq.read_gbq('SELECT * FROM test_dataset.test_table',
    index_col='index_column_name',
    col_order=['col1, col2, col3,...'], project_id = projectid)
```

Finally, if you would like to create a BigQuery table, `my_dataset.my_table`, from the rows of DataFrame, `df`:

```python
df = pandas.DataFrame({
    'string_col_name' : ['hello'],
    'integer_col_name' : [1],
    'boolean_col_name' : [True]})
schema = ['STRING', 'INTEGER', 'BOOLEAN']
data_frame = gbq.to_gbq(df, 'my_dataset.my_table',
    if_exists='fail', schema = schema, project_id = projectid)
```

To add more rows to this, simply:

```python
df2 = pandas.DataFrame({
    'string_col_name' : ['hello2'],
    'integer_col_name' : [2],
    'boolean_col_name' : [False]})
data_frame = gbq.to_gbq(df2, 'my_dataset.my_table', if_exists='append', project_id = projectid)
```

**Note:** A default project id can be set using the command line: `bq init`.

There is a hard cap on BigQuery result sets, at 128MB compressed. Also, the BigQuery SQL query language has some oddities, see here. You can access the management console to determine project id’s by: <https://code.google.com/apis/console/b/0/?noredirect>

---

**Warning:** To use this module, you will need a BigQuery account. See <https://cloud.google.com/products/bigquery> for details.

As of 1/28/14, a known bug is present that could possibly cause data duplication in the resultant dataframe. A fix is imminent, but any client changes will not make it into 0.13.1. See: [http://stackoverflow.com/questions/20984592/bigquery-results-not-including-pagetoken/21009144?noredirect=1#comment32090677_21009144](http://stackoverflow.com/questions/20984592/bigquery-results-not-including-pagetoken/21009144?noredirect=1#comment32090677_21009144)

### 19.11 STATA Format

New in version 0.12.0.

#### 19.11.1 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 [385]: df = DataFrame(randn(10, 2), columns=list('AB'))

In [386]: df.to_stata('stata.dta')

19.11.2 Reading from STATA format

The top-level function read_stata will read a dta format file and return a DataFrame: The class StataReader will read the header of the given dta file at initialization. Its method data() will read the observations, converting them to a DataFrame which is returned:

In [387]: pd.read_stata('stata.dta')
Out[387]:
      A     B
index
0  0.811031 -0.356817
1  1.047085  0.664705
2 -0.086919  0.416905
3 -0.764381 -0.287229
4 -0.089351 -1.035115
5  0.489131 -0.253340
6 -1.948100 -0.116556
7  0.800597  0.796154
8 -0.382952 -0.397373
9 -0.717627  0.156995

[10 rows x 3 columns]

Currently the index is retrieved as a column on read back.

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 variable_labels, which requires data to be called before (see pandas.io.stata.StataReader).

The StataReader supports .dta Formats 104, 105, 108, 113-115 and 117. Alternatively, the function read_stata() can be used.
Functions from pandas.io.data 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

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.

20.1 Yahoo! Finance

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, 01, 27)
In [5]: f=web.DataReader("F", 'yahoo', start, end)
In [6]: f.ix[’2010-01-04’]
Out[6]:
Open    10.17
High    10.28
Low     10.05
Close   10.28
Volume  60855800.00
Adj Close  9.75
Name: 2010-01-04 00:00:00, dtype: float64
20.2 Google Finance

In [7]: import pandas.io.data as web

In [8]: import datetime

In [9]: start = datetime.datetime(2010, 1, 1)

In [10]: end = datetime.datetime(2013, 01, 27)

In [11]: f=web.DataReader("F", ‘google’, start, end)

In [12]: f.ix[‘2010-01-04’]
Out[12]:
Open 10.17
High 10.28
Low 10.05
Close 10.28
Volume 60855796
Name: 2010-01-04 00:00:00, dtype: object

20.3 FRED

In [13]: import pandas.io.data as web

In [14]: import datetime

In [15]: start = datetime.datetime(2010, 1, 1)

In [16]: end = datetime.datetime(2013, 01, 27)

In [17]: gdp=web.DataReader("GDP", "fred", start, end)

In [18]: gdp.ix[‘2013-01-01’]
Out[18]:
GDP 16535.3
Name: 2013-01-01 00:00:00, dtype: float64

# Multiple series:
In [19]: inflation = web.DataReader(["CPIAUCSL", "CPILFESL"], "fred", start, end)

In [20]: inflation.head()
Out[20]:
                 CPIAUCSL  CPILFESL
DATE
2010-01-01    217.4780  220.5440
2010-02-01    217.3560  220.6680
2010-03-01    217.3800  220.7490
2010-04-01    217.2810  220.8080
2010-05-01    217.2300  221.0270

[5 rows x 2 columns]
20.4 Fama/French

Dataset names are listed at Fama/French Data Library.

In [21]: import pandas.io.data as web

In [22]: ip=web.DataReader("5_Industry_Portfolios", "famafrench")

In [23]: ip[4].ix[192607]
Out[23]:
1  Cnsmr  5.43
2  Manuf  2.73
3  HiTec  1.83
4  Hlth  1.64
5  Other  2.15
Name: 192607, dtype: float64

20.5 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.

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     
     Canada  
     2008  36005.5004978584
     2007  36182.9138439757
     2006  35785.9698172849
     2005  35087.8925933298
     Mexico  
     2008  8113.10219480083
     2007  8119.21298908649
     2006  7961.96818458178
     2005  7666.69796097264
     United States  
     2008  43069.5819857208
     2007  43635.5852068142
     2006  43228.114717107
     2005  42516.393469993

The resulting dataset is a properly formatted DataFrame with a hierarchical index, so it is easy to apply .groupby transformations to it:
In [6]: dat[‘NY.GDP.PCAP.KD’].groupby(level=0).mean()
Out[6]:
country
Canada 35765.569188
Mexico 7965.245332
United States 43112.417952
dtype: float64

Now imagine you want to compare GDP to the share of people with cellphone contracts around the world.

In [7]: wb.search(‘cell.*%’).iloc[:,:2]
Out[7]:
id name
3990 IT.CEL.SETS.FE.ZS Mobile cellular telephone users, female (% of ... 
3991 IT.CEL.SETS.MA.ZS Mobile cellular telephone users, male (% of po... 
4027 IT.MOB.COV.ZS Population coverage of mobile cellular telepho...

Notice that this second search was much faster than the first one because Pandas now has a cached list of available data series.

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:

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())

=========================================
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

v

Chapter 20. Remote Data Access
21.1 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.

21.1.1 Pure python

We have a DataFrame to which we want to apply a function row-wise.

```
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
7 450 -1.073215   0.837519   x
8  159  0.119209   1.103245   x
9 650 -1.044236  -1.118384   x
10 389 -0.861849  -0.542980  x
11 772 -2.104569  -0.994002  x
12 174 -0.494929   1.508742  x
13 394  1.071804  -0.328697  x
14 199  0.721555  -0.562235  x
15 318 -0.706771   0.001596  x
... ...  ...   ...
[1000 rows x 4 columns]
```

Here’s the function in pure python:
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:

In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...

We achieve our result by using apply (row-wise):

In [5]: %timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
1 loops, best of 3: 231 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 [6]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
594726 function calls (594725 primitive calls) in 0.339 seconds

Ordered by: internal time
List reduced from 104 to 4 due to restriction <4>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.189</td>
<td>0.000</td>
<td>0.297</td>
<td>0.000</td>
<td>&lt;ipython-input-4-91e33489f136&gt;:1(integrate_f)</td>
</tr>
<tr>
<td>552423</td>
<td>0.103</td>
<td>0.000</td>
<td>0.103</td>
<td>0.000</td>
<td>&lt;ipython-input-3-bc41a25943f6&gt;:1(f)</td>
</tr>
<tr>
<td>3000</td>
<td>0.006</td>
<td>0.000</td>
<td>0.028</td>
<td>0.000</td>
<td>index.py:1019(get_value)</td>
</tr>
<tr>
<td>3000</td>
<td>0.005</td>
<td>0.000</td>
<td>0.034</td>
<td>0.000</td>
<td>series.py:489(<strong>getitem</strong>)</td>
</tr>
</tbody>
</table>

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.

### 21.1.2 Plain cython

First we’re going to need to import the cython magic function to ipython:

In [7]: %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 [8]: %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 [9]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 127 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

## 21.1.3 Adding type

We get another huge improvement simply by providing type information:

```
In [10]: %%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 [11]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 30.1 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 [12]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
3000    0.006    0.000    0.027    0.000 index.py:1019(get_value)
3000    0.005    0.000    0.034    0.000 series.py:489(__getitem__)
3000    0.004    0.000    0.004    0.000 pandas/index.py:1019(pandas.Index.
6004    0.004    0.000    0.004    0.000 {method ‘get_value’ of ‘pandas.
```

## 21.1.4 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 [13]: %cython
    ....: cimport numpy as np
    ....: import numpy as np
    ....: cdef double f_typed(double x) except-2:
    ....:     return x * (x - 1)
    ....: cpdef double integrate_f_typed(double a, double b, int N):
    ....:     cdef int i
    ....:     cdef double s, dx
    ....:     s = 0
    ....:     dx = (b - a) / N
    ....:     for i in range(N):
    ....:         s += f_typed(a + i * dx)
    ....:     return s * dx
    ....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_b, np.ndarray col_N):
    ....:     assert (col_a.dtype == np.float and col_b.dtype == np.float and col_N.dtype == np.int)
    ....:     cdef Py_ssize_t i, n = len(col_N)
    ....:     assert (len(col_a) == len(col_b) == n)
    ....:     cdef np.ndarray[double] res = np.empty(n)
    ....:     for i in range(len(col_a)):
    ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
    ....:     return res

The implementation is simple, it creates an array of zeros and loops over the rows, applying our integrate_f_typed, and putting this in the zeros array.

**Warning:** In 0.13.0 since Series has internaly been refactored 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 [14]: %timeit apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
100 loops, best of 3: 2.56 ms per loop

We’ve gotten another big improvement. Let’s check again where the time is spent:

In [15]: %prun -l 4 apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
33 function calls in 0.003 seconds

Ordered by: internal time
List reduced from 13 to 4 due to restriction <4>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
<td>_cython_magic_5a6f3e72f584366fd3c783dd53b41dc3.apply_integrate_f_typed</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>&lt;string&gt;:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>3</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>frame.py:1635(<strong>getitem</strong>)</td>
</tr>
<tr>
<td>3</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>index.py:604(<strong>contains</strong>)</td>
</tr>
</tbody>
</table>
As one might expect, the majority of the time is now spent in apply_integrate_f, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

### 21.1.5 More advanced techniques

There is still scope for improvement, here’s an example of using some more advanced cython techniques:

```python
In [16]: %cython
....: cimport cython
....: cimport numpy as np
....: import numpy as np
....: cdef double f_typed(double x) except -2:
      ....:     return x * (x - 1)
....: cdef double integrate_f_typed(double a, double b, int N):
    ....:     cdef int i
    ....:     cdef double s, dx
    ....:     s = 0
    ....:     dx = (b - a) / N
    ....:     for i in range(N):
    ....:         s += f_typed(a + i * dx)
    ....:     return s * dx
....: @cython.boundscheck(False)
    ....: @cython.wraparound(False)
      ....:     cdef 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 [17]: %timeit apply_integrate_f_wrap(df['a'].values, df['b'].values, df['N'].values)
100 loops, best of 3: 2.16 ms 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.

### 21.1.6 Further topics

- Loading C modules into cython.

Read more in the cython docs.

### 21.2 Expression Evaluation via eval() (Experimental)

New in version 0.13. The top-level function `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()`.

### 21.2.1 Supported Syntax

These operations are supported by `eval()`:

- Arithmetic operations except for the left shift (`<<`) and right shift (`>>`) operators, e.g., `df + 2 * pi / s ** 4 % 42 - the_golden_ratio`
- Comparison operations, e.g., `2 < df < df2`
- Boolean operations, e.g., `df < df2 and df3 < df4 or not df_bool`
- List and tuple literals, e.g., `[1, 2] or (1, 2)`
- Attribute access, e.g., `df.a`
- Subscript expressions, e.g., `df[0]`
- Simple variable evaluation, e.g., `pd.eval('df')` (this is not very useful)

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.`
21.2.2 eval() Examples

eval() works wonders for expressions containing large arrays

First let’s create 4 decent-sized arrays to play with:

```python
In [18]: import pandas as pd
In [19]: from pandas import DataFrame, Series
In [20]: from numpy.random import randn
In [21]: import numpy as np
In [22]: nrows, ncols = 20000, 100
In [23]: df1, df2, df3, df4 = [DataFrame(randn(nrows, ncols)) for __ in xrange(4)]
```

Now let’s compare adding them together using plain ol’ Python versus eval():

```python
In [24]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 55.9 ms per loop
In [25]: %timeit pd.eval('df1 + df2 + df3 + df4')
10 loops, best of 3: 31.1 ms per loop
```

Now let’s do the same thing but with comparisons:

```python
In [26]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
10 loops, best of 3: 67.1 ms per loop
In [27]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
10 loops, best of 3: 25 ms per loop
```

eval() also works with unaligned pandas objects:

```python
In [28]: s = Series(randn(50))
In [29]: %timeit df1 + df2 + df3 + df4 + s
10 loops, best of 3: 91.7 ms per loop
In [30]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
10 loops, best of 3: 22.6 ms per loop
```

**Note:** Operations such as

```python
1 and 2  # would parse to 1 & 2, but should evaluate to 2
3 or 4   # would parse to 3 | 4, but should evaluate to 3
~1       # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type bool or np.bool_. Again, you should perform these kinds of operations in plain Python.

21.2.3 The DataFrame.eval method (Experimental)

In addition to the top level eval() function you can also evaluate an expression in the “context” of a DataFrame.
In [31]: df = DataFrame(randn(5, 2), columns=['a', 'b'])

In [32]: df.eval('a + b')
Out[32]:
0   -0.246747
1     0.867786
2   -1.626063
3   -1.134978
4   -1.027798
dtype: float64

Any expression that is a valid `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 [33]: df = DataFrame(dict(a=range(5), b=range(5, 10)))

In [34]: df.eval('c = a + b')

In [35]: df.eval('d = a + b + c')

In [36]: df.eval('a = 1')

In [37]: df
Out[37]:
   a  b  c  d
0  0  5  5 10
1  1  6  7 14
2  2  7  9 18
3  3  8 11 22
4  4  9 13 26

[5 rows x 4 columns]

### 21.2.4 Local Variables

You can refer to local variables the same way you would in vanilla Python

In [38]: df = DataFrame(randn(5, 2), columns=['a', 'b'])

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

**Note:** The one exception is when you have a local (or global) with the same name as a column in the `DataFrame`
df = DataFrame(randn(5, 2), columns=['a', 'b'])
a = randn(len(df))
df.eval('a + b')
NameResolutionError: resolvers and locals overlap on names ['a']

To deal with these conflicts, a special syntax exists for referring variables with the same name as a column

In [41]: df.eval('@a + b')
Out[41]:
        0  0.291737
       1 -0.073076
       2 -2.259372
       3 -1.513447
       4 -0.222295
       dtype: float64

The same is true for query()

In [42]: df.query('@a < b')
Out[42]:
       a     b
     1 -1.215949  1.670837
     3 -0.835893  0.315213
     4 -1.337334  0.772364
[3 rows x 2 columns]

21.2.5 eval() Parsers

There are two different parsers and 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 [43]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'
In [44]: x = pd.eval(expr, parser='python')
In [45]: expr_no_parens = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'
In [46]: y = pd.eval(expr_no_parens, parser='pandas')
In [47]: np.all(x == y)
Out[47]: True

The same expression can be “anded” together with the word and as well:

In [48]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'
In [49]: x = pd.eval(expr, parser='python')
In [50]: expr_with_ands = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'
In [51]: y = pd.eval(expr_with_ands, parser='pandas')

In [52]: np.all(x == y)
Out[52]: True

The and and or operators here have the same precedence that they would in vanilla Python.

### 21.2.6 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 eval() engines against it. You will achieve no performance benefits using eval() with engine='python'.

---

You can see this by using eval() with the ’python’ engine is actually a bit slower (not by much) than evaluating the same expression in Python:

In [53]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 62.7 ms per loop

In [54]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
10 loops, best of 3: 50.8 ms per loop

### 21.2.7 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 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().

### 21.2.8 Technical Minutia

- Expressions that would result in an object dtype (including simple variable evaluation) have to 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—that’s a string comparison and-ed together with another boolean expression that’s from a numeric comparison, the numeric comparison will be evaluated by numexpr. In fact, in general, query()/eval() will “pick out” the subexpressions that are eval-able by numexpr and those that must be evaluated in Python space transparently to the user.
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:

```python
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
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The to_sparse method takes a kind argument (for the sparse index, see below) and a fill_value. So if we had a mostly zero Series, we could convert it to sparse with fill_value=0:

```python
In [5]: ts.fillna(0).to_sparse(fill_value=0)
Out[5]:
   0  0.469112
   1 -0.282863
   2  0.000000
   3  0.000000
   4  0.000000
   5  0.000000
   6  0.000000
   7  0.000000
   8 -0.861849
   9 -2.104569
```
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
7   NaN NaN NaN NaN
8   NaN NaN NaN NaN
9   NaN NaN NaN NaN
10  NaN NaN NaN NaN
11  NaN NaN NaN NaN
12  NaN NaN NaN NaN
13  NaN NaN NaN NaN
14  NaN NaN NaN NaN
       ... ... ... ...
[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 NaN  NaN  NaN
1 -0.282863 NaN  NaN  NaN
2   NaN    NaN  NaN  NaN
3   NaN    NaN  NaN  NaN
4   NaN    NaN  NaN  NaN
5   NaN    NaN  NaN  NaN
6   NaN    NaN  NaN  NaN
7   NaN    NaN  NaN  NaN
8 -0.861849 NaN  NaN  NaN
9 -2.104569 NaN  NaN  NaN
dtype: float64
```
22.1 SparseArray

SparseArray is the base layer for all of the sparse indexed data structures. It is a 1-dimensional ndarray-like object storing only values distinct from the fill_value:

In [12]: arr = np.random.randn(10)


In [14]: sparr = 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], dtype=int32)

Like the indexed objects (SparseSeries, SparseDataFrame, SparsePanel), a SparseArray can be converted back to a regular ndarray by calling to_dense:

In [16]: sparr.to_dense()
Out[16]:
array([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453, nan, 0.606 , 1.3342])

22.2 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):

In [17]: spl = SparseList()

In [18]: spl
Out[18]: <pandas.sparse.list.SparseList object at 0x12b1e110>

The two important methods are append and to_array. append can accept scalar values or any 1-dimensional sequence:

In [19]: spl.append(np.array([1., nan, nan, 2., 3.]))

In [20]: spl.append(5)

In [21]: spl.append(sparr)

In [22]: spl
Out[22]:
<pandas.sparse.list.SparseList object at 0x12b1e110>
[1.0, nan, nan, 2.0, 3.0]
Fill: nan
IntIndex
Indices: array([0, 3, 4], dtype=int32)

[5.0]
Fill: nan
IntIndex
Indices: array([0], dtype=int32)

[[-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], dtype=int32)

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 [23]: spl.to_array()
Out[23]:
[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],
 Fill: nan
IntIndex
Indices: array([ 0, 3, 4, 5, 6, 7, 11, 12, 14, 15], dtype=int32)

### 22.3 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.
23.1 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]):
    ...
```

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")
```

or return if any value is True.

```python
>>> if pd.Series([False, True, False]).any():
    print("I am any")
```

To evaluate single-element pandas objects in a boolean context, use the method .bool():

```python
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
```
23.1.1 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))
```

```python
>>> s == 4
0 False
1 False
2 False
3 False
4 True
dtype: bool
```

See boolean comparisons for more examples.

23.2 NaN, Integer NA values and NA type promotions

23.2.1 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.

23.2.2 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'))
```

```python
In [6]: s
Out[6]:
a     1
b     2
c     3
d     4
e     5
dtype: int64
```

```python
In [7]: s.dtype
Out[7]: dtype('int64')
```

```python
In [8]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])
```

```python
In [9]: s2
```
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.

### 23.2.3 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.

### 23.2.4 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>numpy.integer</td>
<td>int8, int16, int32, int64</td>
</tr>
<tr>
<td>numpy.unsignedinteger</td>
<td>uint8, uint16, uint32, uint64</td>
</tr>
<tr>
<td>numpy.object_</td>
<td>object_</td>
</tr>
<tr>
<td>numpy.bool_</td>
<td>bool_</td>
</tr>
<tr>
<td>numpy.character</td>
<td>string_, unicode_</td>
</tr>
</tbody>
</table>

The R language, by contrast, only has a handful of built-in data types: integer, numeric (floating-point), character, and boolean. NA types are implemented by reserving special bit patterns for each type to be used as the missing value. While doing this with the full NumPy type hierarchy would be possible, it would be a more substantial trade-off (especially for the 8- and 16-bit data types) and implementation undertaking.

An alternate approach is that of using masked arrays. A masked array is an array of data with an associated boolean `mask` denoting whether each value should be considered NA or not. I am personally not in love with this approach as I feel that overall it places a fairly heavy burden on the user and the library implementer. Additionally, it exacts a fairly high performance cost when working with numerical data compared with the simple approach of using NaN. Thus, I have chosen the Pythonic “practicality beats purity” approach and traded integer NA capability for a much simpler approach of using a special value in float and object arrays to denote NA, and promoting integer arrays to floating when NAs must be introduced.
23.3 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).

23.4 Label-based slicing conventions

23.4.1 Non-monotonic indexes require exact matches

23.4.2 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  0.499281
   b -1.405256
   c  0.162565
   d -0.067785
   e -1.260006
   f -1.132896
   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  0.162565
   d -0.067785
   e -1.260006
   dtype: float64
```

However, if you only had c and e, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```python
s.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:
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.

### 23.5 Miscellaneous indexing gotchas

#### 23.5.1 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 -2.006481 0.301016 0.059117 1.138469
  b -2.400634 -0.280853 0.025653 -1.386071
  c  0.863937 0.252462 1.500571 1.053202
  d -2.338595 -0.374279 -2.359958 -1.157886
  e -0.551865 1.592673 1.559318 1.562443
  f  0.763264 0.162027 -0.902704 1.106010

[6 rows x 4 columns]

In [17]: df.ix[['b', 'c', 'e']]
Out[17]:
   one  two  three  four
  b -2.400634 -0.280853 0.025653 -1.386071
  c  0.863937 0.252462 1.500571 1.053202
  e -0.551865 1.592673 1.559318 1.562443

[3 rows x 4 columns]
```

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 -2.400634 -0.280853 0.025653 -1.386071
  c  0.863937 0.252462 1.500571 1.053202
  e -0.551865 1.592673 1.559318 1.562443

[3 rows x 4 columns]
```

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 -2.400634 -0.280853  -0.025653 -1.386071
c  0.863937  0.252462   1.500571  1.053202
e -0.551865  1.592673   1.559318  1.562443
[3 rows x 4 columns]

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
[3 rows x 4 columns]

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.

23.5.2 Reindex potentially changes underlying Series dtype

The use of reindex_like can potentially change the dtype of a Series.

```
series = pandas.Series([1, 2, 3])
x = pandas.Series([True])
x.dtype
x = pandas.Series([True]).reindex_like(series)
x.dtype
```
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.

### 23.6 Timestamp limitations

#### 23.6.1 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:

```python
In [25]: begin = Timestamp.min

In [26]: begin
Out[26]: Timestamp('1677-09-22 00:12:43.145225', tz=None)

In [27]: end = Timestamp.max

In [28]: end
```

If you need to represent time series data outside the nanosecond timespan, use PeriodIndex:

```python
In [29]: span = period_range('1215-01-01', '1381-01-01', freq='D')

In [30]: span
Out[30]: <class 'pandas.tseries.period.PeriodIndex'>
freq: D
[1215-01-01, ..., 1381-01-01]
length: 60632
```

### 23.7 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:

```python
In [31]: 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 [32]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [33]: 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 [34]: df
Out[34]:
<table>
<thead>
<tr>
<th>actual</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>nominal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999-01-27</td>
<td>19:00:00</td>
<td>19:56:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
<tr>
<td>1999-01-27</td>
<td>20:00:00</td>
<td>19:56:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
<tr>
<td>1999-01-27</td>
<td>21:00:00</td>
<td>20:56:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
<tr>
<td>1999-01-27</td>
<td>21:00:00</td>
<td>21:18:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
<tr>
<td>1999-01-27</td>
<td>22:00:00</td>
<td>21:56:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
<tr>
<td>1999-01-27</td>
<td>23:00:00</td>
<td>22:56:00</td>
<td>KORD</td>
<td>19990127</td>
</tr>
</tbody>
</table>

4

nominal
1999-01-27 19:00:00 0.81
1999-01-27 20:00:00 0.01
1999-01-27 21:00:00 -0.59
1999-01-27 21:00:00 -0.99
1999-01-27 22:00:00 -0.59
1999-01-27 23:00:00 -0.59

[6 rows x 6 columns]

23.8 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.

23.9 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the \text{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.

23.10 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 it’s 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:beautifulsoup'
```

Note that you need bzr and git installed to perform the last two operations.
23.11 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. 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 [35]: x = np.array(list(range(10)), 'i4')  # big endian
In [36]: newx = x.byteswap().newbyteorder()  # force native byteorder
In [37]: s = Series(newx)
```

See the NumPy documentation on byte order for more details.
Note: This is all highly experimental. I would like to get more people involved with building a nice RPy2 interface for pandas.

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.

### 24.1 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>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>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>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>
</tbody>
</table>
24.2 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`):

```python
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):

```python
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
```

24.3 Calling R functions with pandas objects

24.4 High-level interface to R estimators
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.

25.1 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.

25.2 Vincent

The Vincent project leverages Vega (that in turn, leverages d3) to create plots. It has great support for pandas data objects.

25.3 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.

25.4 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 than those offered by pandas.

25.5 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.

25.6 sklearn-pandas

Use pandas DataFrames in your scikit-learn ML pipeline.
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.

### 26.1 Base R

#### 26.1.1 aggregate

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

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

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

```python
In [1]: from pandas import DataFrame

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

In [3]: g = df.groupby(['by1','by2'])
```
In [4]: g[['v1','v2']].mean()
Out[4]:
          v1  v2
by1  by2  
   1   95  5  55
   99  5  55
   2   95   7  77
   99  NaN NaN
big damp  3  33
blue dry   3  33
red red    4  44
   wet  1  11

[8 rows x 2 columns]

For more details and examples see the groupby documentation.

26.1.2 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:

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

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

In [5]: s = pd.Series(np.arange(5),dtype=np.float32)

In [6]: s.isin([2, 4])
Out[6]:
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:

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

The apply() method can be used to replicate this:

In [7]: s = pd.Series(np.arange(5),dtype=np.float32)

In [8]: Series(pd.match(s,[2,4],np.nan))
Out[8]:
0 NaN
1 NaN
2 0
3 NaN
4 1
dtype: float64

For more details and examples see the reshaping documentation.
26.1.3 tapply

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

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

tapply(baseball$batting.average, baseball.example$team, max)
```

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

```
In [9]: import random
In [10]: import string
In [11]: baseball = 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 [12]: baseball.pivot_table(values='batting avg', cols='team', aggfunc=np.max)
```

```
Out[12]:
    team
    team 1 0.321235
    team 2 0.393991
    team 3 0.386815
    team 4 0.387197
    team 5 0.392086
Name: batting avg, dtype: float64
```

For more details and examples see the reshaping documentation.

26.1.4 subset

New in version 0.13. The `query()` method is similar to the base R `subset` function. In R you might want to get the rows of a data.frame where one column’s values are less than another column’s values:

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

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

```
In [13]: df = DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [14]: df.query('a <= b')
```

```
Out[14]:
    a    b
2 -0.838260  0.980077
6 -0.017685  0.027505
```
For more details and examples see the query documentation.

26.1.5 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)
```

```r
df$a + df$b  # same as the previous expression
```

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

```python
In [17]: df = DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [18]: df.eval('a + b')
```

```python
Out[18]:
0   -0.163194
1    0.985872
2    2.864538
3    0.782622
4    0.962818
5    1.974849
6    0.258445
7   -2.288045
8    -0.800437
9    2.667426
dtype: float64
```

```python
In [19]: df.a + df.b  # same as the previous expression
```

```python
Out[19]:
0   -0.163194
```
In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see the `eval` documentation.

### 26.2 zoo

### 26.3 xts

### 26.4 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>

#### 26.4.1 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 [20]: df = 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': sample(1:4, 120, TRUE)
}
```
In [21]: grouped = df.groupby(['month','week'])

In [22]: print grouped['x'].agg([np.mean, np.std])

<table>
<thead>
<tr>
<th>month</th>
<th>week</th>
<th>mean</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>74.750543</td>
<td>37.602035</td>
</tr>
<tr>
<td>2</td>
<td>91.420601</td>
<td>56.817107</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>80.270102</td>
<td>55.994654</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>81.840060</td>
<td>50.966643</td>
</tr>
<tr>
<td>2</td>
<td>97.434542</td>
<td>59.919288</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>79.867371</td>
<td>47.377914</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>83.997435</td>
<td>39.391772</td>
</tr>
<tr>
<td>2</td>
<td>86.244632</td>
<td>41.066830</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>108.811608</td>
<td>45.048738</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>81.647843</td>
<td>50.264539</td>
</tr>
<tr>
<td>2</td>
<td>94.056653</td>
<td>47.677568</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>76.004631</td>
<td>47.048914</td>
<td></td>
</tr>
</tbody>
</table>

[12 rows x 2 columns]

For more details and examples see the groupby documentation.

### 26.5 reshape / reshape2

#### 26.5.1 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 [23]: `a = np.array(range(1,24)+[np.NAN]).reshape(2,3,4)`

In [24]: `DataFrame([tuple(list(x)+[val]) for x, val in np.ndenumerate(a)])`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>
26.5.2 melt.list

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

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

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.

```python
In [25]: a = list(enumerate(range(1,5)+[np.NAN]))
In [26]: DataFrame(a)
Out[26]:
   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.

26.5.3 melt.data.frame

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

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

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

```python
In [27]: cheese = DataFrame({'first' : ['John', 'Mary'],
                         ....:     'last' : ['Doe', 'Bo'],
                         ....:     'height' : [5.5, 6.0],
                         ....:     'weight' : [130, 150]})
...
In [28]: pd.melt(cheese, id_vars=['first', 'last'])
Out[28]:
   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 **26.5.4 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 [30]: df = 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 [31]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [32]: pd.pivot_table(mdf, values='value', rows=['variable','week'],
    .....:     cols=['month'], aggfunc=np.mean)
```

Similarly for *dcast* which uses a data.frame called *df* in R to aggregate information based on *Animal* and *FeedType*:
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 [33]: df = DataFrame({
      ....:   'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
      ....:                 'Animal2', 'Animal3'],
      ....:   'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
      ....:   'Amount': [10, 7, 4, 2, 5, 6, 2],
      ....: })
      ....: 

In [34]: df.pivot_table(values='Amount', rows='Animal', cols='FeedType', aggfunc='sum')
Out[34]:
          FeedType
Animal    A  B
Animal1    10  5
Animal2    2  13
Animal3    6  NaN

[3 rows x 2 columns]
```

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

```python
In [35]: df.groupby(['Animal','FeedType'])['Amount'].sum()
Out[35]:
          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.
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)
```

```python
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
[5 rows x 7 columns]
```

### 27.1 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]:
```
Calling the DataFrame without the list of column names would display all columns (akin to SQL’s `*`).

### 27.2 WHERE

Filtering in SQL is done via a WHERE clause.

```sql
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

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

```python
In [7]: tips[tips['time'] == 'Dinner'].head(5)
Out[7]:
   total_bill  tip   sex  smoker  day      time  size
0   16.99   1.01 Female  No  Sun  Dinner     2
1   10.34   1.66    Male  No  Sun  Dinner     3
2   21.01   3.50    Male  No  Sun  Dinner     3
3   23.68   3.31    Male  No  Sun  Dinner     2
4   24.59   3.61 Female  No  Sun  Dinner     4
```

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

```python
In [8]: is_dinner = tips['time'] == 'Dinner'
In [9]: is_dinner.value_counts()
Out[9]:
    True    176
    False   68
Name: time, dtype: int64
```

```python
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).
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;

# tips of more than $5.00 at Dinner meals
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
Out[11]:
   total_bill  tip  sex  smoker  day  time  size
 0    23.00   3.70  Male     No  Sat     4
 1    24.42   6.15  Male     No  Sat     4
 2    34.81   6.45  Female    No  Sun     4
 3    54.14   6.65  Male     No  Sun     4
 4    41.00   6.75  Female    No  Sun     4
 5    32.60   5.20  Male     No  Sun     4
 6    59.00   6.75  Female    No  Sun     4
 7    30.35   6.70  Male     No  Sun     4
 8    70.50   6.70  Male     No  Sun     4
 9    21.00   6.70  Male     No  Sun     4
10   11.00   6.70  Female    No  Sun     4

[15 rows x 7 columns]

-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;

# tips by parties of at least 5 diners OR bill total was more than $45
In [12]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
Out[12]:
   total_bill  tip  sex  smoker  day  time  size
 0    59.00   6.73  Male     No  Sat     4
 1   125.00   4.20  Female    No  Thur     6
 2   141.00   6.70  Male     No  Thur     6
 3   142.00   5.00  Male     No  Thur     6
 4   143.00   5.00  Female    No  Thur     6
 5   155.00   5.14  Female    No  Sun     5
 6   156.00   5.00  Male     No  Sun     6
 7   170.00  10.00  Male     Yes  Sat     3
 8   182.00   3.50  Male     Yes  Sun     3
 9   185.00   5.00  Male     No  Sun     5
10  212.00   9.00  Male     No  Sun     4
11  216.00   3.00  Male     Yes  Sat     5

[13 rows x 7 columns]

NULL checking is done using the notnull() and isnull() methods.

                              'col2': ['F', np.NaN, 'G', 'H', 'I']})

27.2. WHERE
Assume we have a table of the same structure as our DataFrame above. We can see only the records where \texttt{col2} IS NULL with the following query:

\begin{verbatim}
SELECT *
FROM frame
WHERE col2 IS NULL;
\end{verbatim}

\begin{verbatim}
In [15]: frame[frame['col2'].isnull()]
Out[15]:
col1  col2
1    B  NaN
[1 rows x 2 columns]
\end{verbatim}

Getting items where \texttt{col1} IS NOT NULL can be done with \texttt{notnull()}.

\begin{verbatim}
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
\end{verbatim}

\begin{verbatim}
In [16]: frame[frame['col1'].notnull()]
Out[16]:
col1  col2
0    A  F
1    B  NaN
3    C  H
4    D  I
[4 rows x 2 columns]
\end{verbatim}

\section*{27.3 GROUP BY}

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

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

\begin{verbatim}
SELECT sex, count(*)
FROM tips
GROUP BY sex;
\end{verbatim}

```sql
/*
Female 87
Male 157
*/
```
The pandas equivalent would be:

```
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.

```
In [18]: tips.groupby('sex').count()
Out[18]:
       total_bill  tip  sex  smoker  day  time  size
     sex
Female  87  87  87  87  87  87  87
Male    157 157 157 157 157 157 157
```

[2 rows x 7 columns]

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

```
In [19]: tips.groupby('sex')['total_bill'].count()
Out[19]:
       sex
Female  87
Male    157
```

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
```

[4 rows x 2 columns]

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;
```
In [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})
Out[21]:

<table>
<thead>
<tr>
<th>smoker</th>
<th>day</th>
<th>size</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Fri</td>
<td>4 2.812500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>45 3.102889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>57 3.167895</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>45 2.673778</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Fri</td>
<td>15 2.714000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>42 2.875476</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>19 3.516842</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>17 3.030000</td>
<td></td>
</tr>
</tbody>
</table>

[8 rows x 2 columns]

### 27.4 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).

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.

#### 27.4.1 INNER JOIN

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

In [24]: pd.merge(df1, df2, on='key')
merge() also offers parameters for cases when you’d like to join one DataFrame’s column with another DataFrame’s index.

In [25]: indexed_df2 = df2.set_index('key')

In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)

Out[26]:

<table>
<thead>
<tr>
<th>key</th>
<th>value_x</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.163921</td>
<td>0.156585</td>
</tr>
<tr>
<td>D</td>
<td>0.872202</td>
<td>0.711517</td>
</tr>
<tr>
<td>D</td>
<td>0.872202</td>
<td>-0.105817</td>
</tr>
</tbody>
</table>

[3 rows x 3 columns]

27.4.2 LEFT OUTER JOIN

-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;

# show all records from df1
In [27]: pd.merge(df1, df2, on='key', how='left')

Out[27]:

<table>
<thead>
<tr>
<th>key</th>
<th>value_x</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1.348332</td>
<td>NaN</td>
</tr>
<tr>
<td>B</td>
<td>0.163921</td>
<td>0.156585</td>
</tr>
<tr>
<td>C</td>
<td>0.706914</td>
<td>NaN</td>
</tr>
<tr>
<td>D</td>
<td>0.872202</td>
<td>0.711517</td>
</tr>
<tr>
<td>D</td>
<td>0.872202</td>
<td>-0.105817</td>
</tr>
</tbody>
</table>

[5 rows x 3 columns]

27.4.3 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]:

<table>
<thead>
<tr>
<th>key</th>
<th>value_x</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.163921</td>
<td>0.156585</td>
</tr>
</tbody>
</table>
27.4.4 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).

```sql
-- 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 -1.348332      NaN
1     B  0.163921  0.156585
2     C  0.706914      NaN
3     D  0.872202 -0.105817
4     D  0.872202      NaN
5     E      NaN -0.087338
[6 rows x 3 columns]
```

27.5 UNION

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

```python
In [30]: df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],
                       'rank': range(1, 4)})
In [31]: df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],
                       'rank': [1, 4, 5]})

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

In [32]: pd.concat([df1, df2])
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

[6 rows x 2 columns]

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
2  Los Angeles     5

[5 rows x 2 columns]

27.6 UPDATE

27.7 DELETE
Chapter 27. Comparison with SQL
28.1 Input/Output

28.1.1 Pickling

**read_pickle**(path)  Load pickled pandas object (or any other pickled object) from the specified file path

**pandas.read_pickle**

Load pickled pandas object (or any other pickled object) from the specified file path

**Parameters**  **path** : string

File path

**Returns**  **unpickled** : type of object stored in file

28.1.2 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.read_table

**pandas.read_table** *(filepath_or_buffer,  sep='\t', dialect=None, compression=None, doublequote=True, escapechar=None, quoting=0, skipinitialspace=False, lineterminator=None, header='infer', index_col=None, names=None, prefix=None, skiprows=None, skipfooter=None, na_values=None, na_rep=None, true_values=None, false_values=None, delimiter=None, converters=None, usecols=None, engine='c', delim_whitespace=False, as_recarray=False, na_filter=True, low_memory=True, 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, nrows=None, iterator=False, chunksize=None, verbose=False, encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_cols=False, infer_datetime_format=False)*

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  (tab-stop)
  Delimiter to use. Regular expressions are accepted.

- **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

- **dtype** : Type name or dict of column -> type
  Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

- **compression** : {‘gzip’, ‘bz2’, None}, default None
  For on-the-fly decompression of on-disk data

- **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 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)

skiprows : list-like or integer
Row numbers to skip (0-indexed) or number of rows 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 or None (default)
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.

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 Does not support line commenting
    (will return empty line)

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 line at bottom of file to skip

converters : dict, optional
    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’)

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

**Returns** result : DataFrame or TextParser

---

### pandas.read_csv

**pandas.read_csv**

```python
def pandas.read_csv(filepath_or_buffer, sep=',', dialect=None, compression=None, doublequote=True, escapechar=None, quoting=0, skipinitialspace=False, skiprows=None, skipfooter=None, na_values=None, keep_default_na=True, thousands=None, comment=None, decimal='.', parse_dates=False, memory_map=False, nrows=None, iterator=False, chunksize=None, verbose=False, encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_cols=False, infer_datetime_format=False)
```

Read CSV (comma-separated) 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 ','

  Delimiter to use. If sep is None, will try to automatically determine this. Regular expressions are accepted.

- **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

  Disable whitespace at the start of each line for tabular data.
Skip spaces after delimiter

**escapechar**: string  

**dtype**: Type name or dict of column -> type  
Data type for data or columns. E.g. `{‘a’: np.float64, ‘b’: np.int32}`

**compression**: `{‘gzip’, ‘bz2’, None}`, default None  
For on-the-fly decompression of on-disk data

**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 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)

**skiprows**: list-like or integer  
Row numbers to skip (0-indexed) or number of rows 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 or None (default)  
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.

**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 Does not support line commenting (will return empty line)

**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 line at bottom of file to skip

**converters** : dict. optional
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’)

**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

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 con-
    tinuous.

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

**dtype** : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

**compression** : {'gzip', 'bz2', None}, default None

For on-the-fly decompression of on-disk data

**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 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)

**skiprows** : list-like or integer

Row numbers to skip (0-indexed) or number of rows 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 or None (default)

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.

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 Does not support line commenting (will return empty line)

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 line at bottom of file to skip

converters : dict, optional

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 : str, 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’)

**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

**Returns**  
result : DataFrame or TextParser

Also, ‘delimiter’ is used to specify the filler character of the fields if it is not spaces (e.g., ‘~’).

### 28.1.3 Clipboard

**read_clipboard(****kwargs**)  
Read text from clipboard and pass to read_table.

**pandas.read_clipboard**

**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
28.1.4 Excel

**read_excel**(io, sheetname, **kwds) Read an Excel table into a pandas DataFrame

**ExcelFile.parse**(sheetname[, header, ...]) Read an Excel table into DataFrame

**pandas.read_excel**

**pandas.read_excel**(io, sheetname, **kwds)

Read an Excel table into a pandas DataFrame

**Parameters**

io : string, file-like object or xlrd workbook

If a string, expected to be a path to xls or xlsx file

sheetname : string

Name of Excel sheet

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)

skipfooter : int, default 0

Rows at the end to skip (0-indexed)

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

engine : string, default None

If io is not a buffer or path, this must be set to identify io. Acceptable values are None or xlrd

convert_float : boolean, default True
convert integral floats to int (i.e., 1.0 -> 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally.

**Returns**  
parsed : DataFrame  
DataFrame from the passed in Excel file

**pandas.ExcelFile.parse**

ExcelFile.parse(sheetname, 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, **kwds)

Read an Excel table into DataFrame

**Parameters**  
sheetname : string or integer  
Name of Excel sheet or the page number of the sheet

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)

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”)

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 \rightarrow 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

**Returns**

**parsed**: DataFrame

Dataframe parsed from the Excel file

### 28.1.5 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 orients 'index' and 'columns'.
  - The DataFrame columns must be unique for orients '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

---

### 28.1.6 HTML

**read_html**(io, match, flavor, header, ...))  
Read HTML tables into a list of DataFrame objects.

**pandas.read_html**

```python
pandas.read_html (io, match='+', flavor=None, header=None, index_col=None, skiprows=None, infer_types=None, attrs=None, parse_dates=False, tupleize_cols=False, thousands=',', )
```

Read HTML tables into a list of DataFrame objects.

**Parameters**  
io : str or file-like

A URL, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with ‘https’ you might try removing the ‘s’.

**match** : str or compiled regular expression, optional

The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to ‘.+’ (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

**flavor** : str or None, container of strings
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** : bool, optional

This option is deprecated in 0.13, an will have no effect in 0.14. It defaults to True.

**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,

```python
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.

```python
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. In 0.13, this parameter can sometimes interact strangely with infer_types. If you get a large number of NaT values in your results, consider passing infer_types=False and manually converting types afterwards.

**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 ‘,’.

**Returns** : dfs : list of DataFrames

See Also:

pandas.io.parsers.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.

### 28.1.7 HDFStore: PyTables (HDF5)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_hdf(path_or_buf, key, **kwargs)</code></td>
<td>read from the store, close it if we opened it</td>
</tr>
<tr>
<td><code>HDFStore.put(key, value[, format, append])</code></td>
<td>Store object in HDFStore</td>
</tr>
<tr>
<td><code>HDFStore.append(key, value[, format, ...])</code></td>
<td>Append to Table in file. Node must already exist and be Table</td>
</tr>
<tr>
<td><code>HDFStore.get(key)</code></td>
<td>Retrieve pandas object stored in file</td>
</tr>
<tr>
<td><code>HDFStore.select(key[, where, start, stop, ...])</code></td>
<td>Retrieve pandas object stored in file, optionally based on where</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
  
  - `auto_close`: optional, boolean, should automatically close the store
when finished, default is False

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

**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 represenation
  - **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**

```python
HDFStore.get(key)
```

Retrieve pandas object stored in file

**Parameters**  
- `key`: object

**Returns**  
- `obj`: type of object stored in file

**pandas.HDFStore.select**

```python
HDFStore.select(key, where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, auto_close=False, **kwargs)
```

Retrieve pandas object stored in file, optionally based on where criteria

**Parameters**  
- `key`: object
  - `where`: list of Term (or convertable) objects, optional
  - `start`: integer (defaults to None), row number to start selection
  - `stop`: integer (defaults to None), row number to stop selection
  - `columns`: a list of columns that if not None, will limit the return columns
  - `iterator`: boolean, return an iterator, default False
  - `chunksize`: nrows to include in iteration, return an iterator
  - `auto_close`: boolean, should automatically close the store when finished, default is False

**Returns**  
The selected object

**28.1.8 SQL**

```python
read_sql(sql, con[, index_col, ...])
```

Returns a DataFrame corresponding to the result set of the query string.

**pandas.read_sql**

```python
pandas.read_sql(sql, con[, index_col=None, coerce_float=True, params=None])
```

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 will be 0 to len(results) - 1.

**Parameters**
- `sql`: string
  - SQL query to be executed
con: DB connection object, optional

index_col: string, optional

column name to use 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 or tuple, optional

List of parameters to pass to execute method.

read_frame(sql, con[, index_col, ...])

Returns a DataFrame corresponding to the result set of the query string.

write_frame(frame, name, con[, flavor, ...])

Write records stored in a DataFrame to a SQL database.

pandas.io.sql.read_frame

pandas.io.sql.read_frame(sql, con, index_col=None, coerce_float=True, params=None)

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 will be 0 to len(results) - 1.

Parameters

sql: string

SQL query to be executed

con: DB connection object, optional

index_col: string, optional

column name to use 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 or tuple, optional

List of parameters to pass to execute method.

pandas.io.sql.write_frame

pandas.io.sql.write_frame(frame, name, con, flavor='sqlite', if_exists='fail', **kwargs)

Write records stored in a DataFrame to a SQL database.

Parameters

frame: DataFrame

name: name of SQL table

con: an open SQL database connection object

flavor: {'sqlite', 'mysql', 'oracle'}, default 'sqlite'

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.
28.1.9 Google BigQuery

**read_gbq**(query[, project_id, ...]) Load data from Google BigQuery.

to_gbq(dataframe, destination_table[, ...]) Write a DataFrame to a Google BigQuery table.

---

**pandas.io.gbq.read_gbq**

**pandas.io.gbq.read_gbq**(query, project_id=None, destination_table=None, index_col=None, col_order=None, **kwargs)

Load data from Google BigQuery.

**Parameters**

- **query**: str
  SQL-Like Query to return data values
- **project_id**: str (optional)
  Google BigQuery Account project ID. Optional, since it may be located in ~/.bigqueryrc
- **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
- **destination_table**: string (optional)
  If provided, send the results to the given table.
- ****kwargs

**Returns**

- **df**: DataFrame
  DataFrame representing results of query

---

**pandas.io.gbq.to_gbq**

**pandas.io.gbq.to_gbq**(dataframe, destination_table, schema=None, col_order=None, if_exists='fail', **kwargs)

Write a DataFrame to a Google BigQuery table.

**Parameters**

- **dataframe**: DataFrame
  DataFrame to be written

---

**THIS IS AN EXPERIMENTAL LIBRARY**

If the table exists, the DataFrame will be appended. If not, a new table will be created, in which case the schema will have to be specified. By default, rows will be written in the order they appear in the DataFrame, though the user may specify an alternative order.
**destination_table** : string

name of table to be written, in the form ‘dataset.tablename’

**schema** : sequence (optional)

list of column types in order for data to be inserted, e.g. ['INTEGER’, ‘TIMESTAMP’, ‘BOOLEAN’]

**col_order** : sequence (optional)

order which columns are to be inserted, e.g. ['primary_key’, ‘birthday’, ‘username’]

**if_exists** : {‘fail’, ‘replace’, ‘append’} (optional)

- 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.

**kwargs are passed to the Client constructor**

**Raises**

* SchemaMissing :
  Raised if the ‘if_exists’ parameter is set to ‘replace’, but no schema is specified

* TableExists :
  Raised if the specified ‘destination_table’ exists but the ‘if_exists’ parameter is set to ‘fail’ (the default)

* InvalidSchema :
  Raised if the ‘schema’ parameter does not match the provided DataFrame

---

### 28.1.10 STATA

**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)

Read Stata file into DataFrame

**Parameters**

* filepath_or_buffer : string or file-like object
  Path to .dta file or object implementing a binary read() functions

* convert_dates : boolean, defaults to True
  Convert date variables to DataFrame time values

* convert_categoricals : boolean, defaults to True
  Read value labels and convert columns to Categorical/Factor variables

* encoding : string, None or encoding
  Encoding used to parse the files. Note that Stata doesn’t support unicode. None defaults to cp1252.

* index : identifier of index column
identifier of column that should be used as index of the DataFrame

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StataReader.data</td>
<td>Reads observations from Stata file, converting them into a dataframe</td>
</tr>
<tr>
<td>StataReader.data_label()</td>
<td>Returns data label of Stata file</td>
</tr>
<tr>
<td>StataReader.value_labels()</td>
<td>Returns a dict, associating each variable name a dict, associating</td>
</tr>
<tr>
<td>StataReader.variable_labels()</td>
<td>Returns variable labels as a dict, associating each variable name</td>
</tr>
<tr>
<td>StataWriter.write_file()</td>
<td></td>
</tr>
</tbody>
</table>

**pandas.io.stata.StataReader.data**

StataReader.data(convert_dates=True, convert_categoricals=True, index=None)

Reads observations from Stata file, converting them into a dataframe

**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

**Returns** y : DataFrame instance

**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()

**28.2 General functions**

**28.2.1 Data manipulations**
### pandas.melt

```
pandas.melt(frame[, id_vars, value_vars, var_name, ...])
```

“Unpivots” a DataFrame from wide format to long format, optionally leaving id variables set.

**Parameters**

- `frame`: DataFrame
  - `id_vars`: tuple, list, or ndarray
  - `value_vars`: tuple, list, or ndarray
  - `var_name`: scalar, if None uses frame.column.name or ‘variable’
  - `value_name`: scalar, default ‘value’
  - `col_level`: scalar, if columns are a MultiIndex then use this level to melt

**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

>>> melt(df, id_vars=['A'], value_vars=['B'])
   A     variable  value
0  a  A       B       1
1  b  B       B       3
2  c  B       B       5

>>> melt(df, id_vars=['A'], value_vars=['B'],
       var_name='myVarname', value_name='myValname')
   A  myVarname  myValname
0  a  B         B       1
1  b  B         B       3
2  c  B         B       5

>>> df.columns = ['list(ABC)', 'list(DEF)']
```
>>> 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

>>> melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
   (A, D)  variable_0  variable_1  value
0  a          B         B         E   1
1  b          B         B         E   3
2  c          B         B         E   5

 pandas.pivot_table

 pandas.pivot_table(data, values=None, rows=None, cols=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
     rows : list of column names or arrays to group on
     Keys to group on the x-axis of the pivot table
     cols : list of column names or arrays to group on
     Keys to group on the y-axis of the pivot table
     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**

 >>> 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
pandas: powerful Python data analysis toolkit, Release 0.13.1

```python
7  bar  two  small  6
8  bar  two  large  7

>>> table = pivot_table(df, values='D', rows=['A', 'B'],
                       cols=['C'], aggfunc=np.sum)
```

```python
table
foo   one  1  4
     two  6  NaN
bar   one  5  4
     two  6  7
```

**pandas.crosstab**

`pandas.crosstab(rows, cols, 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**

- **rows**: array-like, Series, or list of arrays/Series
  
  Values to group by in the rows

- **cols**: 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)
```

```python
crossstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
```

```
  b  one  two
a  c  dull  shiny  dull  shiny
bar  1  2  1  0
foo  2  2  1  2
```

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
  - Labels to use for bin edges, or False to return integer bin labels
- `retbins`: bool, optional
  - Whether to return the bins or not. Can be useful if bins is given as a scalar.

**Returns**

- `out`: Categorical or array of integers if labels is False
  - `bins`: ndarray of floats
    - Returned only if `retbins` is True.

**Notes**

The cut function can be useful for going from a continuous variable to a categorical variable. For example, cut could convert ages to groups of age ranges.

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Categorical object.
Examples

```python
>>> cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, retbins=True)
(array([[0.191, 3.367], [0.191, 3.367], [0.191, 3.367], [3.367, 6.533],
        [6.533, 9.7], [0.191, 3.367]], dtype=object),
 array([ 0.1905 , 3.36666667, 6.53333333, 9.7 ]))
>>> cut(np.ones(5), 4, labels=False)
array([2, 2, 2, 2, 2])
```

**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
  - Labels to use for bin edges, or False to return integer bin labels
- `retbins` : bool, optional
  - Whether to return the bins or not. Can be useful if bins is given as a scalar.

**Returns**

- `cat` : Categorical

**Notes**

Out of bounds values will be NA in the resulting Categorical object

**pandas.merge**

`pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True)`

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

Examples

>>> 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   bar    2.0  bar    6.0
1   bar    2.0  bar    8.0
2   baz    3.0   NaN   NaN
3   foo    1.0   foo    5.0
4   foo    4.0   foo    5.0
5  NaN    NaN  qux    7.0

pandas.concat

pandas.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False)

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*: 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). Any None objects will be dropped silently unless they are all None in which case an Exception 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.

**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)

Convert categorical variable into dummy/indicator variables

**Parameters**  
*data*: array-like or Series

*prefix*: string, default None

---

614 Chapter 28. API Reference
String to append DataFrame column names

**prefix_sep** : string, default '_'

If appending prefix, separator/delimiter to use

**dummy_na** : bool, default False

Add a column to indicate NaNs, if False NaNs are ignored.

**Returns**  **dummies** : DataFrame

**Examples**

```python
>>> 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
```

See also `Series.str.get_dummies`.

### 28.2.2 Top-level missing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isnull</code></td>
<td>Detect missing values (NaN in numeric arrays, None/NaN in object arrays)</td>
</tr>
<tr>
<td><code>notnull</code></td>
<td>Replacement for <code>numpy.isfinite</code> / <code>numpy.isnan</code> 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**  **isnull** : 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.
pandas: powerful Python data analysis toolkit, Release 0.13.1

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 |

Returns

| isnull | 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.

28.2.3 Top-level dealing with datetimes

| to_datetime(arg[, errors, dayfirst, utc, ...]) | Convert argument to datetime |
| to_timedelta(arg[, box, unit]) | Convert argument to timedelta |
| 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 |

pandas.to_datetime

pandas.to_datetime(arg, errors='ignore', dayfirst=False, utc=None, box=True, format=None, 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' |
| dayfirst | boolean, default False |
| utc | boolean, default None |
| box | boolean, default True |
| format | string, default None |
| coerce | force errors to NaT (False by default) |
| unit | unit of the arg (D,s,ms,us,ns) denote the unit in epoch |
| 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

Examples

Take separate series and convert to datetime

```python
>>> 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')
```

Or from strings

```python
>>> df = df.astype(str)
>>> pd.to_datetime(df.day + df.month + df.year, format='%d%m%Y')
```

pandas.to_timedelta

`pandas.to_timedelta(arg, box=True, unit='ns')`
Convert argument to timedelta

Parameters

- **arg**: string, timedelta, array of strings (with possible NAs)
- **box**: boolean, default True
  - If True returns a Series of the results, if False returns ndarray of values
- **unit**: unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer/float number

Returns ret: timedelta64/arrays of timedelta64 if parsing succeeded

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

28.2. General functions
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**

```python
def 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**: `None`
- **end**: `None`
- **periods**: `int`, default `None`
- **freq**: `str/DateOffset`, default `'D'`
- **name**: `str`, default `None`

Returns:
- **prng**: `PeriodIndex`
```

**28.2.4 Top-level evaluation**

```python
def eval(expr[, parser, engine, truediv, ...])
    Evaluate a Python expression as a string using various backends.

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.

28.2.5 Standard moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolling_count</td>
<td>Rolling count of number of non-NaN observations inside provided window</td>
</tr>
<tr>
<td>rolling_sum</td>
<td>Moving sum</td>
</tr>
<tr>
<td>rolling_mean</td>
<td>Moving mean</td>
</tr>
<tr>
<td>rolling_median</td>
<td>O(N log(window)) implementation using skip list</td>
</tr>
<tr>
<td>rolling_var</td>
<td>Unbiased moving variance</td>
</tr>
<tr>
<td>rolling_std</td>
<td>Unbiased moving standard deviation</td>
</tr>
<tr>
<td>rolling_min</td>
<td>Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs.</td>
</tr>
<tr>
<td>rolling_max</td>
<td>Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs.</td>
</tr>
<tr>
<td>rolling_corr</td>
<td>Moving sample correlation</td>
</tr>
<tr>
<td>rolling_corr_pairwise</td>
<td>Computes pairwise rolling correlation matrices as Panel whose items are</td>
</tr>
<tr>
<td>rolling_cov</td>
<td>Unbiased moving covariance</td>
</tr>
<tr>
<td>rolling_skew</td>
<td>Unbiased moving skewness</td>
</tr>
<tr>
<td>rolling_kurt</td>
<td>Unbiased moving kurtosis</td>
</tr>
<tr>
<td>rolling_apply</td>
<td>Generic moving function application</td>
</tr>
<tr>
<td>rolling_quantile</td>
<td>Moving quantile</td>
</tr>
<tr>
<td>rolling_window</td>
<td>Applies a moving window of type window_type and size window</td>
</tr>
</tbody>
</table>
**pandas.rolling_count**

`pandas.rolling_count(arg, window, freq=None, center=False, time_rule=None)`

Rolling count of number of non-NaN observations inside provided window.

**Parameters**
- `arg`: DataFrame or numpy ndarray-like
- `window`: Number of observations used for calculating statistic
- `freq`: None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic
- `center`: boolean, default False
  Whether the label should correspond with center of window
- `time_rule`: Legacy alias for `freq`

**Returns**
- `rolling_count`: type of caller

**pandas.rolling_sum**

`pandas.rolling_sum(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)`

Moving sum

**Parameters**
- `arg`: Series, DataFrame
- `window`: Number of observations used for calculating statistic
- `min_periods`: int
  Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic time_rule is a legacy alias for freq

**Returns**
- `y`: type of input argument

**pandas.rolling_mean**

`pandas.rolling_mean(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)`

Moving mean

**Parameters**
- `arg`: Series, DataFrame
- `window`: Number of observations used for calculating statistic
- `min_periods`: int
  Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic time_rule is a legacy alias for freq

**Returns**
- `y`: type of input argument
pandas.rolling_median

```python
def pandas.rolling_median(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs):
    O(N log(window)) implementation using skip list
```

Moving median

**Parameters**

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **time_rule**: is a legacy alias for freq

**Returns**

- **y**: type of input argument

pandas.rolling_var

```python
def pandas.rolling_var(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs):
    Unbiased moving variance
```

Unbiased moving variance

**Parameters**

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **time_rule**: is a legacy alias for freq

**Returns**

- **y**: type of input argument

pandas.rolling_std

```python
def pandas.rolling_std(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs):
    Unbiased moving standard deviation
```

Unbiased moving standard deviation

**Parameters**

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **time_rule**: is a legacy alias for freq

**Returns**

- **y**: type of input argument
pandas.rolling_min

pandas.rolling_min(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)
Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs. Moving minimum

Parameters
arg : Series, DataFrame
    window : Number of observations used for calculating statistic
    min_periods : int
        Minimum number of observations in window required to have a value
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
time_rule is a legacy alias for freq

Returns
y : type of input argument

pandas.rolling_max

pandas.rolling_max(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)
Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs. Moving maximum

Parameters
arg : Series, DataFrame
    window : Number of observations used for calculating statistic
    min_periods : int
        Minimum number of observations in window required to have a value
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
time_rule is a legacy alias for freq

Returns
y : type of input argument

pandas.rolling_corr

pandas.rolling_corr(arg1, arg2, window, min_periods=None, freq=None, center=False, time_rule=None)
Moving sample correlation

Parameters
arg1 : Series, DataFrame, or ndarray
arg2 : Series, DataFrame, or ndarray
    window : Number of observations used for calculating statistic
    min_periods : int
        Minimum number of observations in window required to have a value
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
time_rule is a legacy alias for freq

Returns
y : type depends on inputs
    DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
    Computes result for each column Series / Series -> Series
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.rolling_corr_pairwise**

```python
pandas.rolling_corr_pairwise(df, window, min_periods=None)
```
Computes pairwise rolling correlation matrices as Panel whose items are dates

**Parameters**

- **df**: DataFrame
- **window**: int
- **min_periods**: int, default None

**Returns**

- **correls**: Panel

**pandas.rolling_cov**

```python
pandas.rolling_cov(arg1, arg2, window, min_periods=None, freq=None, center=False, time_rule=None)
```
Unbiased moving covariance

**Parameters**

- **arg1**: Series, DataFrame, or ndarray
- **arg2**: Series, DataFrame, or ndarray
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
- **freq**: None or string alias / date offset object, default=None
- **center**: False
- **time_rule**: None

**Returns**

- **y**: type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns)
  - DataFrame / Series -> computes result for each column
  - Series / Series -> Series

**pandas.rolling_skew**

```python
pandas.rolling_skew(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)
```
Unbiased moving skewness

**Parameters**

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
- **freq**: None or string alias / date offset object, default=None
- **center**: False
- **time_rule**: None

**Returns**

- **y**: type of input argument
**pandas.rolling_kurt**

`pandas.rolling_kurt(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)`

Unbiased moving kurtosis

**Parameters**
- `arg`: Series, DataFrame
- `window`: Number of observations used for calculating statistic
- `min_periods`: int
  - Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- `time_rule`: Legacy alias for `freq`

**Returns**
- `y`: type of input argument

**pandas.rolling_apply**

`pandas.rolling_apply(arg, window, func, min_periods=None, freq=None, center=False, time_rule=None)`

Generic moving function application

**Parameters**
- `arg`: Series, DataFrame
- `window`: Number of observations used for calculating statistic
- `func`: function
  - Must produce a single value from an ndarray input
- `min_periods`: int
  - Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- `center`: boolean, default False
  - Whether the label should correspond with center of window
- `time_rule`: Legacy alias for `freq`

**Returns**
- `y`: type of input argument

**pandas.rolling_quantile**

`pandas.rolling_quantile(arg, window, quantile, min_periods=None, freq=None, center=False, time_rule=None)`

Moving quantile

**Parameters**
- `arg`: Series, DataFrame
- `window`: Number of observations used for calculating statistic
- `quantile`: 0 <= quantile <= 1
- `min_periods`: int
  - Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic

center : boolean, default False
    Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns  y : type of input argument

pandas.rolling_window

pandas.rolling_window (arg, window=None, win_type=None, min_periods=None, freq=None, center=False, mean=True, time_rule=None, axis=0, **kwargs)
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
        Minimum number of observations in window required to have a value.

    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic

    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

    time_rule : Legacy alias for freq

    axis : {0, 1}, default 0

Returns  y : type of input argument

Notes

The recognized window 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).

28.2.6 Standard expanding window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expanding_count</td>
<td>Expanding count of number of non-NaN observations.</td>
</tr>
<tr>
<td>expanding_sum</td>
<td>Expanding sum</td>
</tr>
<tr>
<td>expanding_mean</td>
<td>Expanding mean</td>
</tr>
<tr>
<td>expanding_median</td>
<td>O(N log(window)) implementation using skip list</td>
</tr>
<tr>
<td>expanding_var</td>
<td>Unbiased expanding variance</td>
</tr>
<tr>
<td>expanding_std</td>
<td>Unbiased expanding standard deviation</td>
</tr>
<tr>
<td>expanding_min</td>
<td>Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs</td>
</tr>
<tr>
<td>expanding_max</td>
<td>Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs</td>
</tr>
<tr>
<td>expanding_corr</td>
<td>Expanding sample correlation</td>
</tr>
<tr>
<td>expanding_corr_pairwise</td>
<td>Computes pairwise expanding correlation matrices as Panel whose items are</td>
</tr>
<tr>
<td>expanding_cov</td>
<td>Unbiased expanding covariance</td>
</tr>
<tr>
<td>expanding_skew</td>
<td>Unbiased expanding skewness</td>
</tr>
<tr>
<td>expanding_kurt</td>
<td>Unbiased expanding kurtosis</td>
</tr>
<tr>
<td>expanding_apply</td>
<td>Generic expanding function application</td>
</tr>
<tr>
<td>expanding_quantile</td>
<td>Expanding quantile</td>
</tr>
</tbody>
</table>

pandas.expanding_count

pandas.expanding_count( arg[, freq=None, center=False, time_rule=None])

Expanding count of number of non-NaN observations.

Parameters
arg : DataFrame or numpy ndarray-like
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic
center : boolean, default False
Whether the label should correspond with center of window
time_rule : Legacy alias for freq

Returns
expanding_count : type of caller

pandas.expanding_sum

pandas.expanding_sum( arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Expanding sum
**Parameters**

- **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

**Returns**

- **y**: type of input argument

---

**pandas.expanding_mean**

```
pandas.expanding_mean(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```

Expanding mean

**Parameters**

- **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

**Returns**

- **y**: type of input argument

---

**pandas.expanding_median**

```
pandas.expanding_median(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```

O(N log(window)) implementation using skip list

Expanding median

**Parameters**

- **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

**Returns**

- **y**: type of input argument

---

**pandas.expanding_var**

```
pandas.expanding_var(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```

Unbiased expanding variance

**Parameters**

- **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic
Returns $y$ : type of input argument

**pandas.expanding_std**

```python
pandas.expanding_std(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```
Unbiased expanding standard deviation

**Parameters**
- **arg** : Series, DataFrame
  - **min_periods** : int
    Minimum number of observations in window required to have a value
  - **freq** : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic

**Returns** $y$ : type of input argument

**pandas.expanding_min**

```python
pandas.expanding_min(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```
Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs. Expanding minimum

**Parameters**
- **arg** : Series, DataFrame
  - **min_periods** : int
    Minimum number of observations in window required to have a value
  - **freq** : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic

**Returns** $y$ : type of input argument

**pandas.expanding_max**

```python
pandas.expanding_max(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```
Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs. Expanding maximum

**Parameters**
- **arg** : Series, DataFrame
  - **min_periods** : int
    Minimum number of observations in window required to have a value
  - **freq** : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic

**Returns** $y$ : type of input argument

**pandas.expanding_corr**

```python
pandas.expanding_corr(arg1, arg2, min_periods=1, freq=None, center=False, time_rule=None)
```
Expanding sample correlation

---

28.2. General functions
Parameters  **arg1** : Series, DataFrame, or ndarray

**arg2** : Series, DataFrame, or ndarray

**min_periods** : int

Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=\None

Frequency to conform to before computing statistic

Returns  **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
Computes result for each column Series / Series -> Series

**pandas.expanding_corr_pairwise**

**pandas.expanding_corr_pairwise** *(df, min_periods=1)*

Computes pairwise expanding correlation matrices as Panel whose items are dates

Parameters  **df** : DataFrame

**min_periods** : int, default 1

Returns  **correls** : Panel

**pandas.expanding_cov**

**pandas.expanding_cov** *(arg1, arg2, min_periods=\1, freq=\None, center=\False, time_rule=\None)*

Unbiased expanding covariance

Parameters  **arg1** : Series, DataFrame, or ndarray

**arg2** : Series, DataFrame, or ndarray

**min_periods** : int

Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=\None

Frequency to conform to before computing statistic

Returns  **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
Computes result for each column Series / Series -> Series

**pandas.expanding_skew**

**pandas.expanding_skew** *(arg, min_periods=1, freq=\None, center=\False, time_rule=\None, **kwargs)*

Unbiased expanding skewness

Parameters  **arg** : Series, DataFrame

**min_periods** : int

Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=\None
pandas: powerful Python data analysis toolkit, Release 0.13.1

Frequency to conform to before computing statistic

**Returns**  \( y \) : type of input argument

### pandas.expanding_kurt

**Syntax**
```
pandas.expanding_kurt(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)
```

Unbiased expanding kurtosis

**Parameters**
- **arg** : Series, DataFrame
- **min_periods** : int
  - Minimum number of observations in window required to have a value
- **freq** : None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

**Returns**  \( y \) : type of input argument

### pandas.expanding_apply

**Syntax**
```
pandas.expanding_apply(arg, func, min_periods=1, freq=None, center=False, time_rule=None)
```

Generic expanding function application

**Parameters**
- **arg** : Series, DataFrame
- **func** : function
  - Must produce a single value from an ndarray input
- **min_periods** : int
  - Minimum number of observations in window required to have a value
- **freq** : None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **center** : boolean, default False
  - Whether the label should correspond with center of window
- **time_rule** : Legacy alias for freq

**Returns**  \( y \) : type of input argument

### pandas.expanding_quantile

**Syntax**
```
pandas.expanding_quantile(arg, quantile, min_periods=1, freq=None, center=False, time_rule=None)
```

Expanding quantile

**Parameters**
- **arg** : Series, DataFrame
- **quantile** : 0 <= quantile <= 1
- **min_periods** : int
  - Minimum number of observations in window required to have a value
- **freq** : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic

**center**: boolean, default False

Whether the label should correspond with center of window

**time_rule**: Legacy alias for freq

**Returns** y: type of input argument

### 28.2.7 Exponentially-weighted moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ewma</code></td>
<td>Exponentially-weighted moving average</td>
</tr>
<tr>
<td><code>ewmstd</code></td>
<td>Exponentially-weighted moving std</td>
</tr>
<tr>
<td><code>ewmvar</code></td>
<td>Exponentially-weighted moving variance</td>
</tr>
<tr>
<td><code>ewmcorr</code></td>
<td>Exponentially-weighted moving correlation</td>
</tr>
<tr>
<td><code>ewmcov</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, time_rule=None, adjust=True)*

Exponentially-weighted moving average

**Parameters**

- **arg**: Series, DataFrame
  - **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, :math: \( \alpha = 1 - \exp(\log(0.5) / halflife) \)
  - **min_periods**: int, default 0
  - Number of observations in sample to require (only affects beginning)
  - **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic time_rule is a legacy alias for freq
  - **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)

**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 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.

**pandas.ewmstd**

```python
pandas.ewmstd(arg, com=None, span=None, halflife=None, min_periods=0, bias=False, time_rule=None)
```

Exponentially-weighted moving std

**Parameters**

- `arg`: Series, DataFrame
  - `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, :math: \alpha = 1 - \exp(\log(0.5) / halflife)
  - `min_periods`: int, default 0
    - Number of observations in sample to require (only affects beginning)
  - `freq`: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic `time_rule` is a legacy alias for `freq`
  - `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)
  - `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 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.

**pandas.ewmvar**

```python
pandas.ewmvar(arg, com=None, span=None, halflife=None, min_periods=0, bias=False, freq=None, time_rule=None)
```

Exponentially-weighted moving variance

**Parameters**

- `arg`: Series, DataFrame
  - `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, \( \alpha = 1 - \exp(\log(0.5) / halflife) \)

min_periods : int, default 0
    Number of observations in sample to require (only affects beginning)

freq : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic time_rule is a legacy alias for freq

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)

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 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.

pandas.ewmcorr

pandas.ewmcorr(arg1, arg2, com=None, span=None, halflife=None, min_periods=0, freq=None, time_rule=None)
    Exponentially-weighted moving correlation

Parameters arg1 : Series, DataFrame, or ndarray
    arg2 : Series, DataFrame, or ndarray

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, \( \alpha = 1 - \exp(\log(0.5) / halflife) \)

min_periods : int, default 0
    Number of observations in sample to require (only affects beginning)

freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

**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)

**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 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.

**pandas.ewmcov**

`pandas.ewmcov(arg1, arg2, com=None, span=None, halflife=None, min_periods=0, bias=False, freq=None, time_rule=None)`

Exponentially-weighted moving covariance

**Parameters**

arg1 : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

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

Number of observations in sample to require (only affects beginning)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

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)

**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 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 $c = \frac{s - 1}{2}$.

So a “20-day EWMA” would have center 9.5.

### 28.3 Series

#### 28.3.1 Constructor

```python
Series([data, index, dtype, name, copy, ...])  # One-dimensional ndarray with axis labels (including time series).
```

**pandas.Series**

```python
class pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
```

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be 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**

- **T**
  Support for compatibility

- **at**

- **axes**

- **base**

- **blocks**
  Internal property, property synonym for as_blocks()

- **data**

- **dtype**

- **dtypes**
  For compat

- **empty**
  True if NDFrame is entirely empty [no items]

- **flags**

- **ftype**

Continued on next page
Table 28.21 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftypes</td>
<td>for compat</td>
</tr>
<tr>
<td>iat</td>
<td></td>
</tr>
<tr>
<td>iloc</td>
<td></td>
</tr>
<tr>
<td>imag</td>
<td></td>
</tr>
<tr>
<td>is_time_series</td>
<td></td>
</tr>
<tr>
<td>ix</td>
<td></td>
</tr>
<tr>
<td>loc</td>
<td></td>
</tr>
<tr>
<td>ndim</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td></td>
</tr>
<tr>
<td>shape</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>strides</td>
<td></td>
</tr>
<tr>
<td>values</td>
<td>Return Series as ndarray</td>
</tr>
<tr>
<td>weekday</td>
<td></td>
</tr>
</tbody>
</table>

pandas.Series.T

Series.T
support for compatibility

pandas.Series.at

Series.at

pandas.Series.axes

Series.axes

pandas.Series.base

Series.base

pandas.Series.blocks

Series.blocks
Internal property, property synonym for as_blocks()

pandas.Series.data

Series.data

pandas.Series.dtype

Series.dtype
pandas.Series.dtypes

Series.dtypes
for compat

pandas.Series.empty

Series.empty
True if NDFrame is entirely empty [no items]

pandas.Series.flags

Series.flags

pandas.Series.ftype

Series.ftype

pandas.Series.ftypes

Series.ftypes
for compat

pandas.Series.iat

Series.iat

pandas.Series.iloc

Series.iloc

pandas.Series.imag

Series.imag

pandas.Series.is_time_series

Series.is_time_series

pandas.Series.ix

Series.ix
pandas.Series.loc

Series.loc

pandas.Series.ndim

Series.ndim

pandas.Series.real

Series.real

pandas.Series.shape

Series.shape

pandas.Series.size

Series.size

pandas.Series.strides

Series.strides

pandas.Series.values

Series.values

Return Series as ndarray

Returns arr: numpy.ndarray

pandas.Series.weekday

Series.weekday

<table>
<thead>
<tr>
<th>is_copy</th>
<th>str</th>
</tr>
</thead>
</table>

Methods

abs() Return an object with absolute value taken.

add(other[, level, fill_value, axis]) Binary operator add with support to substitute a fill_value for missing data

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
Table 28.22 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all([axis, out])</td>
<td>Returns True if all elements evaluate to True.</td>
</tr>
<tr>
<td>any([axis, out])</td>
<td>Returns True if any of the elements of a evaluate to True.</td>
</tr>
<tr>
<td>append(to_append[, verify_integrity])</td>
<td>Concatenate two or more Series. The indexes must not overlap.</td>
</tr>
<tr>
<td>apply(func[, convert_dtype, args])</td>
<td>Invoke function on values of Series. Can be ufunc (a NumPy function.</td>
</tr>
<tr>
<td>argmax([axis, out, skipna])</td>
<td>Index of first occurrence of maximum of values.</td>
</tr>
<tr>
<td>argmin([axis, out, skipna])</td>
<td>Index of first occurrence of minimum of values.</td>
</tr>
<tr>
<td>argsort([axis, kind, order])</td>
<td>Overrides ndarray.argsort.</td>
</tr>
<tr>
<td>as_blocks([columns])</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has.</td>
</tr>
<tr>
<td>as_matrix([columns])</td>
<td>Convert the frame to its Numpy-array matrix representation. Columns</td>
</tr>
<tr>
<td>asfreq(freq[, method, how, normalize])</td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset</td>
</tr>
<tr>
<td>asof(where)</td>
<td>Return last good (non-NaN) value in TimeSeries if value is NaN for</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.,</td>
</tr>
<tr>
<td>bool()</td>
<td>Return the bool of a single element PandasObject</td>
</tr>
<tr>
<td>clip([lower, upper, out])</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>clip_lower(threshold)</td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>clip_upper(threshold)</td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td>combine(other, func[, fill_value])</td>
<td>Perform elementwise binary operation on two Series using given function</td>
</tr>
<tr>
<td>combine_first(other)</td>
<td>Combine Series values, choosing the calling Series’s values</td>
</tr>
<tr>
<td>compound([axis, skipna, level])</td>
<td>Return the compound percentage of the values for the requested axis</td>
</tr>
<tr>
<td>consolidate([implace])</td>
<td>Compute NDFrame with “consolidated” internals (data of each dtype.</td>
</tr>
<tr>
<td>convert_objects([convert_dates, ...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>corr(other[, method, min_periods])</td>
<td>Compute correlation with other Series, excluding missing values</td>
</tr>
<tr>
<td>cov(other[, min_periods])</td>
<td>Compute covariance with Series, excluding missing values</td>
</tr>
<tr>
<td>cummax([axis, dtype, out, skipna])</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>cummin([axis, dtype, out, skipna])</td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td>cumprod([axis, dtype, out, skipna])</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>cumsum([axis, dtype, out, skipna])</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>describe([percentile_width])</td>
<td>Generate various summary statistics of Series, excluding NaN</td>
</tr>
<tr>
<td>diff([periods])</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>div(other[, level, fill_value, axis])</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>divide(other[, level, fill_value, axis])</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>dot(other)</td>
<td>Matrix multiplication with DataFrame or inner-product with Series</td>
</tr>
<tr>
<td>drop(labels[, axis, level, inplace])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>drop_duplicates([take_last, inplace])</td>
<td>Return Series with duplicate values removed</td>
</tr>
<tr>
<td>dropna([axis, inplace])</td>
<td>Return Series without null values</td>
</tr>
<tr>
<td>duplicated([take_last])</td>
<td>Return boolean Series denoting duplicate values</td>
</tr>
<tr>
<td>eq(other)</td>
<td>Determines if two NDFrame objects contain the same elements. NaNs in the</td>
</tr>
<tr>
<td>equals(other)</td>
<td></td>
</tr>
<tr>
<td>ffill([axis, inplace, limit, downcast])</td>
<td>Synonym for NDFrame.fillna(method=’ffill’).</td>
</tr>
<tr>
<td>fillna([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td>filter([items, like, regex, axis])</td>
<td>Restrict the info axis to set of items or wildcard</td>
</tr>
<tr>
<td>first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>first_valid_index()</td>
<td>Return label for first non-NA/null value</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>floordiv</td>
<td>Binary operator floordiv with support to substitute a fill_value for missing data.</td>
</tr>
<tr>
<td>from_array</td>
<td>Read delimited file into Series.</td>
</tr>
<tr>
<td>get</td>
<td>Returns value occupying requested label, default to specified missing value if no such label.</td>
</tr>
<tr>
<td>dtypes</td>
<td>Return the counts of dtypes in this object.</td>
</tr>
<tr>
<td>ftypes</td>
<td>Return the counts of ftypes in this object.</td>
</tr>
<tr>
<td>value</td>
<td>Quickly retrieve single value at passed index label.</td>
</tr>
<tr>
<td>values</td>
<td>Same as values (but handles sparseness conversions); is a view.</td>
</tr>
<tr>
<td>groupby</td>
<td>Group series using mapper (dict or key function, apply given function.</td>
</tr>
<tr>
<td>head</td>
<td>Returns first n rows.</td>
</tr>
<tr>
<td>hist</td>
<td>Draw histogram of the input series using matplotlib.</td>
</tr>
<tr>
<td>idxmax</td>
<td>Index of first occurrence of maximum of values.</td>
</tr>
<tr>
<td>idxmin</td>
<td>Index of first occurrence of minimum of values.</td>
</tr>
<tr>
<td>iget</td>
<td>Return the i-th value or values in the Series by location.</td>
</tr>
<tr>
<td>iget_value</td>
<td>Return the i-th value or values in the Series by location.</td>
</tr>
<tr>
<td>interpolate</td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td>irow</td>
<td>Return the i-th value or values in the Series by location.</td>
</tr>
<tr>
<td>isnan</td>
<td>Return a boolean Series showing whether each element is null.</td>
</tr>
<tr>
<td>item</td>
<td>Returns a boolean same-sized object indicating if the values are null.</td>
</tr>
<tr>
<td>iteritems</td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td>itertuples</td>
<td>alias used to get around 2to3. Deprecated.</td>
</tr>
<tr>
<td>keys</td>
<td>Alias for index.</td>
</tr>
<tr>
<td>kurt</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>kurtosis</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>last</td>
<td>Return label for last non-NA/null value.</td>
</tr>
<tr>
<td>load</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>lt</td>
<td>Return the mean absolute deviation of the values for the requested axis.</td>
</tr>
<tr>
<td>map</td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td>mask</td>
<td>Returns copy whose values are replaced with nan if the.</td>
</tr>
<tr>
<td>max</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>mean</td>
<td>Return the mean of the values for the requested axis.</td>
</tr>
<tr>
<td>median</td>
<td>Return the median of the values for the requested axis.</td>
</tr>
<tr>
<td>min</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td>mod</td>
<td>Binary operator mod with support to substitute a fill_value for missing data.</td>
</tr>
<tr>
<td>mode</td>
<td>Returns the mode(s) of the dataset.</td>
</tr>
<tr>
<td>mul</td>
<td>Binary operator mul with support to substitute a fill_value for missing data.</td>
</tr>
<tr>
<td>nonzero</td>
<td>numpy like, returns same as nonzero.</td>
</tr>
<tr>
<td>notnull</td>
<td>Return a boolean same-sized object indicating if the values are</td>
</tr>
<tr>
<td>unique</td>
<td>Return count of unique elements in the Series.</td>
</tr>
<tr>
<td>order</td>
<td>Sorts Series object, by value, maintaining index-value link.</td>
</tr>
<tr>
<td>pct_change</td>
<td>Percent change over given number of periods.</td>
</tr>
<tr>
<td>plot</td>
<td>Plot the input series with the index on the x-axis using matplotlib.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pop(item)</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>pow(other[, level, fill_value, axis])</td>
<td>Binary operator pow with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>prod((axis, skipna, level, numeric_only))</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>product((axis, skipna, level, numeric_only))</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>ptp((axis, out))</td>
<td></td>
</tr>
<tr>
<td>put(*args, **kwargs)</td>
<td>Return value at the given quantile, a la scoreatpercentile in</td>
</tr>
<tr>
<td>rad2d(other[, level, fill_value, axis])</td>
<td>Binary operator rad2d with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>rank((method, na_option, ascending))</td>
<td>Compute data ranks (1 through n).</td>
</tr>
<tr>
<td>ravel((order))</td>
<td></td>
</tr>
<tr>
<td>rdiv(other[, level, fill_value, axis])</td>
<td>Binary operator rtruediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>reindex((index))</td>
<td>Conform Series to new index with optional filling logic, placing</td>
</tr>
<tr>
<td>reindex_axis(labels[, axis])</td>
<td>for compatibility with higher dims</td>
</tr>
<tr>
<td>reindex_like(other[, method, copy, limit])</td>
<td>return an object with matching indices to myself</td>
</tr>
<tr>
<td>rename((index))</td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td>rename_axis(mapping[, axis, copy, inplace])</td>
<td>Alter index and / or columns using input function or functions.</td>
</tr>
<tr>
<td>reorder_levels(order)</td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td>repeat(reps)</td>
<td>See ndarray.repeat</td>
</tr>
<tr>
<td>replace(to_replace, value, inplace, limit, ...)</td>
<td>Replace values given in 'to_replace' with 'value'.</td>
</tr>
<tr>
<td>resample(rule[, how, axis, fill_method, ...])</td>
<td>Convenience method for frequency conversion and resampling of regular time series</td>
</tr>
<tr>
<td>reset_index([level, drop, name, inplace])</td>
<td>Analogous to the pandas.DataFrame.reset_index() function, see</td>
</tr>
<tr>
<td>reshape(*args, **kwargs)</td>
<td>See numpy.ndarray.reshape</td>
</tr>
<tr>
<td>rfloordiv(other[, level, fill_value, axis])</td>
<td>Binary operator rfloordiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>rmod(other[, level, fill_value, axis])</td>
<td>Binary operator rmod with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>rmul(other[, level, fill_value, axis])</td>
<td>Binary operator rmul with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>round(decimals, out)</td>
<td>Return $a$ with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>rpow(other[, level, fill_value, axis])</td>
<td>Binary operator rpow with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>rsub(other[, level, fill_value, axis])</td>
<td>Binary operator rsub with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>rtruediv(other[, level, fill_value, axis])</td>
<td>Binary operator rtruediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>save(path)</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>select(crit[, axis])</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td>set_value(label, value)</td>
<td>Quickly set single value at passed label.</td>
</tr>
<tr>
<td>shift(periods[, freq, axis])</td>
<td>Shift index by desired number of periods with an optional time freq</td>
</tr>
<tr>
<td>skew((axis, skipna, level, numeric_only))</td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td>sort((axis, kind, order, ascending))</td>
<td>Sort values and index labels by value, in place.</td>
</tr>
<tr>
<td>sort_index([ascending])</td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td>squeeze()</td>
<td>squeeze length 1 dimensions</td>
</tr>
<tr>
<td>std((axis, skipna, level, ddof))</td>
<td>Return unbiased standard deviation over requested axis</td>
</tr>
<tr>
<td>sub(other[, level, fill_value, axis])</td>
<td>Binary operator sub with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>subtract(other[, level, fill_value, axis])</td>
<td>Binary operator sub with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>sum((axis, skipna, level, numeric_only))</td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td>swapaxes(axis1, axis2[, copy])</td>
<td>Interchange axes and swap values axes appropriately</td>
</tr>
<tr>
<td>swaplevel(i, j[, copy])</td>
<td>Swap levels i and j in a MultiIndex</td>
</tr>
<tr>
<td>tail(n)</td>
<td>Returns last n rows</td>
</tr>
<tr>
<td>take(indices[, axis, convert])</td>
<td>Analogous to ndarray.take, return Series corresponding to requested</td>
</tr>
<tr>
<td>to_clipboard([excel, sep])</td>
<td>Attempt to write text representation of object to the system clipboard</td>
</tr>
<tr>
<td>to_csv(path[, index, sep, na_rep, ...])</td>
<td>Write Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>to_dense()</td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td>to_dict()</td>
<td>Convert Series to {label -&gt; value} dict</td>
</tr>
</tbody>
</table>
Table 28.22 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_frame(name)</code></td>
<td>Convert Series to DataFrame</td>
</tr>
<tr>
<td><code>to_hdf(path_or_buf, key, **kwargs)</code></td>
<td>Activate the HDFStore.</td>
</tr>
<tr>
<td><code>to_json(path_or_buf, orient, date_format, ...)</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>to_msgpack(path_or_buf)</code></td>
<td>Msgpack (serialize) object to input file path.</td>
</tr>
<tr>
<td><code>to_period(freq, copy)</code></td>
<td>Convert TimeSeries from DatetimeIndex to PeriodIndex with desired.</td>
</tr>
<tr>
<td><code>to_pickle(path)</code></td>
<td>Pickle (serialize) object to input file path.</td>
</tr>
<tr>
<td><code>to_sparse(kind, fill_value)</code></td>
<td>Convert Series to SparseSeries.</td>
</tr>
<tr>
<td><code>to_string(buf, na_rep, float_format, ...)</code></td>
<td>Render a string representation of the Series.</td>
</tr>
<tr>
<td><code>to_timestamp(freq, how, copy)</code></td>
<td>Cast to datetimeindex of timestamps, at beginning of period.</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Convert Series to a nested list</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Support for compatibility</td>
</tr>
<tr>
<td><code>truediv(other[, level, fill_value, axis])</code></td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td><code>truncate(before, after, axis, copy)</code></td>
<td>Truncates a sorted NDFrame before and/or after some particular.</td>
</tr>
<tr>
<td><code>tshift(periods, freq, axis)</code></td>
<td>Shift the time index, using the index's frequency if available.</td>
</tr>
<tr>
<td><code>tz_convert(tz[, copy])</code></td>
<td>Convert TimeSeries to target time zone</td>
</tr>
<tr>
<td><code>tz_localize(tz[, copy, infer_dst])</code></td>
<td>Localize tz-naive TimeSeries to target time zone.</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Return array of unique values in the Series. Significantly faster than.</td>
</tr>
<tr>
<td><code>unstack([level])</code></td>
<td>Unstack, a.k.a.</td>
</tr>
<tr>
<td><code>update(other)</code></td>
<td>Modify Series in place using non-NA values from passed</td>
</tr>
<tr>
<td><code>valid(inplace)</code></td>
<td>Modify Series in place using non-NA values from passed</td>
</tr>
<tr>
<td><code>value_counts([normalize, sort, ascending, bins])</code></td>
<td>Returns Series containing counts of unique values. The resulting Series</td>
</tr>
<tr>
<td><code>var(axis, skipna, level, ddof)</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>view([dtype])</code></td>
<td>Modify Series in place using non-NA values from passed</td>
</tr>
<tr>
<td><code>where(cond[, other, inplace, axis, level, ...])</code></td>
<td>Return an object of same shape as self and whose corresponding</td>
</tr>
<tr>
<td><code>xs(key[, axis, level, drop_level])</code></td>
<td>Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.Series.abs**

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)

Binary operator add 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
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 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 name
    Broadcast across a level, matching Index values on the passed MultiIndex level

Parameters  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(axis=None, out=None)
Returns True if all elements evaluate to True.
Refer to numpy.all for full documentation.
See Also:

numpy.all equivalent function

pandas.Series.any

Series.any(axis=None, out=None)
Returns True if any of the elements of a evaluate to True.
Refer to numpy.any for full documentation.
See Also:

numpy.any equivalent function

pandas.Series.append

Series.append(to_append, verify_integrity=False)
Concatenate two or more Series. The indexes must not overlap

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 minimum of values

See Also:
- DataFrame.idxmax

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

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
pandas.Series.as_blocks

Series.as_blocks(columns=None)
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
are presented in sorted order unless a specific list of columns is provided.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters  columns : array-like
Specific column order

Returns  values : a list of Object

pandas.Series.as_matrix

Series.as_matrix(columns=None)
Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

NOTE: 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,float32 -> float32 float16,float32,float64 -> float64 int32,uint8 -> int32

Returns  values : ndarray
If the caller is heterogeneous and contains booleans or objects, the result will be of
dtype=object

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', 'ffill', '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)

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

**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 autocorrelation

**Returns**

- autocorr: float

**pandas.Series.between**

Series.between(left, right, inclusive=True)

Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

**Parameters**

- left: scalar
  - Left boundary
- right: scalar
  - Right boundary

**Returns**

- is_between: Series
pandas.Series.between_time

**Series.between_time** *(start_time, end_time, include_start=True, include_end=True)*
Select values between particular times of the day (e.g., 9:00-9:30 AM)

**Parameters**
- **start_time**: datetime.time or string
- **end_time**: datetime.time or string
- **include_start**: boolean, default True
- **include_end**: boolean, default True

**Returns**
- **values_between_time**: type of caller

pandas.Series.bfill

**Series.bfill** *(axis=0, inplace=False, limit=None, downcast=None)*
Synonym for 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.clip

**Series.clip** *(lower=None, upper=None, out=None)*
Trim values at input threshold(s)

**Parameters**
- **lower**: float, default None
- **upper**: float, default None

**Returns**
- **clipped**: Series

pandas.Series.clip_lower

**Series.clip_lower** *(threshold)*
Return copy of the input with values below given value truncated

**Returns**
- **clipped**: same type as input

See Also:
- clip

pandas.Series.clip_upper

**Series.clip_upper** *(threshold)*
Return copy of input with values above given value truncated

**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, **kwargs)
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, 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.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**

```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`: if True, attempt to soft convert dates, if ‘coerce’, force conversion (and non-convertibles get NaT)
- `convert_numeric`: if True attempt to coerce to numbers (including strings), non-convertibles get NaN
- `convert_timedeltas`: if True, attempt to soft convert timedeltas, if ‘coerce’, force conversion (and non-convertibles get NaT)
- `copy`: Boolean, if True, return copy, default is True

**Returns**
- `converted`: asm as input object

**pandas.Series.copy**

```python
Series.copy(deep=True)
```

Make a copy of this object

**Parameters**
- `deep`: boolean, default True
  - Make a deep copy, i.e. also copy data

**Returns**
- `copy`: type of caller

**pandas.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, 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=50)

Generate various summary statistics of Series, excluding NaN values. These include: count, mean, std, min, max, and lower%/50%/upper% percentiles

- **Parameters**
  - **percentile_width**: float, optional
    
    width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

**Returns**

- **desc**: 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)

Binary operator truediv 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

**pandas.Series.divide**

*Series.divide* (*other, level=None, fill_value=None, axis=0*)  
Binary operator truediv 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)  
level : int or name  
Broadcast across a level, matching Index values on the passed MultiIndex level  

**Returns**  
result : Series

**pandas.Series.dot**

*Series.dot* (*other*)  
Matrix multiplication with DataFrame or inner-product with Series objects

**Parameters**  
other : Series or DataFrame

**Returns**  
dot_product : scalar or Series

**pandas.Series.drop**

*Series.drop* (*labels, axis=0, level=None, inplace=False, **kwargs*)  
Return new object with labels in requested axis removed

**Parameters**  
labels : single label or list-like  
axis : int or axis name  
level : int or name, default None  
For MultiIndex  
inplace : bool, default False  
If True, do operation inplace and return None.

**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.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.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)

**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.ffill**

Series.ffill (axis=0, inplace=False, limit=None, downcast=None)

Synonym for NDFrame.fillna(method='ffill')

**pandas.Series.fillna**

Series.fillna (value=None, method=None, axis=0, inplace=False, limit=None, downcast=None)

Fill NA/NaN values using the specified method

**Parameters** 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

value : scalar, dict, or Series
Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which
value to use for each index (for a Series) or column (for a DataFrame). (values not in
the dict/Series will not be filled). This value cannot be a list.

axis : {0, 1}, default 0
  • 0: fill column-by-column
  • 1: fill row-by-row

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
  Maximum size gap to forward or backward fill

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 : same type as caller

See Also:
  reindex, asfreq

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

Notes
Arguments are mutually exclusive, but this is not checked for

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)`

Binary operator floordiv 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

**pandas.Series.from_array**

**classmethod** `Series.from_array(arr, index=None, name=None, copy=False, fastpath=False)`

**pandas.Series.from_csv**

**classmethod** `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.ge

Series.ge(other)

pandas.Series.get

Series.get(label, default=None)

Returns value occupying requested label, default to specified missing value if not present. Analogous to dict.get

Parameters  label : object

Label value looking for

default : object, optional

Value to return if label not in index

Returns  y : scalar

pandas.Series.get_dtype_counts

Series.get_dtype_counts()

Return the counts of dtypes in this object

pandas.Series.get_ftype_counts

Series.get_ftype_counts()

Return the counts of ftypes in this object

pandas.Series.get_value

Series.get_value(label)

Quickly retrieve single value at passed index label

Parameters  index : label

Returns  value : scalar value

pandas.Series.get_values

Series.get_values()

same as values (but handles sparseness conversions); is a view
Series.groupby

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**

```python
# DataFrame result >>> data.groupby(func, axis=0).mean()
# DataFrame result >>> data.groupby([‘col1’, ‘col2’])[‘col3’].mean()
# DataFrame with hierarchical index >>> data.groupby([‘col1’, ‘col2’]).mean()
```

Series.gt

Series.gt(other)

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, **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
- **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 minimum of values

**See Also**:
- DataFrame.idxmax
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`

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='infer', **kwargs)`

Interpolate values according to different methods.

**Parameters**
- `method` : {'linear', 'time', 'values', 'index', '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': 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 ‘infer’
* 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 2 2 3 3 dtype: float64

pandas.Series.irow

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

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**

Series.isnull()  
Return a boolean same-sized object indicating if the values are null

**pandas.Series.item**

Series.item()

**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 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, 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 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, 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)

pandas.Series.load

Series.load(path)
Deprecated. Use read_pickle instead.

pandas.Series.lt

Series.lt(other)

pandas.Series.mad

Series.mad(axis=None, skipna=None, level=None, **kwargs)
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, 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**

\[ y : \text{Series} \]

same index as caller

**Examples**

```python
>>> 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)`

Returns copy whose values are replaced with nan if the inverted condition is True

**Parameters**

\[ \text{cond} : \text{boolean NDFrame or array} \]

**Returns**

\[ \text{wh} : \text{same as input} \]

**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**

\[ \text{axis} : \{\text{index (0)}\} \]

\[ \text{skipna} : \text{boolean, default True} \]

Excluding NA/null values. If an entire row/column is NA, the result will be NA

\[ \text{level} : \text{int, default None} \]

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

\[ \text{numeric_only} : \text{boolean, default None} \]

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns**

\[ \text{max} : \text{scalar or Series (if level specified)} \]
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, 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, 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, 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)
Binary operator mod 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

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)
Binary operator mul 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
pandas.Series.multiply

Series.multiply(other, level=None, fill_value=None, axis=0)

Binary operator mul 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

pandas.Series.ne

Series.ne(other)

pandas.Series.nonzero

Series.nonzero()
    numpy like, returns same as nonzero

pandas.Series.notnull

Series.notnull()
    Return a boolean same-sized object indicating if the values are not null

pandas.Series.nunique

Series.nunique()
    Return count of unique elements in the Series

    Returns

    unique: int

pandas.Series.order

Series.order(na_last=True, ascending=True, kind='mergesort')
    Sorts Series object, by value, maintaining index-value link

    Parameters

    na_last: boolean (optional, default=True)
        Put NaN’s at beginning or end

    ascending: boolean, default True
        Sort ascending. Passing False sorts descending

    kind: {'mergesort', 'quicksort', 'heapsort'}, default ‘mergesort’
        Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm
**pandas.Series.pct_change**

Series.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwds)

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**: same type as caller

**pandas.Series.plot**

Series.plot(series, label=None, kind='line', use_index=True, rot=None, xticks=None, yticks=None, xlim=None, ylim=None, ax=None, style=None, grid=None, legend=False, logx=False, logy=False, secondary_y=False, **kwds)

Plot the input series with the index on the x-axis using matplotlib

Parameters

- **label**: label argument to provide to plot
- **kind**: {'line', 'bar', 'barh', 'kde', 'density'}
  - bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estimation plot
- **use_index**: boolean, default True
  - Plot index as axis tick labels
- **rot**: int, default None
  - Rotation for tick labels
- **xticks**: sequence
  - Values to use for the xticks
- **yticks**: sequence
  - Values to use for the yticks
- **xlim**: 2-tuple/list
- **ylim**: 2-tuple/list
- **ax**: matplotlib axis object
  - If not passed, uses gca()
- **style**: string, default matplotlib default
matplotlib line style to use

**grid** : matplotlib grid

**legend** : matplotlib legend

**logx** : boolean, default False
   For line plots, use log scaling on x axis

**logy** : boolean, default False
   For line plots, use log scaling on y axis

**secondary_y** : boolean or sequence of ints, default False
   If True then y-axis will be on the right

**figsize** : a tuple (width, height) in inches

**kwds** : keywords
   Options to pass to matplotlib plotting method

---

**Notes**

See matplotlib documentation online for more on this subject

---

**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)
```

Binary operator pow 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

---

**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, 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.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, 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)*

### pandas.Series.quantile

**Series.quantile** *(q=0.5)*

Return value at the given quantile, a la scoreatpercentile in scipy.stats

**Parameters**

- **q** : quantile
  
  0 <= q <= 1

**Returns**  
**quantile** : float
**pandas.Series.radd**

**Series.radd**(other, level=None, fill_value=None, axis=0)

Binary operator radd 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

**pandas.Series.rank**

**Series.rank**(method='average', na_option='keep', ascending=True)

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'}  
  - 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
- **na_option**: {'keep'}  
  keep: leave NA values where they are
- **ascending**: boolean, default True  
  False for ranks by high (1) to low (N)

**Returns**

ranks: Series

**pandas.Series.ravel**

**Series.ravel**(order='C')

**pandas.Series.rdiv**

**Series.rdiv**(other, level=None, fill_value=None, axis=0)

Binary operator rtruediv 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

`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**: {'backfill', 'bfill', 'pad', 'ffill', None}, default None
  Method to use for filling holes in reindexed DataFrame
  - *pad / ffill*: propagate last valid observation forward to next valid
  - *backfill / bfill*: use NEXT valid observation 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
- **takeable**: boolean, default False
  treat the passed as positional values

**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 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.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

```python
Series.repeat(reps)
```

See `ndarray.repeat`

### 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 from a DataFrame). Returns the caller if this is True.

**limit**: int, default None

Maximum size gap to forward or backward fill

**regex**: bool or same types as `to_replace`, default False

Whether to interpret `to_replace` and/or `value` as regular expressions. If this is True then `to_replace` must be a string. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method**: string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a list.

Returns **filled**: NDFrame

Raises  **AssertionError**

- If `regex` is not a bool and `to_replace` is not None.

**TypeError**

- If `to_replace` is a dict and `value` is not a list, dict, ndarray, or Series

- If `to_replace` is None and `regex` is not compilable into a regular expression or is a list, dict, ndarray, or Series.

**ValueError**

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

See Also:

NDFrame.reindex, NDFrame.asfreq, NDFrame.fillna

**Notes**

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers are strings, then you can do this.

- This method has a lot of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

**Series.resample**

Series.resample *(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, offset=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

closed : ['right', 'left']

Which side of bin interval is closed

label : ['right', 'left']

Which bin edge label to label bucket with

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.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)

See numpy.ndarray.reshape
**pandas.Series.rfloordiv**

Series.rfloordiv(other, level=None, fill_value=None, axis=0)

Binary operator rfloordiv 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

**pandas.Series.rmod**

Series.rmod(other, level=None, fill_value=None, axis=0)

Binary operator rmod 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

**pandas.Series.rmul**

Series.rmul(other, level=None, fill_value=None, axis=0)

Binary operator rmul 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

**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.
pandas.Series.rpow

Series.rpow(other, level=None, fill_value=None, axis=0)
Binary operator rpow 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

pandas.Series.rsub

Series.rsub(other, level=None, fill_value=None, axis=0)
Binary operator rsub 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

pandas.Series.rtruediv

Series.rtruediv(other, level=None, fill_value=None, axis=0)
Binary operator rtruediv 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
pandas.Series.save

Series.save(path)
Deprecated. Use to_pickle instead

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.set_value

Series.set_value(label, value)
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

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, **kwds)
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’)

Returns
- shifted : same type as caller

Notes
If freq is specified then the index values are shifted but the data if not realigned
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)}
    Exclude NA/null values. If an entire row/column is NA, the result will be NA
skipna : boolean, default True
    If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing
    into a scalar
level : int, default None
    Include only float, int, boolean data. If None, will attempt to use everything, then
    use only numeric data
numeric_only : boolean, default None

Returns

skew : scalar or Series (if level specified)

pandas.Series.sort

Series.sort (axis=0, kind='quicksort', order=None, ascending=True)
Sort values and index labels by value, in place. For compatibility with ndarray API. No return value

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
ascending : boolean, default True
    Sort ascending. Passing False sorts descending

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

>>> result1 = s.sort_index(ascending=False)
>>> result2 = s.sort_index(ascending=[1, 0])
pandas.Series.sortlevel

Series.sortlevel(level=0, ascending=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
- ascending : bool, default True

Returns
- sorted : Series

pandas.Series.squeeze

Series.squeeze()
squeeze length 1 dimensions

pandas.Series.std

Series.std(axis=None, skipna=None, level=None, ddof=1, **kwargs)
Return unbiased standard deviation 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, 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
- stdev : scalar or Series (if level specified)

pandas.Series.sub

Series.sub(other, level=None, fill_value=None, axis=0)
Binary operator sub 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
**pandas.Series.subtract**

Series.subtract(other, level=None, fill_value=None, axis=0)

Binary operator sub 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

**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, 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)
Analogous to ndarray.take, return Series corresponding to requested indices

Parameters
- **indices**: list / array of ints
- **convert**: translate negative to positive indices (default)

Returns
- **taken**: Series

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)
Write Series to a comma-separated values (csv) file

Parameters
- **path**: string file path or file handle / StringIO
- **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.

### 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** Write-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_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
pandas: powerful Python data analysis toolkit, Release 0.13.1

- split : dict like {index -> [index], columns -> [columns], data -> [values]}
- records : dict 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
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_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_string

Series.to_string(buf=None, na_rep='NaN', float_format=None, nanRep=None, length=False, dtype=False, name=False)
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
    length : boolean, default False
        Add the Series length
    dtype : boolean, default False
        Add the Series dtype
    name : boolean, default False
        Add the Series name (which may be None)

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()**  
support for compatibility

`pandas.Series.truediv`

**Series.truediv(other, level=None, fill_value=None, axis=0)**  
Binary operator truediv 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

`pandas.Series.truncate`

**Series.truncate(before=None, after=None, axis=None, copy=True)**  
Truncates a sorted NDataFrame 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**

```python
Series.tshift(periods=1, freq=None, axis=0, **kwds)
```
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**

```python
Series.tz_convert(tz, copy=True)
```
Convert TimeSeries to target time zone

**Parameters**
- `tz`: string or pytz.timezone object
- `copy`: boolean, default True
  - Also make a copy of the underlying data

**Returns**
- `converted`: TimeSeries

**pandas.Series.tz_localize**

```python
Series.tz_localize(tz, copy=True, infer_dst=False)
```
Localize tz-naive TimeSeries to target time zone Entries will retain their “naive” value but will be annotated as being relative to the specified tz.

After localizing the TimeSeries, you may use tz_convert() to get the Datetime values recomputed to a different tz.

**Parameters**
- `tz`: string or pytz.timezone object
- `copy`: boolean, default True
  - Also make a copy of the underlying data
- `infer_dst`: boolean, default False
  - Attempt to infer fall dst-transition hours based on order

**Returns**
- `localized`: TimeSeries
pandas.Series.unique

Series.unique()  
Return array of unique values in the Series. Significantly faster than numpy.unique

Returns uniques : ndarray

pandas.Series.unstack

Series.unstack(level=-1)  
Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame

Parameters level : int, string, or list of these, default last level
  Level(s) to unstack, can pass level name

Returns unstacked : DataFrame

Examples

```python
>>> 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)  
Returns Series containing counts of unique values. The resulting Series will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values
Parameters **normalize** : boolean, default False

   If True then the Series 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

Returns **counts** : Series

**pandas.Series.var**

Series.var *(axis=None, skipna=None, level=None, ddof=1, **kwargs)*

   Return unbiased variance 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, 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 **variance** : 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
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=True, 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, default True
    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

Examples

>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
   A  B  C
  a  4  5  2
Name: a
>>> df.xs('C', axis=1)
   A  B  C
a  2
b  9
c  3
Name: C
>>> s = df.xs('a', copy=False)
>>> s['A'] = 100
>>> df
  A  B  C
a  100  5  2
b   4  0  9
c   9  7  3

>>> 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
tree  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

28.3.2 Attributes and underlying data

Axes

• index: axis labels

<table>
<thead>
<tr>
<th>pandas.Series.values</th>
<th>Return Series as ndarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Series.dtype</td>
<td></td>
</tr>
<tr>
<td>pandas.Series.fctype</td>
<td></td>
</tr>
</tbody>
</table>

**Series.values**

Series.values
Return Series as ndarray

| Returns | arr : numpy.ndarray |

**Series.dtype**

Series.dtype

**Series.fctype**

Series.fctype

28.3. Series
28.3.3 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</td>
</tr>
</tbody>
</table>

**pandas.Series.astype**

Series.astype (dtype, copy=True, raise_on_error=True)  
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

**Returns**
- casted : type of caller

**pandas.Series.copy**

Series.copy (deep=True)  
Make a copy of this object

**Parameters**
- deep : boolean, 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

**pandas.Series.notnull**

Series.notnull()  
Return a boolean same-sized object indicating if the values are not null

28.3.4 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.get(label[, default])</td>
<td>Returns value occupying requested label, default to specified missing value if not present.</td>
</tr>
<tr>
<td>Series.at</td>
<td></td>
</tr>
<tr>
<td>Series.iat</td>
<td></td>
</tr>
<tr>
<td>Series.ix</td>
<td></td>
</tr>
<tr>
<td>Series.loc</td>
<td></td>
</tr>
<tr>
<td>Series.iloc</td>
<td></td>
</tr>
<tr>
<td>Series.<strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Series.iteritems()</td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
</tbody>
</table>

Chapter 28. API Reference
pandas.Series.get

Series.get(label, default=None)
Returns value occupying requested label, default to specified missing value if not present. Analogous to dict.get

Parameters
- label : object
  Label value looking for
- default : object, optional
  Value to return if label not in index

Returns
- y : scalar

pandas.Series.at

Series.at

pandas.Series.iat

Series.iat

pandas.Series.ix

Series.ix

pandas.Series.loc

Series.loc

pandas.Series.iloc

Series.iloc

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.
28.3.5 Binary operator functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.add</td>
<td>Binary operator add with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.sub</td>
<td>Binary operator sub with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.mul</td>
<td>Binary operator mul with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.div</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.floordiv</td>
<td>Binary operator floordiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.mod</td>
<td>Binary operator mod with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.pow</td>
<td>Binary operator pow with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.radd</td>
<td>Binary operator radd with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rsub</td>
<td>Binary operator rsub with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rmul</td>
<td>Binary operator rmul with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rdiv</td>
<td>Binary operator rtruediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rfloordiv</td>
<td>Binary operator rfloordiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rmod</td>
<td>Binary operator rmod with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.rpow</td>
<td>Binary operator rpow with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.combine</td>
<td>Perform elementwise binary operation on two Series using given function</td>
</tr>
<tr>
<td>Series.combine_first</td>
<td>Combine Series values, choosing the calling Series’s values</td>
</tr>
<tr>
<td>Series.round</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>Series.lt</td>
<td></td>
</tr>
<tr>
<td>Series.gt</td>
<td></td>
</tr>
<tr>
<td>Series.le</td>
<td></td>
</tr>
<tr>
<td>Series.ge</td>
<td></td>
</tr>
<tr>
<td>Series.ne</td>
<td></td>
</tr>
<tr>
<td>Series.eq</td>
<td></td>
</tr>
</tbody>
</table>

**pandas.Series.add**

```python
Series.add(other[, level, fill_value, axis])
```

Binary operator add 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

**pandas.Series.sub**

```python
Series.sub(other[, level, fill_value, axis])
```

Binary operator sub 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

pandas.Series.mul

Series.mul (other, level=None, fill_value=None, axis=0)
Binary operator mul 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

pandas.Series.div

Series.div (other, level=None, fill_value=None, axis=0)
Binary operator truediv 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

pandas.Series.truediv

Series.truediv (other, level=None, fill_value=None, axis=0)
Binary operator truediv 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

### pandas.Series.floordiv

**Series.floordiv** *(other, level=``None`` , fill_value=``None`` , axis=``0``)*  
Binary operator floordiv 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

### pandas.Series.mod

**Series.mod** *(other, level=``None`` , fill_value=``None`` , axis=``0``)*  
Binary operator mod 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

### pandas.Series.pow

**Series.pow** *(other, level=``None`` , fill_value=``None`` , axis=``0``)*  
Binary operator pow 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
**pandas.Series.radd**

Series.radd(other, level=None, fill_value=None, axis=0)

Binary operator radd 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

**pandas.Series.rsub**

Series.rsub(other, level=None, fill_value=None, axis=0)

Binary operator rsub 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

**pandas.Series.rmul**

Series.rmul(other, level=None, fill_value=None, axis=0)

Binary operator rmul 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

**pandas.Series.rdiv**

Series.rdiv(other, level=None, fill_value=None, axis=0)

Binary operator rtruediv 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

**pandas.Series.rtruediv**

Series.%s (other, level=None, fill_value=None, axis=0)

Binary operator rtruediv 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

**pandas.Series.rfloordiv**

Series.%s (other, level=None, fill_value=None, axis=0)

Binary operator rfloordiv 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

**pandas.Series.rmod**

Series.%s (other, level=None, fill_value=None, axis=0)

Binary operator rmod 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
**pandas.Series.rpow**

Series.rpow(other, level=None, fill_value=None, axis=0)

Binary operator rpow 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

**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)
pandas.Series.gt

Series.gt(other)

pandas.Series.le

Series.le(other)

pandas.Series.ge

Series.ge(other)

pandas.Series.ne

Series.ne(other)

pandas.Series.eq

Series.eq(other)

28.3.6 Function application, GroupBy

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.apply</td>
<td>Invoke function on values of Series. Can be ufunc (a NumPy function)</td>
</tr>
<tr>
<td>Series.map</td>
<td>Map values of Series using input correspondence (which can be</td>
</tr>
<tr>
<td></td>
<td>Python function that only works on single values</td>
</tr>
<tr>
<td>Series.groupby</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, args])

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

**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
given example
>>> 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.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

```python
# DataFrame result >>> data.groupby(func, axis=0).mean()
# DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()
# DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()
```

## 28.3.7 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.abs()</code></td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td><code>Series.any(axis, out)</code></td>
<td>Returns True if any of the elements of a evaluate to True.</td>
</tr>
<tr>
<td><code>Series.autocorr()</code></td>
<td>Lag-1 autocorrelation</td>
</tr>
<tr>
<td><code>Series.between(left, right[, inclusive])</code></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right. NA values</td>
</tr>
<tr>
<td><code>Series.clip(lower, upper, out)</code></td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td><code>Series.clip_lower(threshold)</code></td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td><code>Series.clip_upper(threshold)</code></td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td><code>Series.corr(other[, method, min_periods])</code></td>
<td>Compute correlation with other Series, excluding missing values</td>
</tr>
<tr>
<td><code>Series.count([level])</code></td>
<td>Return number of non-NA/null observations in the Series</td>
</tr>
<tr>
<td><code>Series.cov(other, min_periods)</code></td>
<td>Compute covariance with Series, excluding missing values</td>
</tr>
<tr>
<td><code>Series.cummax([axis, dtype, out, skipna])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>Series.cummin([axis, dtype, out, skipna])</code></td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td><code>Series.cumprod([axis, dtype, out, skipna])</code></td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td><code>Series.cumsum([axis, dtype, out, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td><code>Series.describe([percentile_width])</code></td>
<td>Generate various summary statistics of Series, excluding NaN</td>
</tr>
<tr>
<td><code>Series.diff([periods])</code></td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td><code>Series.kurt([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td><code>Series.mad([axis, skipna, level])</code></td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.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>Series.mean([axis, skipna, level, numeric_only])</code></td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.median([axis, skipna, level, ...])</code></td>
<td>Return the median of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Series.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>Series.mode()</code></td>
<td>Returns the mode(s) of the dataset.</td>
</tr>
<tr>
<td><code>Series.nunique()</code></td>
<td>Return count of unique elements in the Series</td>
</tr>
<tr>
<td><code>Series.pct_change([periods, fill_method, ...])</code></td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td><code>Series.prod([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.quantile([q])</code></td>
<td>Return value at the given quantile, a la scoreatpercentile in</td>
</tr>
<tr>
<td><code>Series.skew([axis, skipna, level, numeric_only])</code></td>
<td>Compute data ranks (1 through n).</td>
</tr>
<tr>
<td><code>Series.std([axis, skipna, level, ddof])</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>Series.sum([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased standard deviation over requested axis</td>
</tr>
<tr>
<td><code>Series.unique()</code></td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.var([axis, skipna, level, ddof])</code></td>
<td>Return array of unique values in the Series. Significantly faster than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.abs()</code></td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td><code>Series.any(axis, out)</code></td>
<td>Returns True if any of the elements of a evaluate to True.</td>
</tr>
<tr>
<td><code>Series.autocorr()</code></td>
<td>Lag-1 autocorrelation</td>
</tr>
<tr>
<td><code>Series.between(left, right[, inclusive])</code></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right. NA values</td>
</tr>
<tr>
<td><code>Series.clip(lower, upper, out)</code></td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td><code>Series.clip_lower(threshold)</code></td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td><code>Series.clip_upper(threshold)</code></td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td><code>Series.corr(other[, method, min_periods])</code></td>
<td>Compute correlation with other Series, excluding missing values</td>
</tr>
<tr>
<td><code>Series.count([level])</code></td>
<td>Return number of non-NA/null observations in the Series</td>
</tr>
<tr>
<td><code>Series.cov(other, min_periods)</code></td>
<td>Compute covariance with Series, excluding missing values</td>
</tr>
<tr>
<td><code>Series.cummax([axis, dtype, out, skipna])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>Series.cummin([axis, dtype, out, skipna])</code></td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td><code>Series.cumprod([axis, dtype, out, skipna])</code></td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td><code>Series.cumsum([axis, dtype, out, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td><code>Series.describe([percentile_width])</code></td>
<td>Generate various summary statistics of Series, excluding NaN</td>
</tr>
<tr>
<td><code>Series.diff([periods])</code></td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td><code>Series.kurt([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td><code>Series.mad([axis, skipna, level])</code></td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.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>Series.mean([axis, skipna, level, numeric_only])</code></td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.median([axis, skipna, level, ...])</code></td>
<td>Return the median of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Series.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>Series.mode()</code></td>
<td>Returns the mode(s) of the dataset.</td>
</tr>
<tr>
<td><code>Series.nunique()</code></td>
<td>Return count of unique elements in the Series</td>
</tr>
<tr>
<td><code>Series.pct_change([periods, fill_method, ...])</code></td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td><code>Series.prod([axis, skipna, level, numeric_only])</code></td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.quantile([q])</code></td>
<td>Return value at the given quantile, a la scoreatpercentile in</td>
</tr>
<tr>
<td><code>Series.skew([axis, skipna, level, numeric_only])</code></td>
<td>Compute data ranks (1 through n).</td>
</tr>
<tr>
<td><code>Series.std([axis, skipna, level, ddof])</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>Series.sum([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased standard deviation over requested axis</td>
</tr>
<tr>
<td><code>Series.unique()</code></td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td><code>Series.var([axis, skipna, level, ddof])</code></td>
<td>Return array of unique values in the Series. Significantly faster than</td>
</tr>
</tbody>
</table>
Table 28.28 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.value_counts()</code></td>
<td>Returns Series containing counts of unique values. The resulting Series</td>
</tr>
<tr>
<td><code>Series.abs()</code></td>
<td>Return an object with absolute value taken. Only applicable to objects that are all numeric</td>
</tr>
<tr>
<td><code>Series.any()</code></td>
<td>Returns True if any of the elements evaluate to True. Refer to numpy.any for full documentation.</td>
</tr>
<tr>
<td><code>Series.autocorr()</code></td>
<td>Lag-1 autocorrelation</td>
</tr>
<tr>
<td><code>Series.between()</code></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right. NA values will be treated as False</td>
</tr>
<tr>
<td><code>Series.clip()</code></td>
<td>Trim values at input threshold(s)</td>
</tr>
</tbody>
</table>

28.3. Series
pandas.Series.clip_lower

Series.clip_lower(threshold)
Return copy of the input with values below given value truncated

Returns clipped : same type as input

See Also:
clip

pandas.Series.clip_upper

Series.clip_upper(threshold)
Return copy of input with values above given value truncated

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, 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=50)
Generate various summary statistics of Series, excluding NaN values. These include: count, mean, std, min, max, and lower%/50%/upper% percentiles

Parameters  
percentile_width : float, optional
width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

Returns  
desc : 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.kurt

Series.kurt (axis= None, skipna= None, level= None, numeric_only= None, **kwargs)
Return unbiased kurtosis 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, 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(axis=None, skipna=None, level=None, **kwargs)
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, default None
If the axis is aMultiIndex (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, 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, 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, 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, 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 \texttt{sort=False} is not defined.

\textbf{Returns} modes : Series (sorted)

\textbf{pandas.Series.nunique}

\textbf{Series.nunique()}

Return count of unique elements in the Series

\textbf{Returns} nunique : int

\textbf{pandas.Series.pct_change}

\textbf{Series.pct_change(periods=1, fill_method='pad', limit=None, freq=None, \textit{**kwds})}

Percent change over given number of periods

\textbf{Parameters} \textbf{periods} : int, default 1

Periods to shift for forming percent change

\textbf{fill_method} : str, default ‘pad’

How to handle NAs before computing percent changes

\textbf{limit} : int, default None

The number of consecutive NAs to fill before stopping

\textbf{freq} : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. ‘M’ or BDay())

\textbf{Returns} chg : same type as caller

\textbf{pandas.Series.prod}

\textbf{Series.prod(axis=None, skipna=None, level=None, numeric_only=None, \textit{**kwds})}

Return the product of the values for the requested axis

\textbf{Parameters} \textbf{axis} : \{index (0)\}

\textbf{skipna} : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

\textbf{level} : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

\textbf{numeric_only} : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

\textbf{Returns} prod : scalar or Series (if level specified)
**pandas.Series.quantile**

Series.quantile(q=0.5)

Return value at the given quantile, a la scoreatpercentile in scipy.stats

**Parameters**
- q : quantile
- 0 <= q <= 1

**Returns**
- quantile : float

**pandas.Series.rank**

Series.rank(method='average', na_option='keep', ascending=True)

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'}
  - 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
- na_option : {'keep'}
  - keep: leave NA values where they are
- ascending : boolean, default True
  - False for ranks by high (1) to low (N)

**Returns**
- ranks : Series

**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, 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, **kwargs)

Return unbiased standard deviation 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, 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 stdev : 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, 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.unique

Series.unique()

Return array of unique values in the Series. Significantly faster than numpy.unique

Returns uniques : ndarray

pandas.Series.var

Series.var(axis=None, skipna=None, level=None, ddof=1, **kwargs)

Return unbiased variance 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, 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**
- **variance**: scalar or Series (if level specified)

### pandas.Series.value_counts

**Series.value_counts** *(normalize=False, sort=True, ascending=False, bins=None)*

Returns Series containing counts of unique values. The resulting Series will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values

**Parameters**
- **normalize**: boolean, default False
  - If True then the Series 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

**Returns**
- **counts**: Series

### 28.3.8 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.align</strong> <em>(other[, join, axis, level, ...]</em>)</td>
<td>Align two object on their axes with the</td>
</tr>
<tr>
<td><strong>Series.drop</strong> <em>(labels[, axis, level, inplace]</em>)</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td><strong>Series.first</strong> <em>(offset)</em></td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td><strong>Series.head</strong> <em>(n)</em></td>
<td>Returns first n rows</td>
</tr>
<tr>
<td><strong>Series.idxmax</strong> <em>(axis, out, skipna)</em></td>
<td>Index of first occurrence of maximum of values.</td>
</tr>
<tr>
<td><strong>Series.idxmin</strong> <em>(axis, out, skipna)</em></td>
<td>Index of first occurrence of minimum of values.</td>
</tr>
<tr>
<td><strong>Series.isin</strong> <em>(values)</em></td>
<td>Return a boolean Series showing whether each element</td>
</tr>
<tr>
<td><strong>Series.last</strong> <em>(offset)</em></td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td><strong>Series.reindex</strong> <em>(index)</em></td>
<td>Conform Series to new index with optional filling logic, placing</td>
</tr>
<tr>
<td><strong>Series.reindex_like</strong> <em>(other[, method, copy, limit]</em>)</td>
<td>Return an object with matching indices to myself</td>
</tr>
<tr>
<td><strong>Series.rename</strong> <em>(index)</em></td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><strong>Series.reset_index</strong> <em>(level, drop, name, inplace)</em></td>
<td>Analogous to the pandas.DataFrame.reset_index() function, see</td>
</tr>
<tr>
<td><strong>Series.select</strong> <em>(crit[, axis]</em>)</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td><strong>Series.take</strong> <em>(indices[, axis, convert]</em>)</td>
<td>Analogous to ndarray.take, return Series corresponding to requested</td>
</tr>
<tr>
<td><strong>Series.tail</strong> <em>(n)</em></td>
<td>Returns last n rows</td>
</tr>
</tbody>
</table>

Continued on next page
**pandas.Series.truncate**

Series.truncate([before, after, axis, copy])
Truncates a sorted NDFrame before and/or after some particular

**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 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 name
    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, **kwargs)
Return new object with labels in requested axis removed

**Parameters**
- **labels**: single label or list-like
  - **axis**: int or axis name
  - **level**: int or name, default None
    For MultiIndex
  - **inplace**: bool, default False
    If True, do operation inplace and return None.

**Returns**
- **dropped**: type of caller
**pandas.Series.first**

Series.\texttt{first}(\texttt{offset})

Convenience method for subsetting initial periods of time series data based on a date offset

- **Parameters**  \texttt{offset} : string, DateOffset, dateutil.relativedelta
- **Returns**  \texttt{subset} : type of caller

**Examples**

ts.last(\texttt{‘10D’}) -> First 10 days

**pandas.Series.head**

Series.\texttt{head}(n=5)

Returns first \texttt{n} rows

**pandas.Series.idxmax**

Series.\texttt{idxmax}(axis=None, \texttt{out}=None, \texttt{skipna}=True)

Index of first occurrence of maximum of values.

- **Parameters**  \texttt{skipna} : boolean, default True
  
  Exclude NA/null values

- **Returns**  \texttt{idxmax} : Index of minimum of values

**See Also:**

DataFrame.\texttt{idxmax}

**Notes**

This method is the Series version of ndarray.\texttt{argmax}.

**pandas.Series.idxmin**

Series.\texttt{idxmin}(axis=None, \texttt{out}=None, \texttt{skipna}=True)

Index of first occurrence of minimum of values.

- **Parameters**  \texttt{skipna} : boolean, default True
  
  Exclude NA/null values

- **Returns**  \texttt{idxmin} : Index of minimum of values

**See Also:**

DataFrame.\texttt{idxmin}

**Notes**

This method is the Series version of ndarray.\texttt{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**: {'backfill', 'bfill', 'pad', 'ffill', None}, default None

  Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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

- **takeable**: boolean, default False

  treat the passed as positional values

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.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)

Analogous to ndarray.take, return Series corresponding to requested indices

---

28.3. Series
Parameters  indices : list / array of ints

   convert : translate negative to positive indices (default)

   Returns  taken : Series

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

28.3.9 Missing data handling

Series.dropna(\(axis, inplace\)) Return Series without null values

Series.fillna(\(value, method, axis, ...\)) Fill NA/NaN values using the specified method

Series.interpolate(\(method, axis, limit, ...\)) Interpolate values according to different methods.

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(\(value=None, method=None, axis=0, inplace=False, limit=None, downcast=None\))

   Fill NA/NaN values using the specified method

   Parameters  method : ['backfill', 'ffill', 'pad', 'bfill', None], default None
Method to use for filling holes in reindexed Series 

| pad / ffill | propagate last valid observation forward to next valid |
| bfill | use NEXT valid observation to fill gap |

**value**

- scalar, dict, or Series
- Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series will not be filled). This value cannot be a list.

**axis**

- {0, 1}, default 0
  - 0: fill column-by-column
  - 1: fill row-by-row

**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
  - Maximum size gap to forward or backward fill

**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 : same type as caller

See Also:

- reindex, asfreq

**pandas.Series.interpolate**

```
Series.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast='infer', **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’: 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:
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 ‘infer’
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

28.3.10 Reshaping, sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.argsort(axis, kind, order)</td>
<td>Overrides ndarray.argsort.</td>
</tr>
<tr>
<td>Series.order(na_last, ascending, kind)</td>
<td>Sorts Series object, by value, maintaining index-value link.</td>
</tr>
<tr>
<td>Series.reorder_levels(order)</td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td>Series.sort(axis, kind, order, ascending)</td>
<td>Sort values and index labels by value, in place.</td>
</tr>
<tr>
<td>Series.sort_index(level)</td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td>Series.sortlevel(i, j, copy)</td>
<td>Sort Series with MultiIndex by chosen level. Data will be</td>
</tr>
<tr>
<td>Series.swaplevel(i, j)</td>
<td>Swap levels i and j in a MultiIndex</td>
</tr>
<tr>
<td>Series.unstack(level)</td>
<td>Unstack, a.k.a.</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’
Order 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=True, ascending=True, kind='mergesort')
Sorts Series object, by value, maintaining index-value link

Parameters
- na_last : boolean (optional, default=True)
  Put NaN’s at beginning or end
- ascending : boolean, default True
  Sort ascending. Passing False sorts descending
- kind : {'mergesort', 'quicksort', 'heapsort'}, default 'mergesort'
  Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm

Returns y : Series

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, kind='quicksort', order=None, ascending=True)
Sort values and index labels by value, in place. For compatibility with ndarray API. No return value

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
- ascending : boolean, default True
  Sort ascending. Passing False sorts descending

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 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

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

Parameters level : int, string, or list of these, default last level

Level(s) to unstack, can pass level name

Returns unstacked : DataFrame

Examples

```python
>>> 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.

28.3.11 Combining / joining / merging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.append</td>
<td>Concatenate two or more Series. The indexes must not overlap</td>
</tr>
<tr>
<td>Series.replace</td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
<tr>
<td>Series.update</td>
<td>Modify Series in place using non-NA values from passed</td>
</tr>
</tbody>
</table>

pandas.Series.append

Series.append (to_append[, verify_integrity])
Concatenate two or more Series. The indexes must not overlap

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

Series.replace ([to_replace, value, inplace, ...])
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.
- **value**: any
- **inplace**: boolean, default False
- **limit**: int
- **regex**: boolean, default False

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

### 28.3.12 Time series-related

**Series.asfreq**

Series.asfreq(freq[, method, how, 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

**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**

```python
Series.shift(periods=1, freq=None, axis=0, **kwds)
```

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')

**Returns**

- **shifted**: same type as caller

Notes

If freq is specified then the index values are shifted but the data if not realigned

**pandas.Series.first_valid_index**

```python
Series.first_valid_index()
```

Return label for first non-NA/null value

**pandas.Series.last_valid_index**

```python
Series.last_valid_index()
```

Return label for last non-NA/null value

**pandas.Series.weekday**

```python
Series.weekday
```

**pandas.Series.resample**

```python
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

closed

label

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.Series.tz_convert

Series.tz_convert (tz, copy=True)
 Convert TimeSeries to target time zone

Parameters tz : string or pytz.timezone object

copy : boolean, default True
Also make a copy of the underlying data

Returns converted : TimeSeries

pandas.Series.tz_localize

Series.tz_localize (tz, copy=True, infer_dst=False)
 Localize tz-naive TimeSeries to target time zone Entries will retain their “naive” value but will be annotated as being relative to the specified tz.
After localizing the TimeSeries, you may use tz_convert() to get the Datetime values recomputed to a different tz.

Parameters tz : string or pytz.timezone object

copy : boolean, default True
Also make a copy of the underlying data

infer_dst : boolean, default False
Attempt to infer fall dst-transition hours based on order

Returns localized : TimeSeries
## 28.3.13 String handling

Series.str can be used to access the values of the series as strings and apply several methods to it. Due to implementation details the methods show up here as methods of the `StringMethods` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringMethods.cat</code></td>
<td>Concatenate arrays of strings with given separator</td>
</tr>
<tr>
<td><code>StringMethods.center</code></td>
<td>“Center” strings, filling left and right side with additional whitespace</td>
</tr>
<tr>
<td><code>StringMethods.contains</code></td>
<td>Check whether given pattern is contained in each string in the array</td>
</tr>
<tr>
<td><code>StringMethods.count</code></td>
<td>Count occurrences of pattern in each string</td>
</tr>
<tr>
<td><code>StringMethods.decode</code></td>
<td>Decode character string to unicode using indicated encoding</td>
</tr>
<tr>
<td><code>StringMethods.encode</code></td>
<td>Encode character string to some other encoding using indicated encoding</td>
</tr>
<tr>
<td><code>StringMethods.endswith</code></td>
<td>Return boolean array indicating whether each string ends with passed pattern</td>
</tr>
<tr>
<td><code>StringMethods.extract</code></td>
<td>Find groups in each string using passed regular expression</td>
</tr>
<tr>
<td><code>StringMethods.findall</code></td>
<td>Find all occurrences of pattern or regular expression</td>
</tr>
<tr>
<td><code>StringMethods.get</code></td>
<td>Extract element from lists, tuples, or strings in each element in the array</td>
</tr>
<tr>
<td><code>StringMethods.join</code></td>
<td>Join lists contained as elements in array, a la <code>str.join</code></td>
</tr>
<tr>
<td><code>StringMethods.len</code></td>
<td>Compute length of each string in array.</td>
</tr>
<tr>
<td><code>StringMethods.lower</code></td>
<td>Convert strings in array to lowercase</td>
</tr>
<tr>
<td><code>StringMethods.lstrip</code></td>
<td>Strip whitespace (including newlines) from left side of each string in the array</td>
</tr>
<tr>
<td><code>StringMethods.match</code></td>
<td>Deprecated: Find groups in each string using passed regular expression.</td>
</tr>
<tr>
<td><code>StringMethods.pad</code></td>
<td>Pad strings with whitespace</td>
</tr>
<tr>
<td><code>StringMethods.repeat</code></td>
<td>Duplicate each string in the array by indicated number of times</td>
</tr>
<tr>
<td><code>StringMethods.replace</code></td>
<td>Replace</td>
</tr>
<tr>
<td><code>StringMethods.rstrip</code></td>
<td>Strip whitespace (including newlines) from right side of each string in the array</td>
</tr>
<tr>
<td><code>StringMethods.slice</code></td>
<td>Slice substrings from each element in array</td>
</tr>
<tr>
<td><code>StringMethods.slice_replace</code></td>
<td>Split each string (a la <code>re.split</code>) in array by given pattern, propagating NA</td>
</tr>
<tr>
<td><code>StringMethods.startswith</code></td>
<td>Return boolean array indicating whether each string starts with passed pattern</td>
</tr>
<tr>
<td><code>StringMethods.strip</code></td>
<td>Strip whitespace (including newlines) from each string in the array</td>
</tr>
<tr>
<td><code>StringMethods.title</code></td>
<td>Convert strings to titlecased version</td>
</tr>
<tr>
<td><code>StringMethods.upper</code></td>
<td>Convert strings in array to uppercase</td>
</tr>
<tr>
<td><code>StringMethods.get_dummies</code></td>
<td>Split each string by sep and return a frame of dummy/indicator variables.</td>
</tr>
</tbody>
</table>

### pandas.core.strings.StringMethods.cat

`StringMethods.cat(others=None, sep=None, na_rep=None)`

Concatenate arrays of strings with given separator

- **Parameters**
  - `arr`: list or array-like
    - `others`: list or array, or list of arrays
    - `sep`: string or None, default None
    - `na_rep`: string or None, default None
    - If None, an NA in any array will propagate

- **Returns**
  - `concat`: array

### pandas.core.strings.StringMethods.center

`StringMethods.center(width)`

“Center” strings, filling left and right side with additional whitespace
Parameters  width : int
Minimum width of resulting string; additional characters will be filled with spaces

Returns  centered : array

pandas.core.strings.StringMethods.contains

StringMethods.**contains**(pat, case=True, flags=0, na=nan, regex=True)
Check whether given pattern is contained in each string in the array

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  Series of boolean values

See Also:

match  analagous, but stricter, relying on re.match instead of re.search

pandas.core.strings.StringMethods.count

StringMethods.**count**(pat, flags=0, **kwargs)
Count occurrences of pattern in each string

Parameters  arr : list or array-like
  pat : string, valid regular expression
  flags : int, default 0 (no flags)
re module flags, e.g. re.IGNORECASE

Returns  counts : arrays

pandas.core.strings.StringMethods.decode

StringMethods.**decode**(encoding, errors='strict')
Decode character string to unicode using indicated encoding

Parameters  encoding : string
errors : string

Returns  decoded : array

28.3. Series  733
**pandas.core.strings.StringMethods.encode**

`StringMethods.encode(encoding, errors='strict')`

Encode character string to some other encoding using indicated encoding.

**Parameters**
- `encoding`: string
- `errors`: string

**Returns**
- `encoded`: array

**pandas.core.strings.StringMethods.endswith**

`StringMethods.endswith(pat, na=nan)`

Return boolean array indicating whether each string ends with passed pattern.

**Parameters**
- `pat`: string
  - Character sequence
- `na`: bool, default NaN

**Returns**
- `endswith`: array (boolean)

**pandas.core.strings.StringMethods.extract**

`StringMethods.extract(pat, flags=0, **kwargs)`

Find groups in each string 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)

**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)')
   letter digit
0     a     1
1     b     2
2    NaN   NaN
```

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.core.strings.StringMethods.findall

StringMethods.findall(pat, flags=0, **kwargs)

Find all occurrences of pattern or regular expression

**Parameters**
- **pat**: string
  - Pattern or regular expression
- **flags**: int, default 0 (no flags)
  - re module flags, e.g. re.IGNORECASE

**Returns**
- **matches**: array

### pandas.core.strings.StringMethods.get

StringMethods.get(i)

Extract element from lists, tuples, or strings in each element in the array

**Parameters**
- **i**: int
  - Integer index (location)

**Returns**
- **items**: array

### pandas.core.strings.StringMethods.join

StringMethods.join(sep)

Join lists contained as elements in array, a la str.join

**Parameters**
- **sep**: string
  - Delimiter

**Returns**
- **joined**: array

### pandas.core.strings.StringMethods.len

StringMethods.len()

Compute length of each string in array.

**Returns**
- **lengths**: array
**pandas.core.strings.StringMethods.lower**

StringMethods.lower()
Convert strings in array to lowercase

Returns lowercase : array

**pandas.core.strings.StringMethods.lstrip**

StringMethods.lstrip(to_strip=None)
Strip whitespace (including newlines) from left side of each string in the array

Parameters to_strip : str or unicode

Returns stripped : array

**pandas.core.strings.StringMethods.match**

StringMethods.match(pat, flags=0, **kwargs)
Deprecated: Find groups in each string 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 of boolean values
if as_indexer=True
Series of tuples
if as_indexer=False, default but deprecated

See Also:
contains analagous, 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.core.strings.StringMethods.pad

StringMethods.pad(width, side='left')
Pad strings with whitespace

Parameters
  arr : list or array-like
  width : int
    Minimum width of resulting string; additional characters will be filled with spaces
  side : {'left', 'right', 'both'}, default 'left'

Returns
  padded : array

pandas.core.strings.StringMethods.repeat

StringMethods.repeat(repeats)
Duplicate each string in the array by indicated number of times

Parameters
  repeats : int or array
    Same value for all (int) or different value per (array)

Returns
  repeated : array

pandas.core.strings.StringMethods.replace

StringMethods.replace(pat, repl, n=-1, case=True, flags=0)
Replace

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 : array

pandas.core.strings.StringMethods.rstrip

StringMethods.rstrip(to_strip=None)
Strip whitespace (including newlines) from right side of each string in the array

Parameters
  to_strip : str or unicode

Returns
  stripped : array
pandas: powerful Python data analysis toolkit, Release 0.13.1

pandas.core.strings.StringMethods.slice

StringMethods.slice (start=None, stop=None, step=1)
Slice substrings from each element in array

Parameters start : int or None
stop : int or None

Returns sliced : array

pandas.core.strings.StringMethods.slice_replace

StringMethods.slice_replace (i=None, j=None)
Slice substrings from each element in array

Parameters start : int or None
stop : int or None

Returns sliced : array

pandas.core.strings.StringMethods.split

StringMethods.split (pat=None, n=-1)
Split each string (a la re.split) in array by given pattern, propagating NA values

Parameters pat : string, default None
String or regular expression to split on. If None, splits on whitespace
n : int, default None (all)

Returns split : array

Notes
Both 0 and -1 will be interpreted as return all splits

pandas.core.strings.StringMethods.startswith

StringMethods.startswith (pat, na=nan)
Return boolean array indicating whether each string starts with passed pattern

Parameters pat : string
Character sequence
na : bool, default NaN

Returns startswith : array (boolean)
**pandas.core.strings.StringMethods.strip**

StringMethodsˌstrip(toˌstrip=None)
Strip whitespace (including newlines) from each string in the array

*Parameters*  
toˌstrip : str or unicode

*Returns* stripped : array

**pandas.core.strings.StringMethods.title**

StringMethodsˌtitle()
Convert strings to titlecased version

*Returns* titled : array

**pandas.core.strings.StringMethods.upper**

StringMethodsˌupper()
Convert strings in array to uppercase

*Returns* uppercase : array

**pandas.core.strings.StringMethods.get_dummies**

StringMethodsˌgetˌdummies(sep='|')
Split each string by sep and return a frame of dummy/indicator variables.

**Examples**

```python
>>> Series(['a|b', 'a', 'a|c']).str.get_dummies()
         a  b  c
0       1  1  0
1       1  0  0
2       1  0  1

>>> pd.Series(['a|b', np.nan, 'a|c']).str.get_dummies()
         a  b  c
0       1  1  0
1       0  0  0
2       1  0  1
```

*See also* pdˌgetˌdummies.

**28.3.14 Plotting**

Seriesˌhist([by, ax, grid, xlabelsize, ...])_  
Draw histogram of the input series using matplotlib

Seriesˌplot(series[, label, kind, ...])_  
Plot the input series with the index on the x-axis using matplotlib
pandas.Series.hist

Series.hist(by=None, ax=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, figsize=None, **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
- **kwds**: keywords
  To be passed to the actual plotting function

**Notes**

See matplotlib documentation online for more on this

pandas.Series.plot

Series.plot(series=None, label=None, kind='line', use_index=True, rot=None, xticks=None, yticks=None, xlim=None, ylim=None, ax=None, style=None, grid=None, legend=False, logx=False, logy=False, secondary_y=False, **kwds)

Plot the input series with the index on the x-axis using matplotlib

**Parameters**

- **label**: label argument to provide to plot
- **kind**: {'line', 'bar', 'barh', 'kde', 'density'}
  - **bar**: vertical bar plot
  - **barh**: horizontal bar plot
  - **kde/density**: Kernel Density Estimation plot
- **use_index**: boolean, default True
  Plot index as axis tick labels
- **rot**: int, default None
  Plot index as axis tick labels
Rotation for tick labels

**xticks** : sequence
  Values to use for the xticks

**yticks** : sequence
  Values to use for the yticks

**xlim** : 2-tuple/list

**ylim** : 2-tuple/list

**ax** : matplotlib axis object
  If not passed, uses gca()

**style** : string, default matplotlib default
  matplotlib line style to use

**grid** : matplotlib grid

**legend:** matplotlib legend

**logx** : boolean, default False
  For line plots, use log scaling on x axis

**logy** : boolean, default False
  For line plots, use log scaling on y axis

**secondary_y** : boolean or sequence of ints, default False
  If True then y-axis will be on the right

**figsize** : a tuple (width, height) in inches

**kwds** : keywords
  Options to pass to matplotlib plotting method

**Notes**

See matplotlib documentation online for more on this subject

### 28.3.15 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.from_csv</td>
<td>Read delimited file into Series</td>
</tr>
<tr>
<td>Series.to_pickle</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>Series.to_csv</td>
<td>Write Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>Series.to_dict</td>
<td>Convert Series to {label -&gt; value} dict</td>
</tr>
<tr>
<td>Series.to_frame</td>
<td>Convert Series to DataFrame</td>
</tr>
<tr>
<td>Series.to_hdf</td>
<td>activate the HDFStore</td>
</tr>
<tr>
<td>Series.to_json</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>Series.to_sparse</td>
<td>Convert Series to SparseSeries</td>
</tr>
<tr>
<td>Series.to_string</td>
<td>Render a string representation of the Series</td>
</tr>
<tr>
<td>Series.to_clipboard</td>
<td>Attempt to write text representation of object to the system clipboard</td>
</tr>
</tbody>
</table>

28.3. Series
**pandas.Series.from_csv**

Class method `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**

Method `Series.to_pickle(path)`

Pickle (serialize) object to input file path

**Parameters**
- **path**: string
  - File path

**pandas.Series.to_csv**

Method `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)`

Write Series to a comma-separated values (csv) file

**Parameters**
- **path**: string file path or file handle / StringIO
  - 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.

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
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.

'rx' 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', 'lzma', 'blosc'}, 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_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_string**

Series.to_string(buf=None, na_rep='NaN', float_format=None, nanRep=None, length=False, dtype=False, name=False)

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

- **length**: boolean, default False
  Add the Series length

- **dtype**: boolean, default False
  Add the Series dtype

- **name**: boolean, default False
  Add the Series name (which may be None)
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

### 28.4 DataFrame

#### 28.4.1 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
  - Dict can contain Series, arrays, constants, or list-like objects
- **index**: Index or array-like
  - Index to use for resulting frame. Will default to np.arange(n) if no indexing information part of input data and no index provided
- **columns**: Index or array-like
  - Column labels to use for resulting frame. Will default to np.arange(n) if no column labels are provided
- **dtype**: dtype, default None
  - Data type to force, 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></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)</td>
</tr>
<tr>
<td>iat</td>
<td></td>
</tr>
<tr>
<td>iloc</td>
<td></td>
</tr>
<tr>
<td>ix</td>
<td></td>
</tr>
<tr>
<td>loc</td>
<td></td>
</tr>
<tr>
<td>ndim</td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td>shape</td>
<td></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
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

pandas.DataFrame.iloc

DataFrame.iloc

pandas.DataFrame.ix

DataFrame.ix

pandas.DataFrame.loc

DataFrame.loc

pandas.DataFrame.ndim

DataFrame.ndim
Number of axes / array dimensions
pandas.DataFrame.shape

DataFrame.shape

pandas.DataFrame.values

DataFrame.values

Numpy representation of NDFrame

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td>add(other[, axis, level, fill_value])</td>
<td>Binary operator add with support to substitute a fill_value for missing data in the input array.</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>append(other[, ignore_index, verify_integrity])</td>
<td>Append columns of other to end of this frame’s columns and index, returning a new object.</td>
</tr>
<tr>
<td>apply(func[, axis, broadcast, raw, reduce, args])</td>
<td>Applies function along input axis of DataFrame.</td>
</tr>
<tr>
<td>applymap(func)</td>
<td>Apply a function to a DataFrame that is intended to operate</td>
</tr>
<tr>
<td>as_blocks([columns])</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has</td>
</tr>
<tr>
<td>as_matrix([columns])</td>
<td>Convert the frame to its Numpy-array matrix representation. Columns</td>
</tr>
<tr>
<td>asfreq(freq[, method, how, normalize])</td>
<td>Convert all TimeSeries inside to specified frequency using DateOffset.</td>
</tr>
<tr>
<td>astype(dtype[, convert_dates, ...])</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>
<tr>
<td>bool()</td>
<td>Return the bool of a single element PandasObject.</td>
</tr>
<tr>
<td>boxplot([column, by, ax, fontsize, rot, grid])</td>
<td>Make a box plot from DataFrame column/columns optionally grouped</td>
</tr>
<tr>
<td>clip([lower, upper, out])</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>clip_lower(threshold)</td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>clip_upper(threshold)</td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td>combine(other, func[, fill_value, overwrite])</td>
<td>Add two DataFrame objects and do not propagate NaN values, so if for a</td>
</tr>
<tr>
<td>combineAdd(other)</td>
<td>Add two DataFrame objects and do not propagate</td>
</tr>
<tr>
<td>combineMult(other)</td>
<td>Multiply two DataFrame objects and do not propagate NaN values, so if if for a</td>
</tr>
<tr>
<td>combine_first(other)</td>
<td>Combine two DataFrame objects and default to non-null values in frame</td>
</tr>
<tr>
<td>compound([axis, skipna, level])</td>
<td>Return the compound percentage of the values for the requested axis</td>
</tr>
<tr>
<td>consolidate([inplace])</td>
<td>Compute NDFrame with “consolidated” internals (data of each dtype)</td>
</tr>
<tr>
<td>convert_objects([convert_dates, ...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>corr([method, min_periods])</td>
<td>Compute pairwise correlation of columns, excluding NA/null values</td>
</tr>
<tr>
<td>corrwith(other[, axis, drop])</td>
<td>Compute pairwise correlation between rows or columns of two DataFrame</td>
</tr>
<tr>
<td>count([axis, level, numeric_only])</td>
<td>Return Series with number of non-NA/null observations over requested</td>
</tr>
<tr>
<td>cov([min_periods])</td>
<td>Compute pairwise covariance of columns, excluding NA/null values</td>
</tr>
<tr>
<td>cummax([axis, dtype, out, skipna])</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>cummin([axis, dtype, out, skipna])</td>
<td>Return cumulative min over requested axis.</td>
</tr>
</tbody>
</table>
Table 28.39 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cumprod(axis, dtype, out, skipna)</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>cumsum(axis, dtype, out, skipna)</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>delevel(*args, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>describe(percentile_width)</td>
<td>Generate various summary statistics of each column, excluding</td>
</tr>
<tr>
<td>diff(periods)</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>div(other[, axis, level, fill_value])</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>divide(other[, axis, level, fill_value])</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>dot(other)</td>
<td>Matrix multiplication with DataFrame or Series objects</td>
</tr>
<tr>
<td>drop(labels[, axis, level, inplace])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>drop_duplicates(cols, take_last, inplace)</td>
<td>Return DataFrame with duplicate rows removed, optionally only</td>
</tr>
<tr>
<td>dropna([how, thresh, subset, inplace])</td>
<td>Return object with labels on given axis omitted where alternately any</td>
</tr>
<tr>
<td>duplicated(cols, take_last)</td>
<td>Return boolean Series denoting duplicate rows, optionally only</td>
</tr>
<tr>
<td>eq(other[, axis, level])</td>
<td>Wraper for flexible comparison methods eq</td>
</tr>
<tr>
<td>eval(expr, **kwargs)</td>
<td>Evaluate an expression in the context of the calling DataFrame</td>
</tr>
<tr>
<td>ffill(axis, inplace, limit, downcast())</td>
<td>Synonym for NDFrame.fillna(method='ffill')</td>
</tr>
<tr>
<td>fillna(value, method, axis, inplace, ...)</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td>filter(items, like, regex, axis)</td>
<td>Restrict the info axis to set of items or wildcard</td>
</tr>
<tr>
<td>first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>first_valid_index()</td>
<td>Return label for first non-NA/null value</td>
</tr>
<tr>
<td>floordiv(other[, axis, level, fill_value])</td>
<td>Binary operator floordiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>from_csv(path[, header, sep, index_col, ...])</td>
<td>Read delimited file into DataFrame</td>
</tr>
<tr>
<td>from_dict(data[, orient, dtype])</td>
<td>Construct DataFrame from dict of array-like or dicts</td>
</tr>
<tr>
<td>from_items(items[, columns, orient])</td>
<td>Convert (key, value) pairs to DataFrame. The keys will be the axis</td>
</tr>
<tr>
<td>from_records(data[, index, exclude, ...])</td>
<td>Convert structured or record ndarray to DataFrame</td>
</tr>
<tr>
<td>get(other[, axis, level])</td>
<td>Wraper for flexible comparison methods ge</td>
</tr>
<tr>
<td>get(key[, default])</td>
<td>Get item from object for given key (DataFrame column, Panel slice,</td>
</tr>
<tr>
<td>get_dtype_counts()</td>
<td>Return the counts of dtypes in this object</td>
</tr>
<tr>
<td>get_ftype_counts()</td>
<td>Return the counts of ftypes in this object</td>
</tr>
<tr>
<td>get_value(index, col)</td>
<td>Quickly retrieve single value at passed column and index</td>
</tr>
<tr>
<td>get_values()</td>
<td>same as values (but handles sparseness conversions)</td>
</tr>
<tr>
<td>groupby(by, axis, level, as_index, sort, ...)</td>
<td>Group series using mapper (dict or key function, apply given function</td>
</tr>
<tr>
<td>gt(other[, axis, level])</td>
<td>Wraper for flexible comparison methods gt</td>
</tr>
<tr>
<td>head(n)</td>
<td>Returns first n rows</td>
</tr>
<tr>
<td>hist(data[, column, by, grid, xlabels, ...])</td>
<td>Draw histogram of the DataFrame’s series using matplotlib / pylab.</td>
</tr>
<tr>
<td>icol(i)</td>
<td></td>
</tr>
<tr>
<td>idxmax(axis, skipna)</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td>idxmin(axis, skipna)</td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td>is_valid(i, j)</td>
<td></td>
</tr>
<tr>
<td>info(VERBOSE, buf, max_cols)</td>
<td>Concise summary of a DataFrame.</td>
</tr>
<tr>
<td>insert(loc, column, value[, allow_duplicates])</td>
<td>Insert column into DataFrame at specified location.</td>
</tr>
<tr>
<td>interpolate(method, axis, limit, inplace, ...))</td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td>irow(i, copy)</td>
<td></td>
</tr>
<tr>
<td>isin(values)</td>
<td>Return boolean DataFrame showing whether each element in the</td>
</tr>
<tr>
<td>isnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>iteritems()</td>
<td>Iterator over (column, series) pairs</td>
</tr>
<tr>
<td>iterkeys(*args, **kwargs)</td>
<td>iiteritems alias used to get around 2to3. Deprecated</td>
</tr>
<tr>
<td>itertuples(index)</td>
<td>Iterate over rows of DataFrame as (index, Series) pairs.</td>
</tr>
<tr>
<td>join(other[, on, how, lsuffix, rsuffix, sort])</td>
<td>Join columns with other DataFrame either on index or on a key.</td>
</tr>
</tbody>
</table>
### Table 28.39 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>keys()</code></td>
<td>Get the ‘info axis’ (see Indexing for more)</td>
</tr>
<tr>
<td><code>kurt([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td><code>kurtosis([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td><code>last(offset)</code></td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td><code>last_valid_index()</code></td>
<td>Return label for last non-NA/null value</td>
</tr>
<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)</code></td>
<td>Returns copy whose values are replaced with nan if the</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>merge(right[, how, on, left_on, right_on, ...])</code></td>
<td>Merge DataFrame objects by performing a database-style join operation by</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>mod(other[, axis, level, fill_value])</code></td>
<td>Binary operator mod with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>mode([axis, numeric_only])</code></td>
<td>Gets the mode of each element along the axis selected.</td>
</tr>
<tr>
<td><code>mul(other[, axis, level, fill_value])</code></td>
<td>Binary operator mul with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>multiply(other[, axis, level, fill_value])</code></td>
<td>Binary operator mul with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>ne(other[, axis, level])</code></td>
<td>Wrapper for flexible comparison methods ne</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are</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>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, rows, cols, ...])</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame. The levels in the</td>
</tr>
<tr>
<td><code>plot([frame, x, y, subplots, sharex, ...])</code></td>
<td>Make line, bar, or scatter plots of DataFrame series with the index on the x-axis</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>Binary operator pow with support to substitute a fill_value for missing data in</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</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>Binary operator radd with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rank([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>Binary operator rtruediv with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>reindex([index, columns])</code></td>
<td>Conform DataFrame to new index with optional filling logic, placing</td>
</tr>
<tr>
<td><code>reindex_axis(labels[, axis, method, level, ...])</code></td>
<td>Conform input object to new index with optional filling logic,</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit])</code></td>
<td>return an object with matching indices 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-ch</td>
</tr>
<tr>
<td><code>reset_index([level, drop, inplace, ...])</code></td>
<td>For DataFrame with multi-level index, return new DataFrame with</td>
</tr>
<tr>
<td><code>rfloordiv(other[, axis, level, fill_value])</code></td>
<td>Binary operator rfloordiv with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rmod(other[, axis, level, fill_value])</code></td>
<td>Binary operator rmod with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rmul(other[, axis, level, fill_value])</code></td>
<td>Binary operator rmul with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rpow(other[, axis, level, fill_value])</code></td>
<td>Binary operator rpow with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rsub(other[, axis, level, fill_value])</code></td>
<td>Binary operator rsub with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td><code>rtruediv(other[, axis, level, fill_value])</code></td>
<td>Binary operator rtruediv with support to substitute a fill_value for missing data in</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.39 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>save(path)</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>select(crit[, axis])</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td>set_index(keys[, drop, append, inplace, ...])</td>
<td>Set the DataFrame index (row labels) using one or more existing</td>
</tr>
<tr>
<td>set_value(index, col, value)</td>
<td>Put single value at passed column and index</td>
</tr>
<tr>
<td>shift([periods, freq, axis])</td>
<td>Shift index by desired number of periods with an optional time freq</td>
</tr>
<tr>
<td>skew([axis, skipna, level, numeric_only])</td>
<td>Return unbiased skew over requested axis</td>
</tr>
<tr>
<td>sort([columns, axis, ascending, inplace])</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>sort_index([axis, by, ascending, inplace, kind])</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>sortlevel([level, axis, ascending, inplace])</td>
<td>Sort multilevel index by chosen axis and primary level.</td>
</tr>
<tr>
<td>squeeze()</td>
<td>squeeze length 1 dimensions</td>
</tr>
<tr>
<td>stack([level, dropna])</td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a</td>
</tr>
<tr>
<td>std([axis, skipna, level, ddof])</td>
<td>Return unbiased standard deviation over requested axis</td>
</tr>
<tr>
<td>sub(other[, axis, level, fill_value])</td>
<td>Binary operator sub with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td>subtract(other[, axis, level, fill_value])</td>
<td>Binary operator sub with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td>sum([axis, skipna, level, numeric_only])</td>
<td>Return the sum of the values for the requested axis</td>
</tr>
<tr>
<td>swapaxes([axis1, axis2[, copy]])</td>
<td>Interchange axes and swap values axes appropriately</td>
</tr>
<tr>
<td>swaplevel(i, j[, axis])</td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
<tr>
<td>tail([n])</td>
<td>Returns last n rows</td>
</tr>
<tr>
<td>take(indices[, axis, convert, is_copy])</td>
<td>Analogous to ndarray.take</td>
</tr>
<tr>
<td>to_clipboard([excel, sep])</td>
<td>Attempt to write text representation of object to the system clipboard</td>
</tr>
<tr>
<td>to_csv(path_or_buf[, sep, na_rep, ...])</td>
<td>Write DataFrame to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>to_dense()</td>
<td>Return dense representation of NDFrame (as opposed to sparse)</td>
</tr>
<tr>
<td>to_dict([outtype])</td>
<td>Convert DataFrame to dictionary.</td>
</tr>
<tr>
<td>to_excel(excel_writer[, sheet_name, na_rep, ...])</td>
<td>Write DataFrame to a excel sheet</td>
</tr>
<tr>
<td>to_gbq(destination_table[, schema, ...])</td>
<td>Write a DataFrame to a Google BigQuery table.</td>
</tr>
<tr>
<td>to_hdf(path_or_buf[, sep, na_rep, ...])</td>
<td>activate the HDFStore</td>
</tr>
<tr>
<td>to_html(buf, columns, col_space, colSpace, ...)</td>
<td>Render a DataFrame as an HTML table.</td>
</tr>
<tr>
<td>to_json(buf, columns, col_space, colSpace, ...)</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>to_latex(buf, columns, col_space, colSpace, ...)</td>
<td>Render a DataFrame to a tabular environment table.</td>
</tr>
<tr>
<td>to_msgpack([path_or_buf])</td>
<td>msgpack (serialize) object to input file path</td>
</tr>
<tr>
<td>to_period(freq, axis, copy)</td>
<td>Convert DataFrame from DatetimeIndex to PeriodIndex with desired</td>
</tr>
<tr>
<td>to_pickle(path)</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>to_records([index, convert_datetime64])</td>
<td>Convert DataFrame to record array. Index will be put in the</td>
</tr>
<tr>
<td>to_sparse([fill_value, kind])</td>
<td>Convert to SparseDataFrame</td>
</tr>
<tr>
<td>to_sql(name, con[, flavor, if_exists])</td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td>to_string(buf, columns, col_space, colSpace, ...)</td>
<td>A class for writing Stata binary dta files from array-like objects</td>
</tr>
<tr>
<td>to_timestamp([freq, how, axis, copy])</td>
<td>Cast to DatetimeIndex of timestamps, at beginning of period</td>
</tr>
<tr>
<td>to_wide(*args, **kwarsgs)</td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td>truediv(other[, axis, level, fill_value])</td>
<td>Binary operator truediv with support to substitute a fill_value for missing data in</td>
</tr>
<tr>
<td>truncate([before, after, axis, copy])</td>
<td>Truncates a sorted DataFrame before and/or after some particular</td>
</tr>
<tr>
<td>tshift([periods, freq, axis])</td>
<td>Shift the time index, using the index’s frequency if available</td>
</tr>
<tr>
<td>tz_convert(tz[, axis, copy])</td>
<td>Convert TimeSeries to target time zone. If it is time zone naive, it</td>
</tr>
<tr>
<td>tz_localize(tz[, axis, copy, infer_dst])</td>
<td>Localize tz-naive TimeSeries to target time zone</td>
</tr>
<tr>
<td>unstack([level])</td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning</td>
</tr>
<tr>
<td>update(other[, join, overwrite, ...])</td>
<td>Modify DataFrame in place using non-NA values from passed</td>
</tr>
<tr>
<td>where(cond[, other, inplace, axis, level, ...])</td>
<td>Return an object of same shape as self and whose corresponding</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 0.13.1

Table 28.39 – continued from previous page

| xs(key[, axis, level, copy, drop_level]) | Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. |

**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)

Binary operator add 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**

Mismatched indices will be unioned together

**pandas.DataFrame.add_prefix**

DataFrame.add_prefix(prefix)

Concatenate prefix string with panel items names.

**Parameters**

prefix : string

**Returns**

with_prefix : type of caller

**pandas.DataFrame.add_suffix**

DataFrame.add_suffix(suffix)

Concatenate suffix string with panel items names.

**Parameters**

suffix : string

**Returns**

with_suffix : type of caller
pandas.DataFrame.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

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 name
   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.all

DataFrame.all (axis=None, bool_only=None, skipna=True, level=None, **kwargs)
Return whether all elements are True over requested axis. %(na_action)s

Parameters axis : {0, 1}
   0 for row-wise, 1 for column-wise
skipna : boolean, default True
   Exclude NA/null values. If an entire row/column is NA, the result will be NA
level : int, default None
   If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame
bool_only : boolean, default None
   Only include boolean data.

Returns any : Series (or DataFrame if level specified)
pandas.DataFrame.any

DataFrame.any (axis=None, bool_only=None, skipna=True, level=None, **kwargs)
Return whether any element is True over requested axis. %(na_action)s

Parameters  
axis : {0, 1}
0 for row-wise, 1 for column-wise
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA
level : int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame
bool_only : boolean, default None
Only include boolean data.

Returns  any : Series (or DataFrame if level specified)

pandas.DataFrame.append

DataFrame.append (other, ignore_index=False, verify_integrity=False)
Append columns of other to end of this frame’s columns and index, returning a new object. Columns not in this frame are added as new columns.

Parameters  other : DataFrame or list of Series/dict-like objects
ignore_index : boolean, default False
If True do not use the index labels. Useful for gluing together record arrays
verify_integrity : boolean, default False
If True, raise ValueError on creating index with duplicates

Returns  appended : DataFrame

Notes
If a list of dict 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

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, 1}

- 0 : apply function to each column
- 1 : 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 func-
tion will receive ndarray objects instead. If you are just applying a NumPy reduc-
tion 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

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(columns=None)
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
are presented in sorted order unless a specific list of columns is provided.
NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters  columns : array-like
Specific column order

Returns  values : a list of Object

pandas.DataFrame.as_matrix

DataFrame.as_matrix(columns=None)
Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless
a specific list of columns is provided.
NOTE: 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, float32 -> float32  float16, float32, float64 -> float64  int32, uint8 -> int32

Returns  values : ndarray
If the caller is heterogeneous and contains booleans or objects, the result will be
of dtype=object

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.astype

DataFrame.astype (dtype, copy=True, raise_on_error=True)
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

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=0, 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, **kwds)
Make a box plot from DataFrame column/columns optionally grouped (stratified) by one or more columns

Parameters
data : DataFrame
column : column names 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 axis object, default None

**fontsize** : int or string

Rotation for ticks

**grid** : boolean, default None (matlab style default)

Axis grid lines

**Returns**

**ax** : matplotlib.axes.AxesSubplot

---

**pandas.DataFrame.clip**

DataFrame.clip(*lower=None, upper=None, out=None*)

Trim values at input threshold(s)

**Parameters**

**lower** : float, default None

**upper** : float, default None

**Returns**

**clipped** : Series

---

**pandas.DataFrame.clip_lower**

DataFrame.clip_lower(*threshold*)

Return copy of the input with values below given value truncated

**Returns**

**clipped** : same type as input

**See Also:**

clip

---

**pandas.DataFrame.clip_upper**

DataFrame.clip_upper(*threshold*)

Return copy of input with values above given value truncated

**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  

dataframe.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  

dataframe.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  

dataframe.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:  

>>> a.combine_first(b)  

dataframe.DataFrame.compound  

Dataframe.compound(axis=None, skipna=None, level=None, **kwargs)  

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, 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 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.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**: if True, attempt to soft convert dates, if ’coerce’, force conversion (and non-convertibles get NaT)

  **convert_numeric**: if True attempt to coerce to numbers (including strings), non-convertibles get NaN

  **convert_timedeltas**: if True, attempt to soft convert timedeltas, if ’coerce’, force conversion (and non-convertibles get NaT)

  **copy**: Boolean, if True, return copy, default is True

  **Returns** **converted**: asm as input object

---

**pandas.DataFrame.copy**

DataFrame."copy"(*deep=True*)

- Make a copy of this object

  **Parameters** **deep**: boolean, 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, 1}
0 to compute column-wise, 1 for row-wise

drop : boolean, default False
Drop missing indices from result, default returns union of all

Returns corrs : 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, 1}
0 for row-wise, 1 for column-wise

level : int, 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**  
*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**  
*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**  
*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**  
*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.delevel

DataFrame.delevel(*args, **kwargs)

pandas.DataFrame.describe

DataFrame.describe(percentile_width=50)

Generate various summary statistics of each column, excluding NaN values. These include: count, mean, std, min, max, and lower%/50%/upper% percentiles

Parameters
percentile_width : float, optional

width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

Returns
DataFrame of summary statistics

pandas.DataFrame.diff

DataFrame.diff(periods=1)

1st discrete difference of object

Parameters
periods : int, default 1

Periods to shift for forming difference

Returns
diffed : DataFrame

pandas.DataFrame.div

DataFrame.div(other, axis='columns', level=None, fill_value=None)

Binary operator truediv 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

Mismatched indices will be unioned together

dataframe.divide

DataFrame.divide(other, axis='columns', level=None, fill_value=None)

Binary operator truediv 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

Mismatched indices will be unioned together

dataframe.dot

DataFrame.dot(other)

Matrix multiplication with DataFrame or Series objects

Parameters other : DataFrame or Series

Returns dot_product : DataFrame or Series

dataframe.drop

DataFrame.drop(labels, axis=0, level=None, inplace=False, **kwargs)

Return new object with labels in requested axis removed

Parameters labels : single label or list-like

axis : int or axis name

level : int or name, default None

For MultiIndex

inplace : bool, default False
If True, do operation inplace and return None.

**Returns**

- dropped : type of caller

### pandas.DataFrame.drop_duplicates

DataFrame.drop_duplicates(cols=None, take_last=False, inplace=False)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

**Parameters**

- cols : 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, 1}, 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(cols=None, take_last=False)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

**Parameters**

- cols : 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` : 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')
```

28.4. DataFrame
pandas.DataFrame.ffill

DataFrame.ffill(\texttt{axis}=0, \texttt{inplace}=\texttt{False}, \texttt{limit}=\texttt{None}, \texttt{downcast}=\texttt{None})

Synonym for NDFrame.fillna(method='ffill')

pandas.DataFrame.fillna

DataFrame.fillna(\texttt{value}=\texttt{None}, \texttt{method}=\texttt{None}, \texttt{axis}=0, \texttt{inplace}=\texttt{False}, \texttt{limit}=\texttt{None}, \texttt{downcast}=\texttt{None})

Fill NA/NaN values using the specified method

\textbf{Parameters} \texttt{method}: \{'backfill', \textquote{bfill'}, \textquote{pad'}, \textquote{ffill'}, \texttt{None}\}, default \texttt{None}

Method to use for filling holes in reindexed Series \texttt{pad} / \texttt{ffill}: propagate last valid observation forward to next valid \texttt{backfill} / \texttt{bfill}: use NEXT valid observation to fill gap

\texttt{value}: scalar, dict, or Series

Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series will not be filled). This value cannot be a list.

\texttt{axis}: \{0, 1\}, default 0

• 0: fill column-by-column
• 1: fill row-by-row

\texttt{inplace}: boolean, default \texttt{False}

If \texttt{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).

\texttt{limit}: int, default \texttt{None}

Maximum size gap to forward or backward fill

\texttt{downcast}: dict, default is \texttt{None}

a dict of item->dtype of what to downcast if possible, or the string \textquote{infer} which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

\textbf{Returns} \texttt{filled}: same type as caller

\textbf{See Also}:

reindex, asfreq

pandas.DataFrame.filter

DataFrame.filter(\texttt{items}=\texttt{None}, \texttt{like}=\texttt{None}, \texttt{regex}=\texttt{None}, \texttt{axis}=\texttt{None})

Restrict the info axis to set of items or wildcard

\textbf{Parameters} \texttt{items}: list-like

List of info axis to restrict to (must not all be present)

\texttt{like}: string

Keep info axis where \textquote{arg in col == True}’

\texttt{regex}: string (regular expression)
Keep info axis with re.search(regex, col) == True

**Notes**

Arguments are mutually exclusive, but this is not checked for

```python
def pandas.DataFrame.first
```

DataFrame.

**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

```python
def pandas.DataFrame.first_valid_index
```

DataFrame.

**DataFrame.first_valid_index**()

Return label for first non-NA/null value

```python
def pandas.DataFrame.floordiv
```

DataFrame.

**DataFrame.floordiv**(other, axis=’columns’, level=None, fill_value=None)

Binary operator floordiv 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**

Mismatched indices will be unioned together
pandas.DataFrame.from_csv

classmethod DataFrame.from_csv(path, header=0, sep=';', index_col=0, parse_dates=True, encoding=None, tupleize_cols=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’.

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

dataFrame.ge

DataFrame.ge(other, axis='columns', level=None)

Wrapper for flexible comparison methods ge
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.DataFrame.get**

```python
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**

```python
DataFrame.get_dtype_counts()
```
Return the counts of dtypes in this object

**pandas.DataFrame.get_ftype_counts**

```python
DataFrame.get_ftype_counts()
```
Return the counts of ftypes in this object

**pandas.DataFrame.get_value**

```python
DataFrame.get_value(index, col)
```
Quickly retrieve single value at passed column and index

**Parameters**
- `index` : row label
- `col` : column label

**Returns**
- `value` : scalar value

**pandas.DataFrame.get_values**

```python
DataFrame.get_values()
```
same as values (but handles sparseness conversions)

**pandas.DataFrame.groupby**

```python
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 result >>> data.groupby(func, axis=0).mean()
# DataFrame result >>> data.groupby(['col1', 'col2'])[['col3']].mean()
# DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()

pandas.DataFrame.gt

Dataframe.gt (other, axis='columns', level=None)

Wrapper for flexible comparison methods gt

pandas.DataFrame.head

Dataframe.head(n=5)

Returns first n rows

pandas.DataFrame.hist

Dataframe.hist (data, column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None, layout=None, **kwds)

Draw histogram of the DataFrame’s series using matplotlib / pylab.

Parameters data : DataFrame

  column : string or sequence

  If passed, will be used to limit data to a subset of columns

by : object, optional

  If passed, then used to form histograms for separate groups

grid : boolean, default True

  Whether to show axis grid lines

xlabelsize : int, default None
If specified changes the x-axis label size

**xrot** : float, default None
rotation of x axis labels

**ylabelsize** : int, default None
If specified changes the y-axis label size

**yrot** : float, default None
rotation of y axis labels

**ax** : matplotlib axes object, default None

**sharex** : bool, if True, the X axis will be shared amongst all subplots.

**sharey** : bool, if True, the Y axis will be shared amongst all subplots.

**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

**kwds** : other plotting keyword arguments
To be passed to hist function

### pandas.DataFrame.iloc

Dataframe.\texttt{iloc}(i)

### pandas.DataFrame.idxmax

Dataframe.\texttt{idxmax}(axis=0, skipna=True)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

**Parameters**

- **axis** : \{0, 1\}
  
  0 for row-wise, 1 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.
DataFrame.\texttt{idxmin} \((axis=0, \text{skipna}=True)\)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

\textbf{Parameters}  
axis : \{0, 1\}
0 for row-wise, 1 for column-wise

\textbf{skipna} : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

\textbf{Returns}  
\texttt{idxmin} : Series

\textbf{See Also:}
\texttt{Series.idxmin}

\textbf{Notes}
This method is the DataFrame version of \texttt{ndarray.argmin}.

DataFrame.\texttt{iget_value} \((i,j)\)

DataFrame.\texttt{info} \((\text{verbose}=\text{True}, \text{buf}=None, \text{max_cols}=None)\)
Concise summary of a DataFrame.

\textbf{Parameters}  
\texttt{verbose} : boolean, default True
If False, don’t print column count summary

\texttt{buf} : writable buffer, defaults to \texttt{sys.stdout}

\texttt{max_cols} : int, default None
Determines whether full summary or short summary is printed

DataFrame.\texttt{insert} \((\text{loc}, \text{column}, \text{value}, \text{allow_duplicates}=\text{False})\)
Insert column into DataFrame at specified location.

If \texttt{allow_duplicates} is False, raises Exception if column is already contained in the DataFrame.

\textbf{Parameters}  
\texttt{loc} : int
Must have \(0 \leq \text{loc} \leq \text{len(columns)}\)

\texttt{column} : object

\texttt{value} : int, Series, or array-like
pandas.DataFrame.interpolate

DataFrame.interpolate(method='linear', axis=0, limit=None, inplace=False, downcast='infer', **kwargs)

Interpolate values according to different methods.

Parameters:

- **method**: {'linear', 'time', 'values', 'index', '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': 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 'infer'
  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 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
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

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
>>> 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.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

\textbf{lsuffix} : string
Suffix to use from left frame’s overlapping columns

\textbf{rsuffix} : string
Suffix to use from right frame’s overlapping columns

\textbf{sort} : boolean, default False
Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

\textbf{Returns} \hspace{1em} \textbf{joined} : DataFrame

\textbf{Notes}

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

\textbf{pandas.DataFrame.keys}

\textbf{DataFrame.keys ()}
Get the ‘info axis’ (see Indexing for more)
This is index for Series, columns for DataFrame and major_axis for Panel.

\textbf{pandas.DataFrame.kurt}

\textbf{DataFrame.kurt (axis=None, skipna=None, level=None, numeric_only=None, \texttt{**kwargs})}
Return unbiased kurtosis over requested axis Normalized by N-1

\textbf{Parameters} \hspace{1em} \textbf{axis} : \{index (0), columns (1)\}

\textbf{skipna} : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

\textbf{level} : int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

\textbf{numeric_only} : boolean, default None
Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

\textbf{Returns} \hspace{1em} \textbf{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 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, 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, **kwargs)
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, 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)

Returns copy whose values are replaced with nan if the inverted condition is True

- **Parameters**
  - cond: boolean NDFrame or array

- **Returns**
  - wh: same as input

**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, 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, 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, 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.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

**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  bar  2       bar  6
 1  bar  2       bar  8
 2  baz  3      NaN   NaN
 3  foo  1       foo  5
 4  foo  4       foo  5
 5  NaN  NaN      qux  7
```

**pandas.DataFrame.min**

DataFrame.min(*axis=None, skipna=None, level=None, numeric_only=None, **kwargs*)

This method returns the minimum of the values in the object. If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

**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, 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)

Binary operator mod 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

Mismatched indices will be unioned together

**pandas.DataFrame.mode**

DataFrame.mode (axis=0, numeric_only=False)

Gets the mode 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.

Parameters  

axis : {0, 1, ‘index’, ‘columns’} (default 0)

- 0/’index’ : get mode of each column
- 1/’columns’ : get mode of each row

numeric_only : boolean, default False

if True, only apply to numeric columns

Returns  

modes : DataFrame (sorted)

**pandas.DataFrame.mul**

DataFrame.mul (other, axis='columns', level=None, fill_value=None)

Binary operator mul 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**

Mismatched indices will be unioned together

---

**pandas.DataFrame.multiply**

DataFrame.multiply(*other*, **axis**='columns', **level**=None, **fill_value**=None)  
Binary operator mul 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**

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
pandas.DataFrame.pct_change

**DataFrame.pct_change** *(periods=1, fill_method=’pad’, limit=None, freq=None, **kwds)*

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: same type as caller

pandas.DataFrame.pivot

**DataFrame.pivot** *(index=None, columns=None, values=None)*

Reshape data (produce a “pivot” table) based on column values. Uses unique values from index / columns to form axes 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
code
>>> df
  foo  bar  baz
0  one  A   1.
1  one  B   2.
2  one  C   3.
3  two  A   4.
```
pandas: powerful Python data analysis toolkit, Release 0.13.1

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

>>> 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, rows=None, cols=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
- **rows**: list of column names or arrays to group on
  Keys to group on the x-axis of the pivot table
- **cols**: list of column names or arrays to group on
  Keys to group on the y-axis of the pivot table
- **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
A 0  foo one small  1
   1  foo one large  2
   2  foo one large  2
   3  foo two small  3
```
4  foo two small  3
5  bar one large  4
6  bar one small  5
7  bar two small  6
8  bar two large  7

```python
>>> table = pivot_table(df, values='D', rows=['A', 'B'],
...                      cols=['C'], aggfunc=np.sum)
```

```python
>>> table
       small  large
foo one   1   4
   two   6   NaN
bar one   5   4
   two   6   7
```

**pandas.DataFrame.plot**

`DataFrame.plot` makes line, bar, or scatter plots of DataFrame series with the index on the x-axis using matplotlib / pylab.

**Parameters**

- `frame`: DataFrame
- `x`: label or position, default None
- `y`: label or position, default None
  - Allows plotting of one column versus another
- `subplots`: boolean, default False
  - Make separate subplots for each time series
- `sharex`: boolean, default True
  - In case subplots=True, share x axis
- `sharey`: boolean, default False
  - In case subplots=True, share y axis
- `use_index`: boolean, default True
  - Use index as ticks for x axis
- `stacked`: boolean, default False
  - If True, create stacked bar plot. Only valid for DataFrame input
- `sort_columns`: boolean, default False
  - Sort column names to determine plot ordering
- `title`: string
  - Title to use for the plot
- `grid`: boolean, default None (matlab style default)
  - Axis grid lines
legend : boolean, default True
  Place legend on axis subplots
ax : matplotlib axis object, default None
style : list or dict
  matplotlib line style per column
  bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density
  Estimation plot scatter: scatter plot
logx : boolean, default False
  For line plots, use log scaling on x axis
logy : boolean, default False
  For line plots, use log scaling on y axis
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
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, should the legend label the axis of the various
  columns automatically
colormap : str or matplotlib colormap object, default None
  Colormap to select colors from. If string, load colormap with that name from
  matplotlib.
kwds : keywords
  Options to pass to matplotlib plotting method

Returns  ax_or_axes : matplotlib.AxesSubplot or list of them

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)
Binary operator pow 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

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, 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, 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 scoreatpercentile in scipy.stats

**Parameters**  **q** : quantile, default 0.5 (50% quantile)

0 <= q <= 1

**axis** : {0, 1}

0 for row-wise, 1 for column-wise

**Returns**  **quantiles** : Series

**pandas.DataFrame.query**

```
DataFrame.query(expr, **kwargs)
```

Query the columns of a frame with a boolean expression.

**Parameters**  **expr** : string

The query string to evaluate. The result of the evaluation of this expression is first passed to `loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `__getitem__()`.

**kwargs** : dict

See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

**Returns**  **q** : DataFrame or Series

See Also:

- `pandas.eval`, `DataFrame.eval`

**Notes**

This method uses the top-level `eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the & and | (bitwise) operators have the precedence of their boolean cousins, and and or. This is syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.
The `index` and `columns` attributes of the `DataFrame` instance is placed in the 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 this variable, and you can also use the name of the index to identify it in a query.

For further details and examples see the `query` documentation in `indexing`.

**Examples**

```python
>>> from numpy.random import randn
>>> from pandas import DataFrame

>>> df = DataFrame(randn(10, 2), columns=list('ab'))

>>> df.query('a > b')

>>> df[df.a > df.b]  # same result as the previous expression
```

### pandas.DataFrame.radd

`DataFrame.radd(other, axis='columns', level=None, fill_value=None)`

Binary operator `radd` 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**

Mismatched indices will be unioned together

### pandas.DataFrame.rank

`DataFrame.rank(axis=0, numeric_only=None, method='average', na_option='keep', ascending=True)`

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, 1}, 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’}
  - 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

**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)

**Returns**  **ranks** : DataFrame

**pandas.DataFrame.rdiv**

DataFrame. rdiv (**other**, **axis**='columns', **level**=None, **fill_value**=None)
Binary operator rtruediv 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**
Mismatched indices will be unioned together

**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** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None
Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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

**takeable** : boolean, default False

treat the passed as positional values

**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** **index** : array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : {0,1,’index’,’columns’}

**method** : {'backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, default None

Method to use for filling holes in reindexed object. pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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 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.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.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

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 inter-pretated 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.\texttt{rfloordiv}(other, axis=’columns’, level=None, fill_value=None)
Binary operator rfloordiv 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 miss-
ing, 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**
Mismatched indices will be unioned together

**pandas.DataFrame.rmod**

DataFrame.\texttt{rmod}(other, axis=’columns’, level=None, fill_value=None)
Binary operator rmod 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 miss-
ing, 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**

Mismatched indices will be unioned together

**pandas.DataFrame.rmul**

DataFrame.*rmul* (*other*, *axis*='*columns*', *level*=None, *fill_value*=None)  
Binary operator rmul 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**

Mismatched indices will be unioned together

**pandas.DataFrame.rpow**

DataFrame.*rpow* (*other*, *axis*='*columns*', *level*=None, *fill_value*=None)  
Binary operator rpow 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

Mismatched indices will be unioned together

**pandas.DataFrame.rsub**

Dataframe\(_{\text{rsub}}\) \((other, axis=’columns’, level=None, fill_value=None)\)

Binary operator rsub 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

Mismatched indices will be unioned together

**pandas.DataFrame.rtruediv**

Dataframe\(_{\text{rtruediv}}\) \((other, axis=’columns’, level=None, fill_value=None)\)

Binary operator rtruediv 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

Mismatched indices will be unioned together
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.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.set_value

DataFrame.set_value(index, col, value)
Put single value at passed column and index
Parameters  \textbf{index} : row label
    \textbf{col} : column label
    \textbf{value} : scalar value

Returns  \textbf{frame} : DataFrame
    If label pair is contained, will be reference to calling DataFrame, otherwise a new object

\texttt{pandas.DataFrame.shift}

\texttt{DataFrame.shift (periods=1, freq=None, axis=0, **kwds)}
    Shift index by desired number of periods with an optional time freq

Parameters  \textbf{periods} : int
    Number of periods to move, can be positive or negative

\textbf{freq} : DateOffset, timedelta, or time rule string, optional
    Increment to use from datetools module or time rule (e.g. ‘EOM’)

Returns  \textbf{shifted} : same type as caller

Notes
    If freq is specified then the index values are shifted but the data if not realigned

\texttt{pandas.DataFrame.skew}

\texttt{DataFrame.skew (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)}
    Return unbiased skew over requested axis Normalized by N-1

Parameters  \textbf{axis} : \{index (0), columns (1)}

\textbf{skipna} : boolean, default True
    Exclude NA/null values. If an entire row/column is NA, the result will be NA

\textbf{level} : int, default None
    If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

\textbf{numeric_only} : boolean, default None
    Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns  \textbf{skew} : Series or DataFrame (if level specified)

\texttt{pandas.DataFrame.sort}

\texttt{DataFrame.sort (columns=None, column=None, axis=0, ascending=True, inplace=False)}
    Sort DataFrame either by labels (along either axis) or by the values in column(s)

Parameters  \textbf{columns} : object
Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

**ascending** : boolean or list, default True
Sort ascending vs. descending. Specify list for multiple sort orders

**axis** : {0, 1}
Sort index/rows versus columns

**inplace** : boolean, default False
Sort the DataFrame without creating a new instance

**Returns**  
**sorted** : DataFrame

**Examples**

```python
>>> 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')
Sort DataFrame either by labels (along either axis) or by the values in a column

**Parameters**  
**axis** : {0, 1}
Sort index/rows versus columns

**by** : object
Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

**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

**Returns**  
**sorted** : DataFrame

**Examples**

```python
>>> result = df.sort_index(by=['A', 'B'], ascending=[True, False])
```

**pandas.DataFrame.sortlevel**

DataFrame.sortlevel(  
  level=0, axis=0, ascending=True, inplace=False)
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, 1}
- `ascending` : boolean, default True
- `inplace` : boolean, default False

Sort the DataFrame without creating a new instance

**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.

**Parameters**
- `level` : int, string, or list of these, default last level
  - Level(s) to stack, can pass level name
- `dropna` : boolean, default True
  - Whether to drop rows in the resulting Frame/Series with no valid values

**Returns**
- `stacked` : DataFrame or Series

**Examples**

```python
>>> s
   a  b
one 1. 2.
two 3. 4.

>>> s.stack()
   a   b
one 1  2
   b  3  4
```

**pandas.DataFrame.std**

DataFrame`.std`(axis=None, skipna=None, level=None, ddof=1, **kwargs)

Return unbiased standard deviation 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, 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 stdev : Series or DataFrame (if level specified)

pandas.DataFrame.sub

DataFrame.sub (other, axis='columns', level=None, fill_value=None)

Binary operator sub 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

Mismatched indices will be unioned together

pandas.DataFrame.subtract

DataFrame.subtract (other, axis='columns', level=None, fill_value=None)

Binary operator sub 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**

Mismatched indices will be unioned together

```python
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, 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)

```python
pandas.DataFrame.swapaxes
```

DataFrame.swapaxes (axis1, axis2, copy=True)
Interchange axes and swap values axes appropriately

**Returns**

y : same as input

```python
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)

```python
pandas.DataFrame.tail
```

DataFrame.tail (n=5)
Returns last n rows

```python
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, sep=',', na_rep='', float_format=None, cols=None, header=True, index=True, index_label=None, mode='w', encoding=None, quoting=None, line_terminator='
', chunksize=None, tupleize_cols=False, date_format=None, **kwds)  
Write DataFrame to a comma-separated values (csv) file.

**Parameters**  
- **path_or_buf**: string or file handle / StringIO  
  - File path  
- **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  
- **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, 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 if the contents are non-ascii, for python
    versions prior to 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
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.

**pandas.DataFrame.to_dense**

DataFrame.to_dense()
    Return dense representation of NDFrame (as opposed to sparse)

**pandas.DataFrame.to_dict**

DataFrame.to_dict(outtype='dict')
    Convert DataFrame to dictionary.

    Parameters outtype : str {'dict', 'list', 'series', 'records'}
Determines the type of the values of the dictionary. The default \texttt{dict} is a nested
dictionary \{column \rightarrow \{index \rightarrow value\}\}. \texttt{list} returns \{column \rightarrow list(values)\}.
\texttt{series} returns \{column \rightarrow Series(values)\}. \texttt{records} returns \{\{columns \rightarrow value\}\}.
Abbreviations are allowed.

\textbf{Returns} \texttt{result} : dict like \{column \rightarrow \{index \rightarrow value\}\}

\texttt{pandas.DataFrame.to_excel}

\texttt{DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, cols=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True)}

Write DataFrame to an excel sheet

\textbf{Parameters} \texttt{excel_writer} : string or ExcelWriter object

File path or existing ExcelWriter

\texttt{sheet_name} : string, default ‘Sheet1’

Name of sheet which will contain DataFrame

\texttt{na_rep} : string, default ‘’

Missing data representation

\texttt{float_format} : string, default None

Format string for floating point numbers

\texttt{cols} : sequence, optional

Columns to write

\texttt{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

\texttt{index} : boolean, default True

Write row names (index)

\texttt{index_label} : string or sequence, default None

Column label for index column(s) if desired. If None is given, and \texttt{header} and
\texttt{index} are True, then the index names are used. A sequence should be given if the
DataFrame uses MultiIndex.

\texttt{startrow} :

upper left cell row to dump data frame

\texttt{startcol} :

upper left cell column to dump data frame

\texttt{engine} : string, default None

write engine to use - you can also set this via the options
\texttt{io.excel.xlsx.writer}, \texttt{io.excel.xls.writer}, and
\texttt{io.excel.xlsm.writer}.

\texttt{merge_cells} : boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.
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, schema=None, col_order=None, if_exists='fail', **kwargs)`

Write a DataFrame to a Google BigQuery table.

If the table exists, the DataFrame will be appended. If not, a new table will be created, in which case the schema will have to be specified. By default, rows will be written in the order they appear in the DataFrame, though the user may specify an alternative order.

**Parameters**

- `destination_table` : string
  name of table to be written, in the form 'dataset.tablename'

- `schema` : sequence (optional)
  list of column types in order for data to be inserted, e.g. ['INTEGER', 'TIMESTAMP', 'BOOLEAN']

- `col_order` : sequence (optional)
  order which columns are to be inserted, e.g. ['primary_key', 'birthday', 'username']

- `if_exists` : {'fail', 'replace', 'append'} (optional)
  - 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.

**kwargs** are passed to the Client constructor

**Raises**

- `SchemaMissing` : Raised if the ‘if_exists’ parameter is set to ‘replace’, but no schema is specified
- `TableExists` : Raised if the specified ‘destination_table’ exists but the ‘if_exists’ parameter is set to ‘fail’ (the default)
- `InvalidSchema` : Raised if the ‘schema’ parameter does not match the provided DataFrame

`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
    indentifer 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', 'Izo', '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_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, sparse=False, index_names=True, justify=None, force_unicode=None, bold_rows=True, classes=None, escape=True, max_rows=None, max_cols=None, show_dimensions=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, if the result is a string , it 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

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

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, force_unicode=None)
```
Render a DataFrame to a tabular environment table. You can splice this into a LaTeX document.

to_latex-specific options:

bold_rows [boolean, default True] Make the row labels bold in the 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, if the result is a string , it 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  
- `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, 1}, 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', if_exists='fail', **kwargs)
Write records stored in a DataFrame to a SQL database.

Parameters
- name : str
  Name of SQL table
- con : an open SQL database connection object
- flavor: {'sqlite', 'mysql', 'oracle'}, default 'sqlite'
- 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.

pandas.DataFrame.to_stata

DataFrame.to_stata (fname, convert_dates=None, write_index=True, encoding='latin-1', byteorder=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()
```
pandas: powerful Python data analysis toolkit, Release 0.13.1

Or with dates

```python
>>> 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, colSpace=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, nanRep=None, index_names=True, justify=None, force_unicode=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, if the result is a string, it 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

- **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 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, 1} 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)

Binary operator truediv 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

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, **kwds)
Shift the time index, using the index’s frequency if available

Parameters periods : int
Number of periods to move, can be positive or negative
copy : boolean, default is True,
return a copy of the truncated section

Returns shifted : NDFrame

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

pandas.DataFrame.tz_convert

DataFrame.tz_convert(tz, axis=0, copy=True)
Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

Parameters tz : string or pytz.timezone object
copy : boolean, default True
Also make a copy of the underlying data

**pandas.DataFrame.tz_localize**

`DataFrame.tz_localize(tz, axis=0, copy=True, infer_dst=False)`  
Localize tz-naive TimeSeries to target time zone

**Parameters**  
- `tz`: string or pytz.timezone object  
- `copy`: boolean, default True  
- `infer_dst`: boolean, default False  

Also make a copy of the underlying data

**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)

**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
code_snippet
>>> 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
```

```python
code_snippet
>>> s.unstack(level=-1)
a   b
one 1 2
two 3 4
```

```python
code_snippet
>>> 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.

**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', 'right', 'outer', 'inner'}, 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, **kwargs)
Return unbiased variance 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, 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**
- **variance** : 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
- **axis**: alignment axis if needed, default None
- **level**: alignment level if needed, default None
- **try_cast**: boolean, default False
- **try_cast**
  - try to cast the result back to the input type (if possible),
- **raise_on_error**: boolean, default True
- **raise_on_error**: 
  - 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=True, 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, default True
  - 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

#### Examples

```python
>>> df
    A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
    A  B  C
  a  4  5  2
Name: a
```
>>> df.xs('C', axis=1)
  a  2
  b  9
  c  3
Name: C

>>> s = df.xs('a', copy=False)

>>> s['A'] = 100

>>> df
   A  B  C
a 100 5  2
b  4  0  9
c  9  7  3

>>> 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   one 4  1  8  9
baz   one 1  6  6  8  0

>>> df.xs(('baz', 2), level=[0, 'third'])
   A  B  C  D
second
three 5  3  5  3

28.4.2 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 matrix representation. Columns</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)</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.values</td>
<td>Numpy representation of NDFrame</td>
</tr>
<tr>
<td>DataFrame.axes</td>
<td></td>
</tr>
<tr>
<td>DataFrame.ndim</td>
<td>Number of axes / array dimensions</td>
</tr>
<tr>
<td>DataFrame.shape</td>
<td></td>
</tr>
</tbody>
</table>
pandas.DataFrame.as_matrix

DataFrame.as_matrix(columns=None)
Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

**NOTE:** 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, float32 -> float32 float16, float32, float64 -> float64 int32, uint8 -> int32**

**Returns** values : ndarray
If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object

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.values

DataFrame.values
Numpy representation of NDFrame

pandas.DataFrame.axes

DataFrame.axes

pandas.DataFrame.ndim

DataFrame.ndim
Number of axes / array dimensions
pandas.DataFrame.shape

DataFrame.shape

28.4.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.astype(dtype[, copy, raise_on_error])</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>DataFrame.convert_objects([convert_dates,...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>DataFrame.copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>DataFrame.isnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>DataFrame.notnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
</tbody>
</table>

pandas.DataFrame.astype

DataFrame.astype(dtype, copy=True, raise_on_error=True)

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

**Returns**

casted : 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 : if True, attempt to soft convert dates, if ‘coerce’, force conversion (and non-convertibles get NaT)
convert_numeric : if True attempt to coerce to numbers (including strings), non-convertibles get NaN
convert_timedeltas : if True, attempt to soft convert timedeltas, if ‘coerce’, force conversion (and non-convertibles get NaT)
copy : Boolean, if True, return copy, default is True

**Returns**

cast : asm as input object

pandas.DataFrame.copy

DataFrame.copy(deep=True)

Make a copy of this object

**Parameters**

depth : boolean, 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

pandas.DataFrame.notnull

DataFrame.notnull()
Return a boolean same-sized object indicating if the values are not null

28.4.4 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></td>
</tr>
<tr>
<td>DataFrame.iat</td>
<td></td>
</tr>
<tr>
<td>DataFrame.ix</td>
<td></td>
</tr>
<tr>
<td>DataFrame.loc</td>
<td></td>
</tr>
<tr>
<td>DataFrame.iloc</td>
<td></td>
</tr>
<tr>
<td>DataFrame.insert(loc, column, value[,...])</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>Iterator 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.itertuples([index])</td>
<td>Iterate over rows of DataFrame as tuples, with index value</td>
</tr>
<tr>
<td>DataFrame.lookup(row_labels, col_labels)</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</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

pandas.DataFrame.iat

DataFrame.iat

pandas.DataFrame.ix

DataFrame.ix
pandas.DataFrame.loc

DataFrame.loc

pandas.DataFrame.iloc

DataFrame.iloc

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 info in order 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
  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
>>> 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.

**pandas.DataFrame.lookup**

Dataframe.

**pandas.DataFrame.pop**

Dataframe.

**pandas.DataFrame.tail**

Dataframe.

**pandas.DataFrame.xs**

Dataframe.
**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, default True

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

---

**Examples**

```python
def
A B C
da 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
>>> s = df.xs('a', copy=False)
>>> s['A'] = 100

A B C
a 100 5 2
b 4 0 9
c 9 7 3
```
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, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
     A  B
0  True  False # Column A in 'other' has a 3, but not at index 1.
1  False  False
2  True  True
```

pandas.DataFrame.query

DataFrame.query(expr, **kwargs)

Query the columns of a frame with a boolean expression.

Parameters expr : string

The query string to evaluate. The result of the evaluation of this expression is first passed to loc and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to __getitem__().
kwarg : dict
See the documentation for eval() for complete details on the keyword arguments accepted by query().

Returns q : DataFrame or Series
See Also:
pandas.eval, DataFrame.eval

Notes
This method uses the top-level eval() function to evaluate the passed query.
The query() method uses a slightly modified Python syntax by default. For example, the & and | (bitwise) operators have the precedence of their boolean cousins, and 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 index and columns attributes of the DataFrame instance is placed in the 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 this variable, and 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
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')

# same result as the previous expression

For more information on .at, .iat, .ix, .loc, and .iloc, see the indexing documentation.

28.4.5 Binary operator functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.add</td>
<td>Binary operator add with support to substitute a fill_value for missing data in add</td>
</tr>
<tr>
<td>DataFrame.sub</td>
<td>Binary operator sub with support to substitute a fill_value for missing data in sub</td>
</tr>
<tr>
<td>DataFrame.mul</td>
<td>Binary operator mul with support to substitute a fill_value for missing data in mul</td>
</tr>
<tr>
<td>DataFrame.div</td>
<td>Binary operator true div with support to substitute a fill_value for missing data in div</td>
</tr>
<tr>
<td>DataFrame.truediv</td>
<td>Binary operator true div with support to substitute a fill_value for missing data in truediv</td>
</tr>
<tr>
<td>DataFrame.floordiv</td>
<td>Binary operator floordiv with support to substitute a fill_value for missing data in floordiv</td>
</tr>
<tr>
<td>DataFrame.mod</td>
<td>Binary operator mod with support to substitute a fill_value for missing data in mod</td>
</tr>
<tr>
<td>DataFrame.pow</td>
<td>Binary operator pow with support to substitute a fill_value for missing data in pow</td>
</tr>
<tr>
<td>DataFrame.radd</td>
<td>Binary operator radd with support to substitute a fill_value for missing data in radd</td>
</tr>
<tr>
<td>DataFrame.rsub</td>
<td>Binary operator rsub with support to substitute a fill_value for missing data in rsub</td>
</tr>
<tr>
<td>DataFrame.rmul</td>
<td>Binary operator rmul with support to substitute a fill_value for missing data in rmul</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.43 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.rdiv</td>
<td>Binary operator rdiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>DataFrame.rtruediv</td>
<td>Binary operator rtruediv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>DataFrame.rfloordiv</td>
<td>Binary operator rfloordiv with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>DataFrame.rmod</td>
<td>Binary operator rmod with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>DataFrame.rpow</td>
<td>Binary operator rpow with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>DataFrame.lt</td>
<td>Wrapper for flexible comparison methods lt</td>
</tr>
<tr>
<td>DataFrame.gt</td>
<td>Wrapper for flexible comparison methods gt</td>
</tr>
<tr>
<td>DataFrame.le</td>
<td>Wrapper for flexible comparison methods le</td>
</tr>
<tr>
<td>DataFrame.ne</td>
<td>Wrapper for flexible comparison methods ne</td>
</tr>
<tr>
<td>DataFrame.eq</td>
<td>Wrapper for flexible comparison methods eq</td>
</tr>
<tr>
<td>DataFrame.combine</td>
<td>Add two DataFrame objects and do not propagate NaN values, so if for a</td>
</tr>
<tr>
<td>DataFrame.combineAdd</td>
<td>Add two DataFrame objects and do not propagate</td>
</tr>
<tr>
<td>DataFrame.combine_first</td>
<td>Combine two DataFrame objects and default to non-null values in frame</td>
</tr>
<tr>
<td>DataFrame.combineMult</td>
<td>Multiply two DataFrame objects and do not propagate NaN values, so if</td>
</tr>
</tbody>
</table>

pandas.DataFrame.add

DataFrame.add (other, axis='columns', level=None, fill_value=None)

    Binary operator add 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

    Mismatched indices will be unioned together

pandas.DataFrame.sub

DataFrame.sub (other, axis='columns', level=None, fill_value=None)

    Binary operator sub 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

Mismatched indices will be unioned together

pandas.DataFrame.mul

DataFrame.mul(other, axis='columns', level=None, fill_value=None)

Binary operator mul 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

Mismatched indices will be unioned together

pandas.DataFrame.div

DataFrame.div(other, axis='columns', level=None, fill_value=None)

Binary operator truediv 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

Mismatched indices will be unioned together

**pandas.DataFrame.truediv**

`DataFrame.truediv(other, axis='columns', level=None, fill_value=None)`

Binary operator truediv 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

Mismatched indices will be unioned together

**pandas.DataFrame.floordiv**

`DataFrame.floordiv(other, axis='columns', level=None, fill_value=None)`

Binary operator floordiv 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

Mismatched indices will be unioned together
pandas.DataFrame.mod

DataFrame.mod \(\text{other}, \text{axis}=\text{columns}, \text{level}=\text{None}, \text{fill\_value}=\text{None}\)

Binary operator mod 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**

Mismatched indices will be unioned together

pandas.DataFrame.pow

DataFrame.pow \(\text{other}, \text{axis}=\text{columns}, \text{level}=\text{None}, \text{fill\_value}=\text{None}\)

Binary operator pow 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**

Mismatched indices will be unioned together

pandas.DataFrame.radd

DataFrame.radd \(\text{other}, \text{axis}=\text{columns}, \text{level}=\text{None}, \text{fill\_value}=\text{None}\)

Binary operator radd 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

Mismatched indices will be unioned together

pandas.DataFrame.rsub

DataFrame.rsub(other, axis='columns', level=None, fill_value=None)

Binary operator rsub 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

Mismatched indices will be unioned together

pandas.DataFrame.rmul

DataFrame.rmul(other, axis='columns', level=None, fill_value=None)

Binary operator rmul 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**  
Mismatched indices will be unioned together

---

**pandas.DataFrame.rdiv**

DataFrame.\texttt{rdiv}(other, axis='columns', level=None, fill_value=None)  
Binary operator rtruediv 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**  
Mismatched indices will be unioned together

---

**pandas.DataFrame.rtruediv**

DataFrame.\texttt{rtruediv}(other, axis='columns', level=None, fill_value=None)  
Binary operator rtruediv 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

Mismatched indices will be unioned together

pandas.DataFrame.rfloordiv

DataFrame.rfloordiv (other, axis='columns', level=None, fill_value=None)

Binary operator rfloordiv 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

Mismatched indices will be unioned together

pandas.DataFrame.rmod

DataFrame.rmod (other, axis='columns', level=None, fill_value=None)

Binary operator rmod 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

Mismatched indices will be unioned together
**pandas.DataFrame.rpow**

`DataFrame.rpow(other, axis='columns', level=None, fill_value=None)`

Binary operator rpow 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**

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)
```
**Returns**  DataFrame

### 28.4.6 Function application, GroupBy

<table>
<thead>
<tr>
<th>pandas.DataFrame.apply</th>
<th>Applies function along input axis of DataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.DataFrame.applymap(funct)</td>
<td>Apply a function to a DataFrame that is intended to operate</td>
</tr>
<tr>
<td>pandas.DataFrame.groupby(by, axis, level, ...)</td>
<td>Group series using mapper (dict or key function, apply given function</td>
</tr>
</tbody>
</table>

#### 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, 1}
  - 0 : apply function to each column
  - 1 : 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

**Additional keyword arguments will be passed as keywords to the function**

**Returns**

- **applied**: Series or DataFrame

**See Also**:

- DataFrame.applymap For elementwise operations

---

**28.4. DataFrame**
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

**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 result >>> data.groupby(func, axis=0).mean()

# DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()

# DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()

**28.4.7 Computations / Descriptive Stats**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.abs()</td>
<td>Return an object with absolute value taken. Only applicable to objects that are all numeric.</td>
</tr>
<tr>
<td>DataFrame.any()</td>
<td>Return whether any element is True over requested axis.</td>
</tr>
<tr>
<td>DataFrame.clip()</td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td>DataFrame.clip_lower()</td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>DataFrame.clip_upper()</td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td>DataFrame.corr()</td>
<td>Compute pairwise correlation of columns, excluding NA/null values.</td>
</tr>
<tr>
<td>DataFrame.corrwith()</td>
<td>Compute pairwise correlation between rows or columns of two DataFrames.</td>
</tr>
<tr>
<td>DataFrame.count()</td>
<td>Return Series with number of non-NA/null observations over requested axis.</td>
</tr>
<tr>
<td>DataFrame.cov()</td>
<td>Compute pairwise covariance of columns, excluding NA/null values.</td>
</tr>
<tr>
<td>DataFrame.cummax()</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>DataFrame.cummin()</td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td>DataFrame.cumprod()</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>DataFrame.cumsum()</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>DataFrame.describe()</td>
<td>Generate various summary statistics of each column, excluding NA/null 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.</td>
</tr>
<tr>
<td>DataFrame.kurt()</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>DataFrame.mean()</td>
<td>Return the mean absolute deviation of the values for the requested axis.</td>
</tr>
<tr>
<td>DataFrame.min()</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>DataFrame.median()</td>
<td>Return the median of the values for the requested axis.</td>
</tr>
<tr>
<td>DataFrame.mode()</td>
<td>Gets the mode 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</td>
</tr>
<tr>
<td>DataFrame.rank()</td>
<td>Compute numerical data ranks (1 through n) along 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>

**DataFrame.abs()**

Returns abs: type of caller
pandas: powerful Python data analysis toolkit, Release 0.13.1

pandas.DataFrame.any

DataFrame.any (axis=None, bool_only=None, skipna=True, level=None, **kwargs)
Return whether any element is True over requested axis. %(na_action)s

Parameters
axis : {0, 1}
  0 for row-wise, 1 for column-wise

skipna : boolean, default True
  Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None
  If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing
  into a DataFrame

bool_only : boolean, default None
  Only include boolean data.

Returns
any : Series (or DataFrame if level specified)

pandas.DataFrame.clip

DataFrame.clip (lower=None, upper=None, out=None)
Trim values at input threshold(s)

Parameters
lower : float, default None

upper : float, default None

Returns
clipped : Series

pandas.DataFrame.clip_lower

DataFrame.clip_lower (threshold)
Return copy of the input with values below given value truncated

Returns
clipped : same type as input

See Also:
clip

pandas.DataFrame.clip_upper

DataFrame.clip_upper (threshold)
Return copy of input with values above given value truncated

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, 1}
    0 to compute column-wise, 1 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, 1}
    0 for row-wise, 1 for column-wise

level : int, 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)}
  - boolean, default True
  - Excludes 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=50)

Generate various summary statistics of each column, excluding NaN values. These include: count, mean, std, min, max, and lower%/50%/upper% percentiles

**Parameters**

- **percentile_width**: float, optional
  - width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

**Returns**

- DataFrame of summary statistics

**pandas.DataFrame.diff**

DataFrame.diff(periods=1)

1st discrete difference of object

**Parameters**

- **periods**: int, default 1
  - Periods to shift for forming difference

**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')
>>> df.eval('c=a + b')
```

**pandas.DataFrame.kurt**

DataFrame.kurt (*axis=None, skipna=None, level=None, numeric_only=None, **kwargs*)

Return unbiased kurtosis 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, 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, **kwargs*)

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, 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, 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, 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, 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)
**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, 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 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.

**Parameters**

**axis** : {0, 1, ‘index’, ‘columns’} (default 0)

- 0/’index’ : get mode of each column
- 1/’columns’ : get mode of each row

**numeric_only** : boolean, default False

if True, only apply to numeric columns

**Returns**

**modes** : DataFrame (sorted)

---

**pandas.DataFrame.pct_change**

DataFrame.pct_change(*periods=1, fill_method=’pad’, limit=None, freq=None, **kwds*)

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 : same type as caller

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, 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 scoreatpercentile in scipy.stats

Parameters q : quantile, default 0.5 (50% quantile)
0 <= q <= 1
axis : {0, 1}
0 for row-wise, 1 for column-wise

Returns quantiles : Series

pandas.DataFrame.rank

DataFrame.rank (axis=0, numeric_only=None, method='average', na_option='keep', ascending=True)
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, 1}, 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'}
• 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

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)

Returns ranks: DataFrame

**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, 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, 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(\(axis=None, \ skipna=None, \ level=None, \ ddf=1, **kwargs\))

Return unbiased standard deviation 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, 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 stdev: Series or DataFrame (if level specified)

**pandas.DataFrame.var**

DataFrame.var(\(axis=None, \ skipna=None, \ level=None, \ ddf=1, **kwargs\))

Return unbiased variance 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, 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 variance: Series or DataFrame (if level specified)

### 28.4.8 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, inplace])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>DataFrame.drop_duplicates([cols, take_last, ...])</td>
<td>Return DataFrame with duplicate rows removed, optionally only</td>
</tr>
<tr>
<td>DataFrame.duplicated([cols, take_last])</td>
<td>Return boolean Series denoting duplicate rows, optionally only</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>
</tbody>
</table>

Continued on next page
Table 28.46 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.head([n])</td>
<td>Returns 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</td>
</tr>
<tr>
<td>DataFrame.reindex_axis(labels[, axis, ...])</td>
<td>Conform input object to new index with optional filling logic,</td>
</tr>
<tr>
<td>DataFrame.reindex_like(other[, method, ...])</td>
<td>return an object with matching indices to myself</td>
</tr>
<tr>
<td>DataFrame.reset_index([other[, method, ...])</td>
<td>For DataFrame with multi-level index, return new DataFrame with</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</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</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 name
    - 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, **kwargs)

Return new object with labels in requested axis removed

**Parameters**

- **labels**: single label or list-like
- **axis**: int or axis name
- **level**: int or name, default None
  For MultiIndex
- **inplace**: bool, default False
  If True, do operation inplace and return None.

**Returns** dropped: type of caller

***pandas.DataFrame.drop_duplicates***

DataFrame.**drop_duplicates**(cols=None, take_last=False, inplace=False)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

**Parameters**

- **cols**: 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.duplicated***

DataFrame.**duplicated**(cols=None, take_last=False)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

**Parameters**

- **cols**: column label or sequence of labels, optional
  Only consider certain columns for identifying duplicates, by default use all of the columns

28.4. DataFrame
**take_last**: boolean, default False

Take the last observed row in a row. Defaults to the first row

**Returns duplicated**: Series

**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

**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, 1}

0 for row-wise, 1 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, 1}

0 for row-wise, 1 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.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**: {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last
valid observation forward to next valid backfill / bfill: use NEXT valid observation
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

**takeable**: boolean, default False

treat the passed as positional values

**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**

**index**: array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating
data

**axis**: {0,1,'index','columns'}

**method**: {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed object. pad / ffill: propagate last valid
observation forward to next valid backfill / bfill: use NEXT valid observation 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(\texttt{(other, method=\text{None}, copy=\text{True}, limit=\text{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 \texttt{s.reindex(index=other.index, columns=other.columns, method=...)}

pandas.DataFrame.rename

DataFrame.rename(\texttt{(index=\text{None}, columns=\text{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 \text{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.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

28.4.9 Missing data handling
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.dropna</td>
<td>Return object with labels on given axis omitted where alternately any or all of the data are missing.</td>
</tr>
<tr>
<td>DataFrame.fillna</td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td>DataFrame.replace</td>
<td>Replace values given in ‘to_replace’ with ‘value’.</td>
</tr>
</tbody>
</table>

### 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, 1}, 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

**DataFrame.fillna** (value=None, method=None, axis=0, inplace=False, limit=None, downcast=None)

Fill NA/NaN values using the specified method.

**Parameters**
- **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
- **value**: scalar, dict, or Series
  - Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series will not be filled). This value cannot be a list.
- **axis**: {0, 1}, default 0
  - 0: fill column-by-column
  - 1: fill row-by-row
- **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

Maximum size gap to forward or backward fill

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 : same type as caller

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: reglexs 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 reglexs otherwise they will match directly. This doesn’t matter much for value since there are only a few possible substitution reglexs 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.

## 28.4.10 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.delevel(*args, **kwargs)</code></td>
<td>Reshape data (produce a “pivot” table) based on column values.</td>
</tr>
<tr>
<td><code>DataFrame.pivot([index, columns, values])</code></td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td><code>DataFrame.sort([columns, column, axis, ...])</code></td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td><code>DataFrame.sort_index([axis, by, ascending, ...])</code></td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td><code>DataFrame.sortlevel([level, axis, ...])</code></td>
<td>Sort multilevel index by chosen axis and primary level.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.48 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.swaplevel(i, j[, axis])</code></td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
<tr>
<td><code>DataFrame.stack([level, dropna])</code></td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a</td>
</tr>
<tr>
<td><code>DataFrame.unstack([level])</code></td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning</td>
</tr>
<tr>
<td><code>DataFrame.T</code></td>
<td>Transpose index and columns</td>
</tr>
<tr>
<td><code>DataFrame.to_panel()</code></td>
<td>Transform long (stacked) format (DataFrame) into wide (3D, Panel)</td>
</tr>
<tr>
<td><code>DataFrame.transpose()</code></td>
<td>Transpose index and columns</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.delevel**

DataFrame.delevel(*args, **kwargs)

**pandas.DataFrame.pivot**

DataFrame.pivot(index=None, columns=None, values=None)

Reshape data (produce a “pivot” table) based on column values. Uses unique values from index / columns to form axes 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(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, column=None, axis=0, ascending=True, inplace=False)
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 or tuple for a nested sort.
ascending : boolean or list, default True
Sort ascending vs. descending. Specify list for multiple sort orders
axis : {0, 1}
Sort index/rows versus columns
inplace : boolean, default False
Sort the DataFrame without creating a new instance

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')
Sort DataFrame either by labels (along either axis) or by the values in a column

Parameters
axis : {0, 1}
Sort index/rows versus columns
by : object
Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

**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

**Returns**  
**sorted** : DataFrame

**Examples**

```python
define result = df.sort_index(by=['A', 'B'], ascending=[True, False])
```

### pandas.DataFrame.sortlevel

DataFrame.sortlevel(\(level=0, \text{axis}=0, \text{ascending}=\text{True, inplace}=False\))

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 : \(\{0, 1\}\)

can ascend : boolean, default True

inplace : boolean, default False

Sort the DataFrame without creating a new instance

**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.

**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

---

28.4. DataFrame
Returns stacked : DataFrame or Series

Examples

```python
>>> s
   a  b
one 1.  2.
two 3.  4.

>>> s.stack()
   one  b
      2
   two  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).

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  two
      a  b
   one  1  2
   two  3  4
dtype: float64

>>> s.unstack(level=-1)
   a  b
one 1  2
   b  2
   c  4

>>> s.unstack(level=0)
   one  two
      a  b
   one 1  3
   two 2  4
```
>>> df = s.unstack(level=0)
>>> df.unstack()

one  a  1.
   b  3.
two  a  2.
   b  4.

pandas.DataFrame.T

DataFrame.T
   Transpose index and columns

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.transpose

DataFrame.transpose()
   Transpose index and columns

28.4.11 Combining / joining / merging

DataFrame.append(other[, ignore_index, ...])  Append columns of other to end of this frame’s columns and index, returning a new object.

   Parameters  other : DataFrame or list of Series/dict-like objects

       ignore_index : boolean, default False

           If True do not use the index labels. Useful for gluing together record arrays

       verify_integrity : boolean, default False

           If True, raise ValueError on creating index with duplicates

   Returns  appended : DataFrame

28.4.  DataFrame
Notes

If a list of dict 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

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 \textit{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

**Examples**

```python
code
>>> 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
```

---

28.4. DataFrame  873
merge(A, B, left_on='lkey', right_on='rkey', how='outer')

<table>
<thead>
<tr>
<th>lkey</th>
<th>value_x</th>
<th>rkey</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 bar</td>
<td>2</td>
<td>bar</td>
<td>6</td>
</tr>
<tr>
<td>1 bar</td>
<td>2</td>
<td>bar</td>
<td>8</td>
</tr>
<tr>
<td>2 baz</td>
<td>3</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3 foo</td>
<td>1</td>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>4 foo</td>
<td>4</td>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>5 NaN</td>
<td>NaN</td>
<td>qux</td>
<td>7</td>
</tr>
</tbody>
</table>

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', 'right', 'outer', 'inner'}, 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.

28.4.12 Time series-related

- **DataFrame.asfreq**(freq[, method, how, normalize]) Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.
- **DataFrame.shift**(periods[, freq, axis]) Shift index by desired number of periods with an optional time freq.
- **DataFrame.first_valid_index**() Return label for first non-NA/null value.
- **DataFrame.last_valid_index**() Return label for last non-NA/null value.
- **DataFrame.resample**(rule[, how, axis, ...]) Convenience method for frequency conversion and resampling of regular time-series data.
- **DataFrame.to_period**(freq[, axis, copy]) Convert DataFrame from DatetimeIndex to PeriodIndex with desired freq.
- **DataFrame.to_timestamp**(freq[, how, axis, copy]) Cast to DatetimeIndex of timestamps, at beginning of period.
- **DataFrame.tz_convert**(tz[, axis, copy]) Convert TimeSeries to target time zone. If it is time zone naive, it
- **DataFrame.tz_localize**(tz[, axis, copy, ...]) Localize tz-naive TimeSeries to target time zone.

DataFrame.asfreq

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=1, freq=None, axis=0, **kwds)

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’)

**Returns**

- **shifted**: same type as caller

**Notes**

If freq is specified then the index values are shifted but the data if not realigned

### 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, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, offset=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 upssampling

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, 1}, 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, 1} default 0

The axis to convert (the index by default)

copy : boolean, default True

If false then underlying input data is not copied
pandas: powerful Python data analysis toolkit, Release 0.13.1

**Returns** df: DataFrame with DatetimeIndex

### pandas.DataFrame.tz_convert

DataFrame.tz_convert (tz, axis=0, copy=True)

Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

**Parameters**
- tz: string or pytz.timezone object
- copy: boolean, default True
  - Also make a copy of the underlying data

### pandas.DataFrame.tz_localize

DataFrame.tz_localize (tz, axis=0, copy=True, infer_dst=False)

Localize tz-naive TimeSeries to target time zone.

**Parameters**
- tz: string or pytz.timezone object
- copy: boolean, default True
  - Also make a copy of the underlying data
- infer_dst: boolean, default False
  - Attempt to infer fall dst-transition times based on order

#### 28.4.13 Plotting

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.boxplot</td>
<td>Make a box plot from DataFrame column/columns optionally grouped (stratified) by one or more columns</td>
</tr>
<tr>
<td>DataFrame.hist</td>
<td>Draw histogram of the DataFrame’s series using matplotlib / pylab.</td>
</tr>
<tr>
<td>DataFrame.plot</td>
<td>Make line, bar, or scatter plots of DataFrame series with the index on the x-axis</td>
</tr>
</tbody>
</table>

### pandas.DataFrame.boxplot

DataFrame.boxplot (column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, **kwds)

Make a box plot from DataFrame column/columns optionally grouped (stratified) by one or more columns

**Parameters**
- data: DataFrame
  - column: column names 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 axis object, default None
  - fontsize: int or string
  - rot: int, default None
    - Rotation for ticks
  - grid: boolean, default None (matlab style default)
    - Axis grid lines
pandas: powerful Python data analysis toolkit, Release 0.13.1

Returns `ax`: matplotlib.axes.AxesSubplot

**pandas.DataFrame.hist**

DataFrame.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, **kwds)

Draw histogram of the DataFrame's series using matplotlib / pylab.

**Parameters**

- `data`: DataFrame
- `column`: string or sequence
  - If passed, will be used to limit data to a subset of columns
- `by`: object, optional
  - If passed, then used to form histograms for separate groups
- `grid`: boolean, default True
  - Whether to show axis grid lines
- `xlabelsize`: int, default None
  - If specified changes the x-axis label size
- `xrot`: float, default None
  - rotation of x axis labels
- `ylabelsize`: int, default None
  - If specified changes the y-axis label size
- `yrot`: float, default None
  - rotation of y axis labels
- `ax`: matplotlib axes object, default None
  - sharex: bool, if True, the X axis will be shared amongst all subplots.
- `sharey`: bool, if True, the Y axis will be shared amongst all subplots.
- `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
- `kwds`: other plotting keyword arguments
  - To be passed to hist function

**pandas.DataFrame.plot**

DataFrame.plot(frame=None, x=None, y=None, subplots=False, sharex=True, sharey=False, use_index=True, figsize=None, grid=None, legend=True, rot=None, ax=None, style=None, title=None, xlim=None, ylim=None, logx=False, logy=False, xticks=None, yticks=None, kind='line', sort_columns=False, fontsize=None, secondary_y=False, **kwds)

Make line, bar, or scatter plots of DataFrame series with the index on the x-axis using matplotlib / pylab.
Parameters frame : DataFrame
  x : label or position, default None
  y : label or position, default None
    Allows plotting of one column versus another
subplots : boolean, default False
    Make separate subplots for each time series
sharex : boolean, default True
    In case subplots=True, share x axis
sharey : boolean, default False
    In case subplots=True, share y axis
use_index : boolean, default True
    Use index as ticks for x axis
stacked : boolean, default False
    If True, create stacked bar plot. Only valid for DataFrame input
sort_columns : boolean, default False
    Sort column names to determine plot ordering
title : string
    Title to use for the plot
grid : boolean, default None (matlab style default)
    Axis grid lines
legend : boolean, default True
    Place legend on axis subplots
ax : matplotlib axis object, default None
style : list or dict
    matplotlib line style per column
kind : {'line', 'bar', 'barh', 'kde', 'density', 'scatter'}
    bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estima-
    tion plot scatter: scatter plot
logx : boolean, default False
    For line plots, use log scaling on x axis
logy : boolean, default False
    For line plots, use log scaling on y axis
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

**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, should the legend label the axis of the various columns automatically

**colormap**: str or matplotlib colormap object, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

**kwds**: keywords

Options to pass to matplotlib plotting method

**Returns**

**ax_or_axes**: matplotlib.AxesSubplot or list of them

### 28.4.14 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.from_csv</td>
<td>Read delimited file into DataFrame</td>
</tr>
<tr>
<td>DataFrame.from_dict</td>
<td>Construct DataFrame from dict of array-like or dicts</td>
</tr>
<tr>
<td>DataFrame.from_items</td>
<td>Convert (key, value) pairs to DataFrame. The keys will be the axis</td>
</tr>
<tr>
<td>DataFrame.from_records</td>
<td>Convert structured or record ndarray to DataFrame</td>
</tr>
<tr>
<td>DataFrame.info</td>
<td>Concise summary of a DataFrame</td>
</tr>
<tr>
<td>DataFrame.to_pickle</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>DataFrame.to_csv</td>
<td>Write DataFrame to a comma-separated values (csv) file</td>
</tr>
<tr>
<td>DataFrame.to_hdf</td>
<td>activate the HDFStore</td>
</tr>
<tr>
<td>DataFrame.to_dict</td>
<td>Convert DataFrame to dictionary</td>
</tr>
<tr>
<td>DataFrame.to_excel</td>
<td>Write DataFrame to a excel sheet</td>
</tr>
<tr>
<td>DataFrame.to_json</td>
<td>Convert the object to a JSON string</td>
</tr>
<tr>
<td>DataFrame.to_html</td>
<td>Render a DataFrame as an HTML table</td>
</tr>
<tr>
<td>DataFrame.to_latex</td>
<td>Render a DataFrame to a tabular environment table</td>
</tr>
<tr>
<td>DataFrame.to_stata</td>
<td>A class for writing Stata binary dta files from array-like objects</td>
</tr>
<tr>
<td>DataFrame.to_records</td>
<td>Convert DataFrame to record array. Index will be put in the</td>
</tr>
<tr>
<td>DataFrame.to_sparse</td>
<td>Convert to SparseDataFrame</td>
</tr>
<tr>
<td>DataFrame.to_string</td>
<td>Render a DataFrame to a console-friendly tabular output.</td>
</tr>
<tr>
<td>DataFrame.to_clipboard</td>
<td>Attempt to write text representation of object to the system clipboard</td>
</tr>
</tbody>
</table>

**DataFrame.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’.

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=True, buf=None, max_cols=None)
Concise summary of a DataFrame.

Parameters verbose : boolean, default True
If False, don’t print column count summary

buf : writable buffer, defaults to sys.stdout

max_cols : int, default None
Determines whether full summary or short summary is printed

pandas.DataFrame.to_pickle

DataFrame.to_pickle(path)
Pickle (serialize) object to input file path

Parameters path : string
File path

**pandas.DataFrame.to_csv**

```python
DataFrame.to_csv(path_or_buf, sep=', ', na_rep=''; float_format=None, cols=None, header=True, index=True, index_label=None, mode='w', encoding=None, nanRep=None, quot=ing=None, line_terminator='n', chunksize=None, tupleize_cols=False, date_format=None, **kwds)
```

Write DataFrame to a comma-separated values (csv) file

**Parameters**

- **path_or_buf**: string or file handle / StringIO
  - File path
- **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
- **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, 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 if the contents are non-ascii, for python versions prior to 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
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.

```
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_dict

DataFrame.to_dict(outtype='dict')
Convert DataFrame to dictionary.

Parameters outtype : str {'dict', 'list', 'series', 'records'}
 Determines the type of the values of the dictionary. The default dict is a nested dictionary {column -> {index -> value}}. list returns {column -> list(values)}. series returns {column -> Series(values)}. records returns [{columns -> value}]. Abbreviations are allowed.

Returns result : dict like {column -> {index -> value}}

pandas.DataFrame.to_excel

DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, cols=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True)
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

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

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.

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(buf=None, columns=None, col_space=None, colSpace=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparse=False, index_names=True, justify=None, force_ascii=None, bold_rows=True, classes=None, escape=True, max_rows=None, max_cols=None, show_dimensions=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

---

28.4. DataFrame
formatter functions to apply to columns’ elements by position or name, default None, if the result is a string, it 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

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_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, force_unicode=None)

Render a DataFrame to a tabular environment table. You can splice this into a LaTeX document.

to_latex-specific options:

bold_rows [boolean, default True] Make the row labels bold in the 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
pandas: powerful Python data analysis toolkit, Release 0.13.1

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, if the result is a string , it 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

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, convert_dates=None, write_index=True, encoding='latin-1', byteorder=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
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()

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_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, nanRep=None, index_names=True, justify=None, force_unicode=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, if the result is a string, it 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

**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_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
28.5 Panel

28.5.1 Constructor

```python
Panel([data, items, major_axis, minor_axis, ...]) Represents wide format panel data, stored as 3-dimensional array
```

```python
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
- `copy`: boolean, default False

Copy data from inputs. Only affects DataFrame / 2d ndarray input.

**Attributes**
- `at`
- `axes`: index(es) of the NDFrame
- `blocks`: Internal property, property synonym for as_blocks()
- `dtypes`: Return the dtypes in this object
- `empty`: True if NDFrame is entirely empty [no items]
- `ftypes`: Return the ftypes (indication of sparse/dense and dtype)
- `iat`
- `iloc`
- `ix`
- `loc`
- `ndim`: Number of axes / array dimensions
- `shape`: tuple of axis dimensions
- `values`: Numpy representation of NDFrame

```python
pandas.Panel.at
```

```python
Panel.at
```
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

pandas.Panel.iloc

Panel.iloc

pandas.Panel.ix

Panel.ix

pandas.Panel.loc

Panel.loc

pandas.Panel.ndim

Panel.ndim
    Number of axes / array dimensions
### pandas.Panel.shape

**Panel.shape**

tuple of axis dimensions

### pandas.Panel.values

**Panel.values**

Numpy representation of NDFrame

<table>
<thead>
<tr>
<th>Method</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>Wrapper method for 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>apply(func[, axis])</td>
<td>Applies function along input axis of the Panel</td>
</tr>
<tr>
<td>as_blocks((columns))</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has</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</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.</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>
<tr>
<td>bool()</td>
<td>Return the bool of a single element PandasObject</td>
</tr>
<tr>
<td>clip([lower, upper, out])</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>clip_lower(threshold)</td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>clip_upper(threshold)</td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td>compound([axis, skipna, level])</td>
<td>Return the compound percentage of the values for the requested axis</td>
</tr>
<tr>
<td>conform(frame[, axis])</td>
<td>Conform input DataFrame to align with chosen axis pair.</td>
</tr>
<tr>
<td>consolidate([inplace])</td>
<td>Compute NDFrame with &quot;consolidated&quot; internals (data of each dtype</td>
</tr>
<tr>
<td>convert_objects([convert_dates, ...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>count([axis])</td>
<td>Return number of observations over requested axis.</td>
</tr>
<tr>
<td>cummax([axis, dtype, out, skipna])</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>cummin([axis, dtype, out, skipna])</td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td>cumprod([axis, dtype, out, skipna])</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>cumsum([axis, dtype, out, skipna])</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>div(other[, axis])</td>
<td>Wrapper method for truediv</td>
</tr>
<tr>
<td>divide(other[, axis])</td>
<td></td>
</tr>
<tr>
<td>drop(labels[, axis, level, inplace])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>dropna([axis, how, inplace])</td>
<td>Drop 2D from panel, holding passed axis constant</td>
</tr>
<tr>
<td>eq(other)</td>
<td>Wrapper for comparison method eq</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements. NaNs in the</td>
</tr>
<tr>
<td>fillna([axis, inplace, limit, downcast])</td>
<td>Synonym for NDFrame.fillna(method='ffill')</td>
</tr>
<tr>
<td>fillna([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.55 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Restrict the info axis to set of items or wildcard</td>
</tr>
<tr>
<td>first</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>floordiv</td>
<td>Wrapper method for floordiv</td>
</tr>
<tr>
<td>fromDict</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>from_dict</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>ge</td>
<td>Wrapper for comparison method ge</td>
</tr>
<tr>
<td>get</td>
<td>Get item from object for given key (DataFrame column, Panel slice,</td>
</tr>
<tr>
<td>get_dtype_counts</td>
<td>Return the counts of dtypes in this object</td>
</tr>
<tr>
<td>get_ftype_counts</td>
<td>Return the counts of ftypes in this object</td>
</tr>
<tr>
<td>get_value</td>
<td>Quickly retrieve single value at (item, major, minor) location</td>
</tr>
<tr>
<td>get_values</td>
<td>Same as values (but handles sparseness conversions)</td>
</tr>
<tr>
<td>groupby</td>
<td>Group data on given axis, returning GroupBy object</td>
</tr>
<tr>
<td>gt</td>
<td>Wrapper for comparison method gt</td>
</tr>
<tr>
<td>head([n])</td>
<td>Interpolate values according to different methods.</td>
</tr>
<tr>
<td>isnull</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>iteritems</td>
<td>Iterate over (label, values) on info axis</td>
</tr>
<tr>
<td>iterkv</td>
<td>Iterates alias used to get around 2to3. Deprecated</td>
</tr>
<tr>
<td>join</td>
<td>Join items with other Panel either on major and minor axes column</td>
</tr>
<tr>
<td>keys</td>
<td>Get the ‘info axis’ (see Indexing for more)</td>
</tr>
<tr>
<td>kurt</td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td>kurtosis</td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td>last</td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td>le</td>
<td>Wrapper for comparison method le</td>
</tr>
<tr>
<td>load(path)</td>
<td>Deprecated</td>
</tr>
<tr>
<td>lt</td>
<td>Wrapper for comparison method lt</td>
</tr>
<tr>
<td>mad</td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td>major_xs(key[, copy])</td>
<td>Return slice of panel along major axis</td>
</tr>
<tr>
<td>mask</td>
<td>Returns copy whose values are replaced with nan if the</td>
</tr>
<tr>
<td>max</td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td>mean</td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td>median</td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td>min</td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td>minor_xs(key[, copy])</td>
<td>Return slice of panel along minor axis</td>
</tr>
<tr>
<td>mod</td>
<td>Wrapper method for mod</td>
</tr>
<tr>
<td>mul</td>
<td>Wrapper method for mul</td>
</tr>
<tr>
<td>multiply</td>
<td>Wrapper method for mul</td>
</tr>
<tr>
<td>ne</td>
<td>Wrapper for comparison method ne</td>
</tr>
<tr>
<td>notnull</td>
<td>Return a boolean same-sized object indicating if the values are</td>
</tr>
<tr>
<td>pct_change</td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td>pop(item)</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>pow</td>
<td>Wrapper method for pow</td>
</tr>
<tr>
<td>prod</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>product</td>
<td>Return the product of the values for the requested axis</td>
</tr>
<tr>
<td>radd</td>
<td>Wrapper method for radd</td>
</tr>
<tr>
<td>rdiv</td>
<td>Wrapper method for rtruediv</td>
</tr>
<tr>
<td>reindex</td>
<td>Conform Panel to new index with optional filling logic, placing</td>
</tr>
<tr>
<td>reindex_axis</td>
<td>Conform Panel to new index with optional filling logic, placing</td>
</tr>
<tr>
<td>reindex_like</td>
<td>Return an object with matching indicies to myself</td>
</tr>
<tr>
<td>rename</td>
<td>Alter axes input function or functions.</td>
</tr>
</tbody>
</table>
### Table 28.55 – continued from previous page

- **rename_axis**(mapper[, axis, copy, inplace]) Alter index and/or columns using input function or functions.
- **replace**(to_replace, value, inplace, limit, ...) Replace values given in ‘to_replace’ with ‘value’.
- **resample**(rule[, how, axis, fill_method, ...]) Convenience method for frequency conversion and resampling of regular time-series data.
- **rfloordiv**(other[, axis]) Wrapper method for rfloordiv
- **rmod**(other[, axis]) Wrapper method for rmod
- **rmul**(other[, axis]) Wrapper method for rmul
- **rpow**(other[, axis]) Wrapper method for rpow
- **rsub**(other[, axis]) Wrapper method for rsub
- **rtruediv**(other[, axis]) Wrapper method for rtruediv
- **save**(path) Deprecated.
- **select**(crit[, axis]) Return data corresponding to axis labels matching criteria
- **set_value**(args) Quickly set single value at (item, major, minor) location
- **shift**(lags[, freq, axis]) Shift major or minor axis by specified number of leads/lags.
- **skew**(axis, skipna, level, numeric_only) Return unbiased skew over requested axis
- **sort_index**(axis, ascending) Sort object by labels (along an axis)
- **squeeze**() squeeze length 1 dimensions
- **std**(axis, skipna, level, numeric_only) Return unbiased standard deviation over requested axis
- **sub**(other[, axis]) Wrapper method for sub
- **subtract**(other[, axis]) Wrapper method for sub
- **sum**(axis, skipna, level, numeric_only) Return the sum of the values for the requested axis
- **swapaxes**(axis1, axis2[, copy]) Interchange axes and swap values axes appropriately
- **swaplevel**(i, j[, axis]) Swap levels i and j in a MultiIndex on a particular axis
- **tail**(n) Swap levels i and j in a MultiIndex on a particular axis
- **take**(indices[, axis, convert, is_copy]) Analogous to ndarray.take
- **toLong**(args, **kwargs) Attempt to write text representation of object to the system clipboard
- **to_dense**() Return dense representation of NDFrame (as opposed to sparse)
- **to_frame**(filter_observations) Transform wide format into long (stacked) format as DataFrame whose
- **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_long**(args, **kwargs) Convert object to a JSON string.
- **to_msgpack**(path_or_buf) msgpack (serialize) object to input file path
- **to_pickle**(path) Pickle (serialize) object to input file path
- **transpose**(args, **kwargs) Permute the dimensions of the Panel
- **truediv**(other[, axis]) Wrapper method for truediv
- **truncate**([before, after, axis, copy]) Truncates a sorted NDFrame before and/or after some particular
- **tz_convert**(tz[, axis, copy]) Convert TimeSeries to target time zone. If it is time zone naive, it
- **tz_localize**(tz[, axis, copy, infer_dst]) Localize tz-naive TimeSeries to target time zone
- **update**(other[, join, overwrite, ...]) Modify Panel in place using non-NA values from passed
- **var**(axis, skipna, level, ddof) Return unbiased variance over requested axis
- **where**(cond[, other, inplace, axis, level, ...]) Return an object of same shape as self and whose corresponding
- **xs**(key[, axis, copy]) Return slice of panel along selected axis

**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)
Wrapper method for add
Parameters
  other : DataFrame or Panel
  axis : [items, major_axis, minor_axis]
  Axis to broadcast over
Returns  Panel

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 name
  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.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

>>> 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(columns=None)
Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
are presented in sorted order unless a specific list of columns is provided.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters
columns: array-like
Specific column order

Returns values: a list of Object

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)
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

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=0, 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)
    Trim values at input threshold(s)
    Parameters  lower : float, default None
                  upper : float, default None
    Returns  clipped : Series

pandas.Panel.clip_lower

Panel.clip_lower (threshold)
    Return copy of the input with values below given value truncated
    Returns  clipped : same type as input
    See Also:
    clip

pandas.Panel.clip_upper

Panel.clip_upper (threshold)
    Return copy of input with values above given value truncated
    Returns  clipped : same type as input
    See Also:
    clip

pandas.Panel.compound

Panel.compound (axis=None, skipna=None, level=None, **kwargs)
    Return the compound percentage of the values for the requested axis
    Parameters  axis : {items (0), major_axis (1), minor_axis (2)}
                  skipna : boolean, default True
    Exclude NA/null values. If an entire row/column is NA, the result will be NA
**level**: int, 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 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.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: if True, attempt to soft convert dates, if ‘coerce’, force conversion (and non-convertibles get NaT)

convert_numeric: if True attempt to coerce to numbers (including strings), non-convertibles get NaN

convert_timedeltas: if True, attempt to soft convert timedeltas, if ‘coerce’, force conversion (and non-convertibles get NaT)

copy: Boolean, if True, return copy, default is True

**Returns** converted: asm as input object
**pandas.Panel.copy**

Panel.copy(deep=True)

Make a copy of this object

Parameters  
- **deep**: boolean, 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: powerful Python data analysis toolkit, Release 0.13.1

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.div

Panel.div(other, axis=0)
Wrapper method for truediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.divide

Panel.divide(other, axis=0)
Wrapper method for truediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.drop

Panel.drop(labels, axis=0, level=None, inplace=False, **kwargs)
Return new object with labels in requested axis removed

Parameters
labels : single label or list-like
axis : int or axis name
level : int or name, default None
For MultiIndex
inplace : bool, default False
If True, do operation inplace and return None.

Returns
dropped : type of caller
pandas.Panel.dropna

Panel.dropna (axis=0, how='any', inplace=False, **kwargs)
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

pandas.Panel.ffill

Panel.ffill (axis=0, inplace=False, limit=None, downcast=None)
Synonym for NDFrame.fillna(method='ffill')

pandas.Panelfillna

Panel.fillna (value=None, method=None, axis=0, inplace=False, limit=None, downcast=None)
Fill NA/NaN values using the specified method

Parameters
- **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
- **value**: scalar, dict, or Series
  Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying
  which value to use for each index (for a Series) or column (for a DataFrame).
  (values not in the dict/Series will not be filled). This value cannot be a list.
- **axis**: {0, 1}, default 0
• 0: fill column-by-column
• 1: fill row-by-row

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
Maximum size gap to forward or backward fill

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 : same type as caller

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

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)
Wrapper method for floordiv

Parameters other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}
    Axis to broadcast over

Returns Panel

pandas.Panel.fromDict

classmethod Panel.fromDict(data, intersect=False, orient='items', dtype=None)
Construct Panel from dict of DataFrame objects

Parameters data : dict
    {field : DataFrame}
    intersect : boolean
    Intersect indexes of input DataFrames
    orient : {'items', 'minor'}, default 'items'
    The “orientation” of the data. If the keys of the passed dict should be the items of
    the result panel, pass ‘items’ (default). Otherwise if the columns of the values of
    the passed DataFrame objects should be the items (which in the case of mixed-
    dtype data you should do), instead pass ‘minor’

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’

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)
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)

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

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='infer', **kwargs)
Interpolate values according to different methods.

Parameters method : {'linear', 'time', 'values', 'index' '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’: 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 ‘infer’
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 2 2 3 3
dtype: float64
```

pandas.Panel.isnull

Panel.isnull()
Return a boolean same-sized object indicating if the values are null

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
rsuffix : string
Suffix to use from right frame’s overlapping columns

Returns joined : Panel
pandas.Panel.keys

Panel.keys()
Get the ‘info axis’ (see Indexing for more)
This is index for Series, columns for DataFrame and major_axis for Panel.

pandas.Panel.kurt

Panel.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis 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, 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
kurt : DataFrame or Panel (if level specified)

pandas.Panel.kurtosis

Panel.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
Return unbiased kurtosis over requested axis 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, 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
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

```python
ts.last('5M') -> Last 5 months
```

**pandas.Panel.le**

```python
Panel.le(other)
Wrapper for comparison method le
```

**pandas.Panel.load**

```python
Panel.load(path)
Depreciated. Use read_pickle instead.
```

**pandas.Panel.lt**

```python
Panel.lt(other)
Wrapper for comparison method lt
```

**pandas.Panel.mad**

```python
Panel.mad(axis=None, skipna=None, level=None, **kwargs)
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, 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**

```python
Panel.major_xs(key, copy=True)
Return slice of panel along major axis
```

**Parameters**

- `key` : object
  - Major axis label
- `copy` : boolean, default True
Copy data

Returns y : DataFrame
index -> minor axis, columns -> items

pandas.Panel.mask

Panel.mask (cond)
Returns copy whose values are replaced with nan if the inverted condition is True

Parameters cond : boolean NDFrame or array

Returns wh: same as input

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, 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, 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, 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, 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=True)
Return slice of panel along minor axis

Parameters
key : object
   Minor axis label
copy : boolean, default True
   Copy data

Returns
y : DataFrame
   index -> major axis, columns -> items
pandas.Panel.mod

Panel.mod(other, axis=0)
Wrapper method for mod

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.mul

Panel.mul(other, axis=0)
Wrapper method for mul

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.multiply

Panel.multiply(other, axis=0)
Wrapper method for mul

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.ne

Panel.ne(other)
Wrapper for comparison method ne

pandas.Panel.notnull

Panel.notnull()
Return a boolean same-sized object indicating if the values are not null

pandas.Panel.pct_change

Panel.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwds)
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**: same type as caller

### 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)**

Wrapper method for pow

**Parameters**

other : DataFrame or Panel

axis : {items, major_axis, minor_axis}

**Returns**

Panel

### 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, 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, 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)  
Wrapper method for radd

**Parameters**  
other : DataFrame or Panel  
axis : {items, major_axis, minor_axis}

**Returns**  
Panel

**pandas.Panel.rdiv**

Panel.rdiv(other, axis=0)  
Wrapper method for rtruediv

**Parameters**  
other : DataFrame or Panel  
axis : {items, major_axis, minor_axis}

**Returns**  
Panel

**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** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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

**takeable** : boolean, default False

treat the passed as positional values

**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=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**

**index** : array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : {0,1,2,'items','major_axis','minor_axis'}

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed object. pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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 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.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 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.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

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)
Wrapper method for rfloordiv

Parameters

other : DataFrame or Panel

axis : {'items', 'major_axis', 'minor_axis'}

Axis to broadcast over

Returns

Panel

pandas.Panel.rmod

Panel.rmod(other, axis=0)
Wrapper method for rmod
**Parameters**

- **other**: DataFrame or Panel
- **axis**: \{items, major_axis, minor_axis\}

**Returns**

Panel

---

**pandas.Panel.rmul**

Panel._rmul mnemonic

- **Parameters**
  - **other**: DataFrame or Panel
  - **axis**: \{items, major_axis, minor_axis\}

- **Returns**
  - Panel

---

**pandas.Panel.rpow**

Panel._rpow mnemonic

- **Parameters**
  - **other**: DataFrame or Panel
  - **axis**: \{items, major_axis, minor_axis\}

- **Returns**
  - Panel

---

**pandas.Panel.rsub**

Panel._rsub mnemonic

- **Parameters**
  - **other**: DataFrame or Panel
  - **axis**: \{items, major_axis, minor_axis\}

- **Returns**
  - Panel

---

**pandas.Panel.rtruediv**

Panel._rtruediv mnemonic

- **Parameters**
  - **other**: DataFrame or Panel
  - **axis**: \{items, major_axis, minor_axis\}

- **Returns**
  - Panel
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.set_value

Panel.set_value(*args)
    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
    Returns  panel : Panel
                  If label combo is contained, will be reference to calling Panel, otherwise a new object

pandas.Panel.shift

Panel.shift(lags, freq=None, axis='major')
    Shift major or minor axis by specified number of leads/lags. Drops periods right now compared with DataFrame.shift
    Parameters  lags : int
                  axis : {'major', 'minor'}
    Returns  shifted : Panel

pandas.Panel.skew

Panel.skew(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
    Return unbiased skew over requested axis Normalized by N-1
    Parameters  axis : {items (0), major_axis (1), minor_axis (2)}
                   skipna : boolean, default True
                   Exclude NA/null values. If an entire row/column is NA, the result will be NA
level : int, 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.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, **kwargs)
Return unbiased standard deviation 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, 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 stdev : DataFrame or Panel (if level specified)
pandas.Panel.sub

Panel.sub (other, axis=0)
Wrapper method for sub

Parameters
other: DataFrame or Panel
axis: {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.subtract

Panel.subtract (other, axis=0)
Wrapper method for sub

Parameters
other: DataFrame or Panel
axis: {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

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
Excluding NA/null values. If an entire row/column is NA, the result will be NA
level: int, 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

Panel.swapaxes (axis1, axis2, copy=True)
Interchange axes and swap values axes appropriately

Returns
y: same as input
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.tail

Panel.tail(n=5)

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.toLong

Panel.toLong(*args, **kwargs)

pandas.Panel.to_clipboard

Panel.to_clipboard(excel=None, sep=None, **kwargs)
Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

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
pandas: powerful Python data analysis toolkit, Release 0.13.1

- 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

**Other Parameters**

- **float_format**: string, default None
  - Format string for floating point numbers
- **cols**: sequence, optional
  - Columns to write
- **header**: boolean or list of string, default True
  - Write out column names. If a list of string is given it is assumed to be aliases for the column names
- **index**: boolean, default True
  - Write row names (index)
- **index_label**: string or sequence, default None
  - Column label for index column(s) if desired. If None is given, and `header` and `index` are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- **startrow**: upper left cell row to dump data frame
- **startcol**: upper left cell column to dump data frame

**Notes**

Keyword arguments (and `na_rep`) are passed to the `to_excel` method for each DataFrame written.
pandas.Panel.to_frame

Panel.to_frame(filter_observations=True)
Transform wide format into long (stacked) format as DataFrame whose columns are the Panel’s items and whose index is a MultiIndex formed of the Panel’s major and minor axes.

Parameters filter_observations : boolean, default True
Drop (major, minor) pairs without a complete set of observations across all the items

Returns y : DataFrame

pandas.Panel.to_hdf

Panel.to_hdf(path_or_buf, key, **kwargs)
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 comprlib is specified compression will be applied where possible
complib : {‘zlib’, ‘bz2’, ‘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(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: powerful Python data analysis toolkit, Release 0.13.1

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.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**

```python
Panel.truediv(other, axis=0)
```

Wrapper method for `truediv`

- **Parameters**
  - `other` : DataFrame or Panel
  - `axis` : {items, major_axis, minor_axis}

- **Returns**
  - Panel

**pandas.Panel.truncate**

```python
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

**pandas.Panel.tshift**

```python
Panel.tshift(periods=1, freq=None, axis='major', **kwds)
```

**pandas.Panel.tz_convert**

```python
Panel.tz_convert(tz, axis=0, copy=True)
```

Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

- **Parameters**
  - `tz` : string or pytz.timezone object
  - `copy` : boolean, default True
    - Also make a copy of the underlying data
pandas.Panel.tz_localize

Panel.tz_localize(tz, axis=0, copy=True, infer_dst=False)
Localize tz-naive TimeSeries to target time zone

Parameters  

tz : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

infer_dst : boolean, default False

Attempt to infer fall dst-transition times based on order

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.

pandas.Panel.var

Panel.var(axis=None, skipna=None, level=None, ddof=1, **kwargs)
Return unbiased variance 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, 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  

variance : 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=True)

Return slice of panel along selected axis

Parameters:
- key : object
  Label
- axis : {‘items’, ‘major’, ‘minor’}, default 1/’major’
- copy : boolean, default True
  Copy data

Returns:
- y : ndim(self)-1

28.5.2 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>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.values</td>
<td>Numpy representation of NDFrame</td>
</tr>
<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.shape</td>
<td>tuple of axis dimensions</td>
</tr>
<tr>
<td>Panel.dtypes</td>
<td>Return the dtypes in this object</td>
</tr>
</tbody>
</table>
pandas.Panel.values

Panel.values
   Numpy representation of NDFrame

pandas.Panel.axes

Panel.axes
   index(es) of the NDFrame

pandas.Panel.ndim

Panel.ndim
   Number of axes / array dimensions

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

28.5.3 Conversion
pandas: powerful Python data analysis toolkit, Release 0.13.1

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.astype</code></td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td><code>Panel.copy</code></td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td><code>Panel.isnull</code></td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td><code>Panel.notnull</code></td>
<td>Return a boolean same-sized object indicating if the values are not null</td>
</tr>
</tbody>
</table>

### pandas.Panel.astype

**Panel.astype** *(dtype, copy=True, raise_on_error=True)*

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

**Returns**
casted : type of caller

### pandas.Panel.copy

**Panel.copy** *(deep=True)*

Make a copy of this object

**Parameters**
- **deep**: boolean, default True
  Make a deep copy, i.e. also copy data

**Returns**
copy : type of caller

### pandas.Panel.isnull

**Panel.isnull**

Return a boolean same-sized object indicating if the values are null

### pandas.Panel.notnull

**Panel.notnull**

Return a boolean same-sized object indicating if the values are not null

28.5.4 Getting and setting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.get_value</code></td>
<td>Quickly retrieve single value at (item, major, minor) location</td>
</tr>
<tr>
<td><code>Panel.set_value</code></td>
<td>Quickly set single value at (item, major, minor) location</td>
</tr>
</tbody>
</table>

### pandas.Panel.get_value

**Panel.get_value** *(*args)*

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)
Returns value: scalar value

pandas.Panel.set_value

Panel.set_value(*args)
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

Returns panel: Panel
If label combo is contained, will be reference to calling Panel, otherwise a new object

28.5.5 Indexing, iteration, slicing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.at</td>
<td>Iterate over info axis</td>
</tr>
<tr>
<td>Panel.iat</td>
<td>Iterate over (label, values) on info axis</td>
</tr>
<tr>
<td>Panel.ix</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>Panel.loc</td>
<td>Return slice of panel along selected axis</td>
</tr>
<tr>
<td>Panel.loc</td>
<td>Return slice of panel along major axis</td>
</tr>
<tr>
<td>Panel.loc</td>
<td>Return slice of panel along minor axis</td>
</tr>
</tbody>
</table>

pandas.Panel.at

Panel.at

pandas.Panel.iat

Panel.iat

pandas.Panel.ix

Panel.ix

pandas.Panel.loc

Panel.loc
pandas.Panel.iloc

Panel.iloc

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=True)
    Return slice of panel along selected axis

Parameters
    key : object
        Label
    axis : {'items', 'major', 'minor'}, default 1/'major'
    copy : boolean, default True
        Copy data

Returns
    y : ndim(self)-1

pandas.Panel.major_xs

Panel.major_xs(key, copy=True)
    Return slice of panel along major axis

Parameters
    key : object
        Major axis label
    copy : boolean, default True
        Copy data

Returns
    y : DataFrame
        index -> minor axis, columns -> items
pandas.Panel.minor_xs

Panel.minor_xs(key, copy=True)
Return slice of panel along minor axis

Parameters key : object
Minor axis label
copy : boolean, default True
Copy data

Returns y : DataFrame
index -> major axis, columns -> items

For more information on .at, .iat, .ix, .loc, and .iloc, see the indexing documentation.

28.5.6 Binary operator functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.add(other[, axis])</td>
<td>Wrapper method for add</td>
</tr>
<tr>
<td>Panel.sub(other[, axis])</td>
<td>Wrapper method for sub</td>
</tr>
<tr>
<td>Panel.mul(other[, axis])</td>
<td>Wrapper method for mul</td>
</tr>
<tr>
<td>Panel.div(other[, axis])</td>
<td>Wrapper method for truediv</td>
</tr>
<tr>
<td>Panel.truediv(other[, axis])</td>
<td>Wrapper method for truediv</td>
</tr>
<tr>
<td>Panel.floordiv(other[, axis])</td>
<td>Wrapper method for floordiv</td>
</tr>
<tr>
<td>Panel.mod(other[, axis])</td>
<td>Wrapper method for mod</td>
</tr>
<tr>
<td>Panel.pow(other[, axis])</td>
<td>Wrapper method for pow</td>
</tr>
<tr>
<td>Panel.radd(other[, axis])</td>
<td>Wrapper method for radd</td>
</tr>
<tr>
<td>Panel.rsub(other[, axis])</td>
<td>Wrapper method for rsub</td>
</tr>
<tr>
<td>Panel.rmul(other[, axis])</td>
<td>Wrapper method for rmul</td>
</tr>
<tr>
<td>Panel.rdiv(other[, axis])</td>
<td>Wrapper method for rtruediv</td>
</tr>
<tr>
<td>Panel.rtruediv(other[, axis])</td>
<td>Wrapper method for rtruediv</td>
</tr>
<tr>
<td>Panel.rfloordiv(other[, axis])</td>
<td>Wrapper method for rfloordiv</td>
</tr>
<tr>
<td>Panel.rmod(other[, axis])</td>
<td>Wrapper method for rmod</td>
</tr>
<tr>
<td>Panel.rpow(other[, axis])</td>
<td>Wrapper method for 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)
Wrapper method for add

Parameters other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel
pandas.Panel.sub

Panel.sub(other, axis=0)
Wrapper method for sub

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Returns Panel

pandas.Panel.mul

Panel.mul(other, axis=0)
Wrapper method for mul

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Returns Panel

pandas.Panel.div

Panel.div(other, axis=0)
Wrapper method for truediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Returns Panel

pandas.Panel.truediv

Panel.truediv(other, axis=0)
Wrapper method for truediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Returns Panel

pandas.Panel.floordiv

Panel.floordiv(other, axis=0)
Wrapper method for floordiv
**pandas: powerful Python data analysis toolkit, Release 0.13.1**

```python
Parameters
    other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}

Returns
    Panel
```

**pandas.Panel.mod**

```python
Panel.mod(other, axis=0)
Wrapper method for mod

Parameters
    other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}

Returns
    Panel
```

**pandas.Panel.pow**

```python
Panel.pow(other, axis=0)
Wrapper method for pow

Parameters
    other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}

Returns
    Panel
```

**pandas.Panel.radd**

```python
Panel.radd(other, axis=0)
Wrapper method for radd

Parameters
    other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}

Returns
    Panel
```

**pandas.Panel.rsub**

```python
Panel.rsub(other, axis=0)
Wrapper method for rsub

Parameters
    other : DataFrame or Panel
    axis : {items, major_axis, minor_axis}

Returns
    Panel
```
pandas.Panel.rmul

Panel.rmul(other, axis=0)
Wrapper method for rmul

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.rdiv

Panel.rdiv(other, axis=0)
Wrapper method for rtruediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.rtruediv

Panel.rtruediv(other, axis=0)
Wrapper method for rtruediv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.rfloordiv

Panel.rfloordiv(other, axis=0)
Wrapper method for rfloordiv

Parameters
other : DataFrame or Panel
axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel

pandas.Panel.rmod

Panel.rmod(other, axis=0)
Wrapper method for rmod
pandas: powerful Python data analysis toolkit, Release 0.13.1

**Parameters**

- **other**: DataFrame or Panel
  
- **axis**: {items, major_axis, minor_axis}
  
**Axis to broadcast over**

**Returns**

Panel

**pandas.Panel.rpow**

Panel\(rpow\)(other, axis=0)

Wrapper method for rpow

**Parameters**

- **other**: DataFrame or Panel
  
- **axis**: {items, major_axis, minor_axis}
  
**Axis to broadcast over**

**Returns**

Panel

**pandas.Panel.lt**

Panel\(lt\)(other)

Wrapper for comparison method lt

**pandas.Panel.gt**

Panel\(gt\)(other)

Wrapper for comparison method gt

**pandas.Panel.le**

Panel\(le\)(other)

Wrapper for comparison method le

**pandas.Panel.ge**

Panel\(ge\)(other)

Wrapper for comparison method ge

**pandas.Panel.ne**

Panel\(ne\)(other)

Wrapper for comparison method ne

**pandas.Panel.eq**

Panel\(eq\)(other)

Wrapper for comparison method eq
28.5.7 Function application, GroupBy

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.apply(func, axis)</code></td>
<td>Applies function along input axis of the Panel</td>
</tr>
<tr>
<td><code>Panel.groupby(function, axis)</code></td>
<td>Group data on given axis, returning GroupBy object</td>
</tr>
</tbody>
</table>

**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.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

28.5.8 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.abs()</code></td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td><code>Panel.clip([lower, upper, out])</code></td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td><code>Panel.clip_lower(threshold)</code></td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td><code>Panel.clip_upper(threshold)</code></td>
<td>Return copy of input with values above given value truncated</td>
</tr>
<tr>
<td><code>Panel.count([axis])</code></td>
<td>Return number of observations over requested axis.</td>
</tr>
<tr>
<td><code>Panel.cummax([axis, dtype, out, skipna])</code></td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td><code>Panel.cummin([axis, dtype, out, skipna])</code></td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td><code>Panel.cumprod([axis, dtype, out, skipna])</code></td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td><code>Panel.cumsum([axis, dtype, out, skipna])</code></td>
<td>Return cumulative sum over requested axis.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.62 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.max()</code></td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td><code>Panel.mean()</code></td>
<td>Return the mean of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Panel.median()</code></td>
<td>Return the median of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Panel.min()</code></td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><code>Panel.pct_change()</code></td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td><code>Panel.prod()</code></td>
<td>Return the product of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Panel.skew()</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>Panel.sum()</code></td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Panel.std()</code></td>
<td>Return unbiased standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>Panel.var()</code></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

Parameters

- `abs`: type of caller

pandas.Panel.clip

Panel.clip(lower=None, upper=None, out=None)

Trim values at input threshold(s)

Parameters

- `lower`: float, default None
- `upper`: float, default None

Returns

- `clipped`: Series

pandas.Panel.clip_lower

Panel.clip_lower(threshold)

Return copy of the input with values below given value truncated

Returns

- `clipped`: same type as input

See Also:

- `clip`

pandas.Panel.clip_upper

Panel.clip_upper(threshold)

Return copy of input with values above given value truncated

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, 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, 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, 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, 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, **kwds)
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 : same type as caller

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, 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.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
- `level`: int, default None
- `numeric_only`: boolean, default None

**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
- `level`: int, default None
- `numeric_only`: boolean, default None

**Returns**

`sum`: DataFrame or Panel (if level specified)
pandas.Panel.std

Panel.std(axis=None, skipna=None, level=None, ddof=1, **kwargs)

Return unbiased standard deviation 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, 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

- **stdev**: DataFrame or Panel (if level specified)

pandas.Panel.var

Panel.var(axis=None, skipna=None, level=None, ddof=1, **kwargs)

Return unbiased variance 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, 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

- **variance**: DataFrame or Panel (if level specified)

28.5.9 Reindexing / Selection / Label manipulation

<table>
<thead>
<tr>
<th>Method</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.filter([items, like, regex, axis])</td>
<td>Restrict the info axis to set of items or wildcard.</td>
</tr>
<tr>
<td>Panel.first(offset)</td>
<td>Convenience method for subsetting initial 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.</td>
</tr>
<tr>
<td>Panel.reindex_axis(labels[, axis, method, ...])</td>
<td>Conform input object to new index with optional filling logic.</td>
</tr>
<tr>
<td>Panel.reindex_like(other[, method, copy, limit])</td>
<td>Return an object with matching indices to myself.</td>
</tr>
<tr>
<td>Panel.rename([items, major_axis, minor_axis])</td>
<td>Alter axes input function or functions.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.63 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.select(crit[, axis])</code></td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td><code>Panel.take(indices[, axis, convert, is_copy])</code></td>
<td>Analogous to ndarray.take</td>
</tr>
<tr>
<td><code>Panel.truncate([before, after, axis, copy])</code></td>
<td>Truncates a sorted NDFrame before and/or after some particular</td>
</tr>
</tbody>
</table>

### 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.drop

**Panel.drop**(labels, axis=0, level=None, inplace=False, **kwargs)

Return new object with labels in requested axis removed.

- **Parameters**
  - **labels**: single label or list-like
    - **axis**: int or axis name
    - **level**: int or name, default None
      - For MultiIndex
      - inplace: bool, default False
        - If True, do operation inplace and return None.

- **Returns**
  - dropped: type of caller

### 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
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 : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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

**takeable** : boolean, default False
treat the passed as positional values

**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=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**

**index** : array-like, optional
New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : {0,1,2,’items’,’major_axis’,’minor_axis’}

**method** : {'backfill', ‘bfill’, ‘pad’, ‘ffill’, None}, default None
Method to use for filling holes in reindexed object. pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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.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

28.5.10 Missing data handling

Panel.dropna([axis, how, inplace])  Drop 2D from panel, holding passed axis constant
Panel.fillna([value, method, axis, inplace, ...])  Fill NA/NaN values using the specified method

pandas.Panel.dropna

Panel.dropna (axis=0, how='any', inplace=False, **kwargs)
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=0, inplace=False, limit=None, downcast=None)

Fill NA/NaN values using the specified method

**Parameters**
- **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
  - value: scalar, dict, or Series
    - Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series will not be filled). This value cannot be a list.
  - axis: {0, 1}, default 0
    - 0: fill column-by-column
    - 1: fill row-by-row
  - 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
    - Maximum size gap to forward or backward fill
  - 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: same type as caller

**See Also:**
- reindex, asfreq

### 28.5.11 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.sort_index(axis, ascending)</td>
<td>Sort object by labels (along an axis)</td>
</tr>
<tr>
<td>Panel.swaplevel(i, j, [axis])</td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
<tr>
<td>Panel.transpose(*args, **kwargs)</td>
<td>Permute the dimensions of the Panel</td>
</tr>
<tr>
<td>Panel.swapaxes(axis1, axis2[, copy])</td>
<td>Interchange axes and swap values axes appropriately</td>
</tr>
<tr>
<td>Panel.conform(frame[, axis])</td>
<td>Conform input DataFrame to align with chosen axis pair.</td>
</tr>
</tbody>
</table>

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` : {'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
28.5.12 Combining / joining / merging

**pandas.Panel.join**

Panel.join(other[, how, 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, overwrite, ...])

Modify Panel in place using non-NA values from passed Panel

Parameters
- **other**: Panel, or object coercible to Panel
  - 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.

28.5.13 Time series-related

**Panel.asfreq**(freq[, method, how, normalize])

Convert all TimeSeries inside to specified frequency using DateOffset

Continued on next page...
pandas: powerful Python data analysis toolkit, Release 0.13.1

Table 28.67 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.shift(lags[, freq, axis])</td>
<td>Shift major or minor axis by specified number of leads/lags.</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, copy])</td>
<td>Convert TimeSeries to target time zone. If it is time zone naive, it</td>
</tr>
<tr>
<td>Panel.tz_localize(tz[, axis, copy, infer_dst])</td>
<td>Localize tz-naive TimeSeries to target time zone</td>
</tr>
</tbody>
</table>

**pandas.Panel.asfreq**

Panel.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
- **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(lags, freq=None, axis='major')

Shift major or minor axis by specified number of leads/lags. Drops periods right now compared with DataFrame.shift

**Parameters**
- **lags**: int
- **axis**: {'major', 'minor'}

**Returns**
- **shifted**: Panel

**pandas.Panel.resample**

Panel.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, offset=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.tz_convert
Panel.tz_convert(tz, axis=0, copy=True)
   Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.
Parameters tz : string or pytz.timezone object
   copy : boolean, default True
       Also make a copy of the underlying data

pandas.Panel.tz_localize
Panel.tz_localize(tz, axis=0, copy=True, infer_dst=False)
   Localize tz-naive TimeSeries to target time zone
Parameters tz : string or pytz.timezone object
   copy : boolean, default True
       Also make a copy of the underlying data
   infer_dst : boolean, default False
       Attempt to infer fall dst-transition times based on order

28.5.14 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel.from_dict</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>Panel.to_pickle</td>
<td>Pickle (serialize) object to input file path</td>
</tr>
<tr>
<td>Panel.to_excel</td>
<td>Write each DataFrame in Panel to a separate excel sheet</td>
</tr>
<tr>
<td>Panel.to_hdf</td>
<td>activate the HDFStore</td>
</tr>
<tr>
<td>Panel.to_json</td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td>Panel.to_sparse</td>
<td>Convert to SparsePanel</td>
</tr>
</tbody>
</table>

Continued on next page
pandas: powerful Python data analysis toolkit, Release 0.13.1

Table 28.68 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Panel.to_frame()</code></td>
<td>Transform wide format into long (stacked) format as DataFrame whose</td>
</tr>
<tr>
<td></td>
<td><code>filter_observations</code></td>
</tr>
<tr>
<td><code>Panel.to_clipboard()</code></td>
<td>Attempt to write text representation of object to the system clipboard</td>
</tr>
<tr>
<td></td>
<td><code>excel, sep</code></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’

**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`
    - `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’, ‘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.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

### 28.6 Panel4D

#### 28.6.1 Constructor

**Panel4D**(*data, labels, items, major_axis, ...*)  
Represents a 4 dimensional structured
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

- **at**: index(es) of the NDFrame
- **axes**: Internal property, property synonym for as_blocks()
- **dtypes**: Return the dtypes in this object
- **empty**: True if NDFrame is entirely empty [no items]
- **ftypes**: Return the ftypes (indication of sparse/dense and dtype)
- **iat**, **iloc**, **ix**, **loc**: Number of axes / array dimensions
- **shape**: tuple of axis dimensions
- **values**: Numpy representation of NDFrame

pandas.Panel4D.at

Panel4D.at

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

pandas.Panel4D.iloc

Panel4D.iloc

pandas.Panel4D.ix

Panel4D.ix

pandas.Panel4D.loc

Panel4D.loc

pandas.Panel4D.ndim

Panel4D.ndim
Number of axes / array dimensions

pandas.Panel4D.shape

Panel4D.shape
tuple of axis dimensions
### Panel4D.values

**Numpy representation of NDFrame**

<table>
<thead>
<tr>
<th>Method</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>Wrapper method for 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>apply(func[, axis])</td>
<td>Applies function along input axis of the Panel</td>
</tr>
<tr>
<td>as_blocks([columns])</td>
<td>Convert the frame to a dict of dtype -&gt; Constructor Types that each has</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</td>
</tr>
<tr>
<td>as_type(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.</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>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>bool()</td>
<td>Return the bool of a single element PandasObject</td>
</tr>
<tr>
<td>clip([lower, upper, out])</td>
<td>Trims values at input threshold(s)</td>
</tr>
<tr>
<td>clip_lower(threshold)</td>
<td>Return copy of the input with values above given value truncated</td>
</tr>
<tr>
<td>clip_upper(threshold)</td>
<td>Return copy of the input with values below given value truncated</td>
</tr>
<tr>
<td>compound([axis, skipna, level])</td>
<td>Return the compound percentage of the values for the requested axis</td>
</tr>
<tr>
<td>conform(frame[, axis])</td>
<td>Conform input DataFrame to align with chosen axis pair.</td>
</tr>
<tr>
<td>consolidate([inplace])</td>
<td>Compute NDFrame with “consolidated” internals (data of each dtype</td>
</tr>
<tr>
<td>convert_objects([convert_dates, ...])</td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td>copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>count([axis])</td>
<td>Return number of observations over requested axis.</td>
</tr>
<tr>
<td>cummax([axis, dtype, out, skipna])</td>
<td>Return cumulative max over requested axis.</td>
</tr>
<tr>
<td>cummin([axis, dtype, out, skipna])</td>
<td>Return cumulative min over requested axis.</td>
</tr>
<tr>
<td>cumprod([axis, dtype, out, skipna])</td>
<td>Return cumulative prod over requested axis.</td>
</tr>
<tr>
<td>cumsum([axis, dtype, out, skipna])</td>
<td>Return cumulative sum over requested axis.</td>
</tr>
<tr>
<td>div(other[, axis])</td>
<td>Wrapper method for truediv</td>
</tr>
<tr>
<td>divide(other[, axis])</td>
<td>Wrapper method for truediv</td>
</tr>
<tr>
<td>drop(*labels[, axis, level, inplace])</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>dropna(*args, **kwargs)</td>
<td>Wraps for comparison method eq</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determines if two NDFrame objects contain the same elements. NaNs in the</td>
</tr>
<tr>
<td>ffill([axis, inplace, limit, downcast])</td>
<td>Synonym for NDFrame.fillna(method= ‘ffill’ )</td>
</tr>
<tr>
<td>fillna([value, method, axis, inplace, ...])</td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td>filter(*args, **kwargs)</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>first(offset)</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>floorDiv(other[, axis])</td>
<td>Wrapper method for floordiv</td>
</tr>
<tr>
<td>fromDict(data[, intersect, orient, dtype])</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>from_dict(data[, intersect, orient, dtype])</td>
<td>Construct Panel from dict of DataFrame objects</td>
</tr>
<tr>
<td>ge(other)</td>
<td>Wrapper for comparison method ge</td>
</tr>
</tbody>
</table>
Table 28.71 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get(key[, default])</code></td>
<td>Get item from object for given key (DataFrame column, Panel slice, same as values (but handles sparseness conversions))</td>
</tr>
<tr>
<td><code>get_dtypes_counts()</code></td>
<td>Return the counts of dtypes in this object</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(*args)</code></td>
<td>Quickly retrieve single value at (item, major, minor) location</td>
</tr>
<tr>
<td><code>get_values()</code></td>
<td>same as values (but handles sparseness conversions)</td>
</tr>
<tr>
<td><code>groupby(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>gt(other)</code></td>
<td>Wrapper for comparison method gt</td>
</tr>
<tr>
<td><code>head([n])</code></td>
<td></td>
</tr>
<tr>
<td><code>interpolate(method, axis, limit, inplace, ...)</code></td>
<td>Interpolate values according to different methods.</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>iteritems()</code></td>
<td>Iterate over (label, values) on info axis</td>
</tr>
<tr>
<td><code>iterkv(*args, **kwargs)</code></td>
<td>iteritems alias used to get around 2to3. Deprecated</td>
</tr>
<tr>
<td><code>join(*args, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Get the ‘info axis’ (see Indexing for more)</td>
</tr>
<tr>
<td><code>kurt([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td><code>kurtosis([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased kurtosis over requested axis</td>
</tr>
<tr>
<td><code>last(offset)</code></td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td><code>le(other)</code></td>
<td>Wrapper for comparison method le</td>
</tr>
<tr>
<td><code>load(path)</code></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><code>lt(other)</code></td>
<td>Wrapper for comparison method lt</td>
</tr>
<tr>
<td><code>mad([axis, skipna, level])</code></td>
<td>Return the mean absolute deviation of the values for the requested axis</td>
</tr>
<tr>
<td><code>major_xs(key[, copy])</code></td>
<td>Return slice of panel along major axis</td>
</tr>
<tr>
<td><code>mask(cond)</code></td>
<td>Returns copy whose values are replaced with nan if the values are null</td>
</tr>
<tr>
<td><code>max([axis, skipna, level, numeric_only])</code></td>
<td>This method returns the maximum of the values in the object.</td>
</tr>
<tr>
<td><code>mean([axis, skipna, level, numeric_only])</code></td>
<td>Return the mean of the values for the requested axis</td>
</tr>
<tr>
<td><code>median([axis, skipna, level, numeric_only])</code></td>
<td>Return the median of the values for the requested axis</td>
</tr>
<tr>
<td><code>min([axis, skipna, level, numeric_only])</code></td>
<td>This method returns the minimum of the values in the object.</td>
</tr>
<tr>
<td><code>minor_xs(key[, copy])</code></td>
<td>Return slice of panel along minor axis</td>
</tr>
<tr>
<td><code>mod(other[, axis])</code></td>
<td>Wrapper method for mod</td>
</tr>
<tr>
<td><code>mul(other[, axis])</code></td>
<td>Wrapper method for mul</td>
</tr>
<tr>
<td><code>multiply(other[, axis])</code></td>
<td>Wrapper method for mul</td>
</tr>
<tr>
<td><code>ne(other)</code></td>
<td>Wrapper for comparison method ne</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Return a boolean same-sized object indicating if the values are not null</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>pop(item)</code></td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td><code>pow(other[, axis])</code></td>
<td>Wrapper method for 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>radd(other[, axis])</code></td>
<td>Wrapper method for radd</td>
</tr>
<tr>
<td><code>rdiv(other[, axis])</code></td>
<td>Wrapper method for rtruediv</td>
</tr>
<tr>
<td><code>reindex([items, major_axis, minor_axis])</code></td>
<td>Conform Panel to new index with optional filling logic, placing</td>
</tr>
<tr>
<td><code>reindex_axis(labels[, axis, method, level, ...])</code></td>
<td>Conform input object to new index with optional filling logic,</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit])</code></td>
<td>return an object with matching indicies to myself</td>
</tr>
<tr>
<td><code>rename([items, major_axis, minor_axis])</code></td>
<td>Alter axes input function or functions.</td>
</tr>
<tr>
<td><code>rename_axis(mapper[, axis, copy, inplace])</code></td>
<td>Alter index and / or columns using input function or functions.</td>
</tr>
<tr>
<td><code>replace(to_replace, value, inplace)</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>rfloordiv(other[, axis])</code></td>
<td>Wrapper method for rfloordiv</td>
</tr>
<tr>
<td><code>rmod(other[, axis])</code></td>
<td>Wrapper method for rmod</td>
</tr>
<tr>
<td><code>rmul(other[, axis])</code></td>
<td>Wrapper method for rmul</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.71 – continued from previous page

- `rpow(other[, axis])`: Wrapper method for rpow
- `rsub(other[, axis])`: Wrapper method for rsub
- `rtruediv(other[, axis])`: Wrapper method for rtruediv
- `save(path)`: Deprecated.
- `select(crit[, axis])`: Return data corresponding to axis labels matching criteria
- `set_value(*args)`: Quickly set single value at (item, major, minor) location
- `shift(*args, **kwargs)`: Return unbiased skew over requested axis
- `sort_index([axis, ascending])`: Return object by labels (along an axis)
- `squeeze()`: squeeze length 1 dimensions
- `std([axis, skipna, level, ddof])`: Return unbiased standard deviation over requested axis
- `subtract(other[, axis])`: Wrapper method for sub
- `sum([axis, skipna, level, numeric_only])`: Return the sum of the values for the requested axis
- `swapaxes(axis1, axis2[, copy])`: Interchange axes and swap values axes appropriately
- `swaplevel(i, j[, axis])`: Swap levels i and j in a MultiIndex on a particular axis
- `tail([n])`: Analogous to ndarray.take
- `take(indices[, axis, convert, is_copy])`: Return dense representation of NDFrame (as opposed to sparse)
- `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_pickle(path)`: Pickle (serialize) object to input file path
- `transpose(*args, **kwargs)`: Permute the dimensions of the Panel
- `truncate([before, after, axis, copy])`: Truncates a sorted NDFrame before and/or after some particular
- `tz_convert(tz[, axis, copy])`: Convert TimeSeries to target time zone. If it is time zone naive, it
- `tz_localize(tz[, axis, copy, infer_dst])`: Localize tz-naive TimeSeries to target time zone
- `update(other[, join, overwrite, ...])`: Modify Panel in place using non-NA values from passed
- `var([axis, skipna, level, ddof])`: Return unbiased variance over requested axis
- `where(cond[, other, inplace, axis, level, ...])`: Return an object of same shape as self and whose corresponding
- `xs(key[, axis, copy])`: Return slice of panel along selected axis

**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)`

Wrapper method for add
Parameters  **other** : Panel or Panel4D

    axis : {labels, items, major_axis, minor_axis}

    Axis to broadcast over

Returns  Panel4D

**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)

Concatenate 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 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 name

    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.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 (columns=\text{None})

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

are presented in sorted order unless a specific list of columns is provided.

**NOTE:** the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

**Parameters**

- **columns** : array-like
  
  Specific column order

**Returns**

values : a list of Object

**pandas.Panel4D.as_matrix**

Panel4D.as_matrix()

**pandas.Panel4D.asfreq**

Panel4D.asfreq (freq, method=\text{None}, how=\text{None}, normalize=\text{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', \text{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)

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

**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

**Returns**

- **values_at_time**: type of caller

---

**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=0, 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)
```

Trim values at input threshold(s)

Parameters  
- **lower**: float, default None
- **upper**: float, default None

Returns **clipped**: Series

**pandas.Panel4D.clip_lower**

```
Panel4D.clip_lower(threshold)
```

Return copy of the input with values below given value truncated

Returns **clipped**: same type as input

See Also:
- clip

**pandas.Panel4D.clip_upper**

```
Panel4D.clip_upper(threshold)
```

Return copy of input with values above given value truncated

Returns **clipped**: same type as input

See Also:
- clip

**pandas.Panel4D.compound**

```
Panel4D.compound(axis=None, skipna=None, level=None, **kwargs)
```

Return the compound percentage 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, 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** : if True, attempt to soft convert dates, if ‘coerce’, force conversion (and non-convertibles get NaT)
  **convert_numeric** : if True attempt to coerce to numbers (including strings), non-convertibles get NaN
  **convert_timedeltas** : if True, attempt to soft convert timedeltas, if ‘coerce’, force conversion (and non-convertibles get NaT)
  **copy** : Boolean, if True, return copy, default is True

**Returns**  converted : asm as input object

**pandas.Panel4D.copy**

Panel4D.copy(deep=True)
Make a copy of this object

**Parameters**  **deep** : boolean, 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.div

Panel4D.div(other, axis=0)
Wrapper method for truediv

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

pandas.Panel4D.divide

Panel4D.divide(other, axis=0)
Wrapper method for truediv

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

pandas.Panel4D.drop

Panel4D.drop(labels, axis=0, level=None, inplace=False, **kwargs)
Return new object with labels in requested axis removed

Parameters
labels : single label or list-like
axis : int or axis name
level : int or name, default None
For MultiIndex
inplace : bool, default False
If True, do operation inplace and return None.

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.fill

Panel4D.fill(axis=0, inplace=False, limit=None, downcast=None)
Synonym for NDFrame.fillna(method='ffill')

pandas.Panel4D.fillna

Panel4D.fillna(value=None, method=None, axis=0, inplace=False, limit=None, downcast=None)
Fill NA/NaN values using the specified method

Parameters  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

value : scalar, dict, or Series
Value to use to fill holes (e.g. 0), alternately a dict/Series of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series will not be filled). This value cannot be a list.

axis : {0, 1}, default 0
• 0: fill column-by-column
• 1: fill row-by-row

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
Maximum size gap to forward or backward fill

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 : same type as caller

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)
Wrapper method for floordiv

Parameters other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns 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’

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’

Returns:
- Panel
```

pandas.Panel4D.ge

```
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)

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)

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='infer', **kwargs)
Interpolate values according to different methods.

Parameters method: {'linear', 'time', 'values', 'index' '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’: use the actual numerical values of the index
is passed to scipy.interpolate.interp1d with the order given both ‘polyno-
mial’ 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 ‘infer’
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 1 2 3
dtype: float64

pandas.Panel4D.isnull

Panel4D.isnull()
Return a boolean same-sized object indicating if the values are null

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)
iteritems 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 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, 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 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, 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: powerful Python data analysis toolkit, Release 0.13.1

**pandas.Panel4D.le**

Panel4D.le(other)
Wrapper for comparison method le

**pandas.Panel4D.load**

Panel4D.load(path)
Deprecated. Use read_pickle instead.

**pandas.Panel4D.lt**

Panel4D.lt(other)
Wrapper for comparison method lt

**pandas.Panel4D.mad**

Panel4D.mad(axis=None, skipna=None, level=None, **kwargs)
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, 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**

Panel4D.major_xs(key, copy=True)
Return slice of panel along major axis

**Parameters**

- **key**: object
  Major axis label
- **copy**: boolean, default True
  Copy data

**Returns**

- **y**: DataFrame
  index -> minor axis, columns -> items
**pandas.Panel4D.mask**

`Panel4D.mask(cond)`

Returns copy whose values are replaced with `nan` if the inverted condition is `True`

**Parameters**
- `cond`: boolean NDFrame or array

**Returns**
- `wh`: same as input

**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`
- `level`: int, default `None`
- `numeric_only`: boolean, default `None`

**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`
- `level`: int, default `None`
- `numeric_only`: boolean, default `None`

**Returns**
- `mean`: Panel or Panel4D (if level specified)
**pandas.Panel4D.median**

Panel4D.

median

<table>
<thead>
<tr>
<th>Parameters</th>
<th>axis</th>
<th>{labels (0), items (1), major_axis (2), minor_axis (3)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipna</td>
<td>boolean, default True</td>
<td>Exclude NA/null values. If an entire row/column is NA, the result will be NA</td>
</tr>
<tr>
<td>level</td>
<td>int, default None</td>
<td>If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Panel</td>
</tr>
<tr>
<td>numeric_only</td>
<td>boolean, default None</td>
<td>Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data</td>
</tr>
</tbody>
</table>

Returns median: Panel or Panel4D (if level specified)

**pandas.Panel4D.min**

Panel4D.

min

<table>
<thead>
<tr>
<th>Parameters</th>
<th>axis</th>
<th>{labels (0), items (1), major_axis (2), minor_axis (3)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipna</td>
<td>boolean, default True</td>
<td>Exclude NA/null values. If an entire row/column is NA, the result will be NA</td>
</tr>
<tr>
<td>level</td>
<td>int, default None</td>
<td>If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Panel</td>
</tr>
<tr>
<td>numeric_only</td>
<td>boolean, default None</td>
<td>Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data</td>
</tr>
</tbody>
</table>

Returns min: Panel or Panel4D (if level specified)

**pandas.Panel4D.minor_xs**

Panel4D.

minor_xs

<table>
<thead>
<tr>
<th>Parameters</th>
<th>key</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>boolean, default True</td>
<td>Copy data</td>
</tr>
</tbody>
</table>

Returns y: DataFrame

index -> major axis, columns -> items
pandas.Panel4D.mod

Panel4D.mod(other, axis=0)
Wrapper method for mod

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel4D

pandas.Panel4D.mul

Panel4D.mul(other, axis=0)
Wrapper method for mul

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel4D

pandas.Panel4D.multiply

Panel4D.multiply(other, axis=0)
Wrapper method for mul

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel4D

pandas.Panel4D.ne

Panel4D.ne(other)
Wrapper for comparison method ne

pandas.Panel4D.notnull

Panel4D.notnull()
Return a boolean same-sized object indicating if the values are not null

pandas.Panel4D.pct_change

Panel4D.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwds)
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**: same type as caller

**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)

Wrapper method for pow

**Parameters**  
**other**: Panel or Panel4D

**axis**: {labels, items, major_axis, minor_axis}

**Axis to broadcast over**

**Returns**  
Panel4D

**pandas.Panel4D.prod**

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, 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.product

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, 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)
Wrapper method for radd

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

pandas.Panel4D.rdiv

Panel4D.rdiv (other, axis=0)
Wrapper method for rtruediv

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Returns
Panel4D

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** : \{‘backfill’, ‘bfill’, ‘pad’, ‘fill’, None\}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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

**takeable** : boolean, default False

treat the passed as positional values

**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**  
*index*: array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : \{0,1,2,‘items’,‘major_axis’,‘minor_axis’\}

**method** : \{‘backfill’, ‘bfill’, ‘pad’, ‘fill’, None\}, default None

Method to use for filling holes in reindexed object. pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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.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
  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, 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)

Wrapper method for rfloordiv

**Parameters**
- **other**: Panel or Panel4D
- **axis**: {labels, items, major_axis, minor_axis}

**Returns**
- Panel4D

**pandas.Panel4D.rmod**

Panel4D.rmod(other, axis=0)

Wrapper method for rmod
Parameters `other` : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D

**pandas.Panel4D.rmul**

`Panel4D.rmul(other, axis=0)`

Wrapper method for rmul

Parameters `other` : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D

**pandas.Panel4D.rpow**

`Panel4D.rpow(other, axis=0)`

Wrapper method for rpow

Parameters `other` : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D

**pandas.Panel4D.rsub**

`Panel4D.rsub(other, axis=0)`

Wrapper method for rsub

Parameters `other` : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D

**pandas.Panel4D.rtruediv**

`Panel4D.rtruediv(other, axis=0)`

Wrapper method for rtruediv

Parameters `other` : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel4D
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.set_value

Panel4D.set_value(*args)
    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
    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, 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.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.Panel4D.std**

Panel4D.std(axis=None, skipna=None, level=None, ddof=1, **kwargs)  
Return unbiased standard deviation 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, 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**  
`stdev` : Panel or Panel4D (if level specified)

**pandas.Panel4D.sub**

Panel4D.sub(other, axis=0)  
Wrapper method for sub

**Parameters**  
other : Panel or Panel4D

axis : {labels, items, major_axis, minor_axis}  
Axis to broadcast over
Returns Panel4D

pandas.Panel4D.subtract

Panel4D.subtract (other, axis=0)

Wrapper method for sub

Parameters

other : Panel or Panel4D
axis : [labels, items, major_axis, minor_axis]

Axis to broadcast over

Returns Panel4D

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, 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

pandas.Panel4D.toLong

Panel4D.toLong(*args, **kwargs)

pandas.Panel4D.to_clipboard

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
                   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.Panel4D.to_dense

Panel4D.to_dense()
   Return dense representation of NDFrame (as opposed to sparse)
pandas.Panel4D.to_excel

Panel4D.to_excel(*args, **kwargs)

pandas.Panel4D.to_frame

Panel4D.to_frame(*args, **kwargs)

pandas.Panel4D.to_hdf

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'

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', '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 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.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.transpose

Panel4D.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.Panel4D.truediv

Panel4D.truediv(other, axis=0)
Wrapper method for truediv

Parameters
other : Panel or Panel4D
axis : {labels, items, major_axis, minor_axis}

Axis to broadcast over

Returns
Panel4D

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,
    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, copy=True)
Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

Parameters
tz : string or pytz.timezone object
    copy : boolean, default True
    Also make a copy of the underlying data

pandas.Panel4D.tz_localize

Panel4D.tz_localize(tz, axis=0, copy=True, infer_dst=False)
Localize tz-naive TimeSeries to target time zone

Parameters
tz : string or pytz.timezone object
    copy : boolean, default True
Also make a copy of the underlying data

\texttt{infer\_dst} : boolean, default False

Attempt to infer fall dst-transition times based on order

\texttt{pandas.Panel4D.update}

\texttt{Panel4D.update} \texttt{(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

\textbf{Parameters}
\begin{itemize}
  \item \texttt{other} : Panel, or object coercible to Panel
  \item \texttt{join} : How to join individual DataFrames
    \{ ‘left’, ‘right’, ‘outer’, ‘inner’ \}, default ‘left’
  \item \texttt{overwrite} : boolean, default True
    If True then overwrite values for common keys in the calling panel
  \item \texttt{filter\_func} : callable\(1\text{d-array}\) -> \(1\text{d-array boolean}\), default None
    Can choose to replace values other than NA. Return True for values that should be updated
  \item \texttt{raise\_conflict} : bool
    If True, will raise an error if a DataFrame and other both contain data in the same place.
\end{itemize}

\texttt{pandas.Panel4D.var}

\texttt{Panel4D.var} \texttt{(axis=None, skipna=None, level=None, ddof=1, **kwargs)}

Return unbiased variance over requested axis Normalized by N-1

\textbf{Parameters}
\begin{itemize}
  \item \texttt{axis} : \{ labels (0), items (1), major\_axis (2), minor\_axis (3) \}
  \item \texttt{skipna} : boolean, default True
    Exclude NA/null values. If an entire row/column is NA, the result will be NA
  \item \texttt{level} : int, default None
    If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Panel
  \item \texttt{numeric\_only} : boolean, default None
    Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data
\end{itemize}

\textbf{Returns} \texttt{variance} : Panel or Panel4D (if level specified)

\texttt{pandas.Panel4D.where}

\texttt{Panel4D.where} \texttt{(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=True)**

Return slice of panel along selected axis

Parameters `key` : object

Label

`axis` : {‘items’, ‘major’, ‘minor’}, default 1/major
`copy` : boolean, default True

Copy data

Returns `y` : ndim(self)-1

### 28.6.2 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>Panel4D.values</th>
<th>Numpy representation of NDFrame</th>
</tr>
</thead>
<tbody>
<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>
<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)</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

**pandas.Panel4D.axes**

Panel4D.axes
index(es) of the NDFrame

**pandas.Panel4D.ndim**

Panel4D.ndim
Number of axes / array dimensions

**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

### 28.6.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel4D.astype(dtype[, copy, raise_on_error])</td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td>Panel4D.copy([deep])</td>
<td>Make a copy of this object</td>
</tr>
<tr>
<td>Panel4D.isnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
<tr>
<td>Panel4D.notnull()</td>
<td>Return a boolean same-sized object indicating if the values are null</td>
</tr>
</tbody>
</table>
**pandas.Panel4D.astype**

Panel4D.astype (dtype, copy=True, raise_on_error=True)

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

**Returns**
- **casted** : type of caller

**pandas.Panel4D.copy**

Panel4D.copy (deep=True)

Make a copy of this object

**Parameters**
- **deep** : boolean, default True
  - Make a deep copy, i.e. also copy data

**Returns**
- **copy** : type of caller

**pandas.Panel4D.isnull**

Panel4D.isnull ()

Return a boolean same-sized object indicating if the values are null

**pandas.Panel4D.notnull**

Panel4D.notnull ()

Return a boolean same-sized object indicating if the values are not null

### 28.7 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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Immutable ndarray implementing an ordered, sliceable set.</th>
</tr>
</thead>
</table>

### 28.7.1 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)
  - Make a copy of input ndarray
- **dtype** : NumPy dtype (default: object)
- **copy** : bool
  - Make a copy of input ndarray
- **name** : object
Name to be stored in the index

Notes

An Index instance can **only** contain hashable objects

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td>base</td>
<td>Base object if memory is from some other object.</td>
</tr>
<tr>
<td>ctypes</td>
<td>An object to simplify the interaction of the array with the ctypes module.</td>
</tr>
<tr>
<td>data</td>
<td>Python buffer object pointing to the start of the array’s data.</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>flat</td>
<td>A 1-D iterator over the array.</td>
</tr>
<tr>
<td>imag</td>
<td>The imaginary part of the array.</td>
</tr>
<tr>
<td>is_monotonic</td>
<td></td>
</tr>
<tr>
<td>itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>names</td>
<td></td>
</tr>
<tr>
<td>nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>nlevels</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>The real part of the array.</td>
</tr>
<tr>
<td>shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
<tr>
<td>values</td>
<td></td>
</tr>
</tbody>
</table>

**pandas.Index.T**

Index.T

Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

```python
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1., 2.],
       [ 3., 4.]])
>>> x.T
array([[ 1., 3.],
       [ 2., 4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1., 2., 3., 4.])
>>> x.T
array([ 1., 2., 3., 4.])
```
pandas.Index.base

Index.base
Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
>>> y = x[2:]
>>> y.base is x
True
```

pandas.Index.ctypes

Index.ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters  None

Returns  c : Python object

Possessing attributes data, shape, strides, etc.

See Also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

• data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

• shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

• strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array.
This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

• data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

• shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

• strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
<ctypes.c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<ctypes.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<ctypes.c_longlong_Array_2 object at 0x01F01300>
```

pandas.Index.data

Index.data
Python buffer object pointing to the start of the array’s data.

pandas.Index.flags

Index.flags
pandas.Index.flat

Index.flat
A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

flatten Return a copy of the array collapsed into one dimension.

Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

pandas.Index.imag

Index.imag
The imaginary part of the array.

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```
pandas.Index.is_monotonic

Index.is_monotonic

pandas.Index.itemsize

Index.itemsize
Length of one array element in bytes.

Examples

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

pandas.Index.names

Index.names

pandas.Index.nbytes

Index.nbytes
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

pandas.Index.ndim

Index.ndim
Number of array dimensions.

Examples
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

pandas.Index.nlevels

Index.nlevels

pandas.Index.real

Index.real
The real part of the array.

See Also:

numpy.real equivalent function

Examples

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')

pandas.Index.shape

Index.shape
Tuple of array dimensions.

Notes
May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
pandas: powerful Python data analysis toolkit, Release 0.13.1

```python
[0., 0., 0., 0., 0., 0., 0., 0.])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

### pandas.Index.size

**Index.size**

Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

### pandas.Index.strides

**Index.strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array `a` is:

```python
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

**See Also:**

`numpy.lib.stride_tricks.as_strided`

**Notes**

Imagine an array of 32-bit integers (each 4 bytes):

```python
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be (20, 4).

**Examples**
```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))

array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])

>>> y.strides
(48, 16, 4)

>>> y[1,1,1]
17

>>> offset = sum(y.strides * np.array((1,1,1)))

>>> offset / y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)

>>> x.strides
(32, 4, 224, 1344)

>>> i = np.array([3,5,2,2])

>>> offset = sum(i * x.strides)

>>> x[3,5,2,2]
813

>>> offset / x.itemsize
813
```

The `pandas.Index.values` class contains the following methods:

### Methods

- **all([axis, out])**
  - Returns True if all elements evaluate to True.

- **any([axis, out])**
  - Returns True if any of the elements of a evaluate to True.

- **append(other)**
  - Append a collection of Index options together.

- **argmax([axis, out])**
  - Return indices of the maximum values along the given axis.

- **argmin([axis, out])**
  - Return indices of the minimum values along the given axis of a.

- **argsort(*args, **kwargs)**
  - See docstring for ndarray.argsort

- **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)**
  - Swap the bytes of the array elements.

- **choose(choices[, out, mode])**
  - Use an index array to construct a new array from a set of choices.

- **clip(a_min, a_max[, out])**
  - Return an array whose values are limited to [a_min, a_max].

- **compress(condition[, axis, out])**
  - Return selected slices of this array along given axis.

---

28.7. Index 1013
# pandas: powerful Python data analysis toolkit, Release 0.13.1

## Table 28.76 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conj()</code></td>
<td>Complex-conjugate all elements.</td>
</tr>
<tr>
<td><code>conjugate()</code></td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td><code>copy(names, name, dtype, deep)</code></td>
<td>Make a copy of this object.</td>
</tr>
<tr>
<td><code>cumprod(axis, dtype, out)</code></td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td><code>cumsum(axis, dtype, out)</code></td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td><code>delete(loc)</code></td>
<td>Make new Index with passed location deleted</td>
</tr>
<tr>
<td><code>diagonal(offset, axis1, axis2)</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>diff(other)</code></td>
<td>Compute sorted set difference of two Index objects</td>
</tr>
<tr>
<td><code>dot(b[, out])</code></td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td><code>drop(labels)</code></td>
<td>Make new Index with passed list of labels deleted</td>
</tr>
<tr>
<td><code>dump(file)</code></td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td><code>dumpstr()</code></td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td><code>equals(other)</code></td>
<td>Determines if two Index objects contain the same elements.</td>
</tr>
<tr>
<td><code>fill(*args, **kwargs)</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>flatten(order)</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>format(name, formatter)</code></td>
<td>Render a string representation of the Index</td>
</tr>
<tr>
<td><code>get_duplicates()</code></td>
<td>Compute join_index and indexers to conform data</td>
</tr>
<tr>
<td><code>get_indexer(target[, method, limit])</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>get_indexer_non_unique(target, **kwargs)</code></td>
<td>return an indexer suitable for taking from a non unique index</td>
</tr>
<tr>
<td><code>get_level_values(level)</code></td>
<td>Return vector of label values for requested level, equal to the length</td>
</tr>
<tr>
<td><code>get_loc(key)</code></td>
<td>Get integer location for requested label</td>
</tr>
<tr>
<td><code>get_values()</code></td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>groupby(to_groupby)</code></td>
<td>Create a new Index from another Index.</td>
</tr>
<tr>
<td><code>holds_integer()</code></td>
<td>Make new Index inserting new item at location</td>
</tr>
<tr>
<td><code>identical(other)</code></td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td><code>insert(loc, item)</code></td>
<td>Form the intersection of two Index objects. Sortedness of the result is</td>
</tr>
<tr>
<td><code>intersection(other)</code></td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td><code>is_(_other)</code></td>
<td>Will not function because object is immutable.</td>
</tr>
<tr>
<td><code>is_floating()</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>is_integer()</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>is_lexsorted_for_tuple(tup)</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>is_numeric()</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>is_type_compatible(typ)</code></td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td><code>isin(values)</code></td>
<td>Compute boolean array of whether each index value is found in the</td>
</tr>
<tr>
<td><code>itemset(*args, **kwars)</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>join(other[, how, level, return_indexers])</code></td>
<td>Internal API method. Compute join_index and indexers to conform data</td>
</tr>
<tr>
<td><code>map(mapper)</code></td>
<td>Create a new Index from another Index.</td>
</tr>
<tr>
<td><code>max(axis, out)</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>mean(axis, dtype, out)</code></td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td><code>min(axis, out)</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>newbyteorder([new_order])</code></td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>order([return_indexer, ascending])</code></td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td><code>prod(axis, out)</code></td>
<td>Return the product of the array elements over the given axis</td>
</tr>
<tr>
<td><code>put(*args, **kwars)</code></td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.76 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ravel(order)</td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td>reindex(target[, method, level, limit, ...])</td>
<td>For Index, simply returns the new index and the results of</td>
</tr>
<tr>
<td>rename(name[, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>repeat(repeats[, axis])</td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td>reshape(shape[, order])</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>resize(new_shape[, refcheck])</td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td>round([decimals, out])</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>searchsorted(v[, side, sorter])</td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td>set_names(names[, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>set_value(arr, key, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>setfield(val, dtype[, offset])</td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td>setflags([write, align, uic])</td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td>shift([periods, freq])</td>
<td>Shift Index containing datetime objects by input number of periods and</td>
</tr>
<tr>
<td>slice_indexer([start, end, step])</td>
<td>For an ordered Index, compute the slice indexer for input labels and</td>
</tr>
<tr>
<td>slice_locs([start, end])</td>
<td>For an ordered Index, compute the slice locations for input labels</td>
</tr>
<tr>
<td>sort(*args, **kwargs)</td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td>squeeze([axis])</td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td>std([axis, dtype, out, ddof])</td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td>sum([axis, dtype, out])</td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td>summary([name])</td>
<td>Summary view of the array with axes interchanged.</td>
</tr>
<tr>
<td>swapaxes(axis1, axis2)</td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td>take(indexer[, axis])</td>
<td>Analogous to ndarray.take</td>
</tr>
<tr>
<td>to_datetime([dayfirst])</td>
<td>For an Index containing strings or datetime.datetime objects, attempt</td>
</tr>
<tr>
<td>to_native_types([slizer])</td>
<td>slice and dice then format</td>
</tr>
<tr>
<td>to_series()</td>
<td>Return a series with both index and values equal to the index keys</td>
</tr>
<tr>
<td>tofile(fid[, sep, format])</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>tolist()</td>
<td>Overridden version of ndarray.tolist</td>
</tr>
<tr>
<td>tostring([order])</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>transpose(*axes)</td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td>union(other)</td>
<td>Form the union of two Index objects and sorts if possible</td>
</tr>
<tr>
<td>unique()</td>
<td>Return array of unique values in the Index. Significantly faster than</td>
</tr>
<tr>
<td>var([axis, dtype, out, ddof])</td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td>view(*args, **kwargs)</td>
<td>View of the array with specified axes transposed.</td>
</tr>
</tbody>
</table>

pandas.Index.all

Index . all (axis=None, out=None)
Returns True if all elements evaluate to True.
Refer to numpy.all for full documentation.

See Also:

numpy . all equivalent function

pandas.Index.any

Index . any (axis=None, out=None)
Returns True if any of the elements of a evaluate to True.
Refer to numpy.any for full documentation.
See Also:

`numpy.any` equivalent function

**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, out=None)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also:

`numpy.argmax` equivalent function

**pandas.Index.argmin**

`Index.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of a.

Refer to `numpy.argmin` for detailed documentation.

See Also:

`numpy.argmin` equivalent function

**pandas.Index.argsort**

`Index.argsort(*args, **kwargs)`

See docstring for `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.

**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.byteswap

```
Index.byteswap(inplace)
```

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

**Parameters**
**inplace**: bool, optional

- If True, swap bytes in-place, default is False.

**Returns**
**out**: ndarray

The byteswapped array. If *inplace* is True, this is a view to self.

**Examples**

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

pandas.Index.choose

```
Index.choose(choices, out=None, mode='raise')
```

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

**See Also:**

- *numpy.choose* equivalent function

pandas.Index.clip

```
Index.clip(a_min, a_max, out=None)
```

Return an array whose values are limited to \([a_{min}, a_{max}]\).

Refer to *numpy.clip* for full documentation.

**See Also:**
**numpy.clip** equivalent function

**pandas.Index.compress**

Index.compress(condition, axis=None, out=None)

Return selected slices of this array along given axis.

Refer to numpy.compress for full documentation.

See Also:

**numpy.compress** equivalent function

**pandas.Index.conj**

Index.conj()

Complex-conjugate all elements.

Refer to numpy.conjugate for full documentation.

See Also:

**numpy.conjugate** equivalent function

**pandas.Index.conjugate**

Index.conjugate()

Return the complex conjugate, element-wise.

Refer to numpy.conjugate for full documentation.

See Also:

**numpy.conjugate** equivalent function

**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.cumprod

`Index.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

**See Also:**

- `numpy.cumprod` equivalent function

### pandas.Index.cumsum

`Index.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

**See Also:**

- `numpy.cumsum` equivalent function

### pandas.Index.delete

`Index.delete(loc)`

Make new Index with passed location deleted

**Returns** `new_index : Index`

### pandas.Index.diagonal

`Index.diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals.

Refer to `numpy.diagonal()` for full documentation.

**See Also:**

- `numpy.diagonal` equivalent function

### pandas.Index.diff

`Index.diff(other)`

Compute sorted set difference of two Index objects

**Returns** `diff : Index`

**Notes**

One can do either of these and achieve the same result

```python
>>> index - index2
>>> index.diff(index2)
```
**pandas.Index.dot**

`Index.dot(b, out=None)`  
Dot product of two arrays.  
Refer to `numpy.dot` for full documentation.  
**See Also:**  
`numpy.dot` equivalent function

**Examples**

```python  
>>> a = np.eye(2)  
>>> b = np.ones((2, 2)) * 2  
>>> a.dot(b)  
array([[ 2.,  2.],  
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```python  
>>> a.dot(b).dot(b)  
array([[ 8.,  8.],  
       [ 8.,  8.]])
```

**pandas.Index.drop**

`Index.drop(labels)`  
Make new Index with passed list of labels deleted  
**Parameters**  
`labels` : array-like  
**Returns**  
`dropped` : Index

**pandas.Index.dump**

`Index.dump(file)`  
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.  
**Parameters**  
`file` : str  
A string naming the dump file.

**pandas.Index.dumps**

`Index.dumps()`  
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.  
**Parameters**  
`None`

**pandas.Index.equals**

`Index.equals(other) `  
Determines if two Index objects contain the same elements.
pandas.Index.fill

Index.fill(*args, **kwargs)
This method will not function because object is immutable.

pandas.Index.flatten

Index.flatten(order='C')
Return a copy of the array collapsed into one dimension.

Parameters order : {'C', 'F', 'A'}, optional
Whether to flatten in C (row-major), Fortran (column-major) order, or preserve
the C/Fortran ordering from a. The default is 'C'.

Returns y : ndarray
A copy of the input array, flattened to one dimension.

See Also:

ravel Return a flattened array.
flat A 1-D flat iterator over the array.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])

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. The mask determines whether labels are
found or not in the current index.

Parameters target : Index
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 : ndarray

Notes

This is a low-level method and probably should be used at your own risk

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, **kwargs)
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)
Get integer location for requested label

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.get_values

Index.get_values()
pandas.Index.getfield

Index.getfield(dtype, offset=0)
Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the
given type and the offset into the current array in bytes. The offset needs to be such that the view dtype
fits in the array dtype: for example an array of dtype complex128 has 16-byte elements. If taking a view
with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters
dtype : str or dtype
    The data type of the view. The dtype size of the view can not be larger than that
    of the array itself.

offset : int
    Number of bytes to skip before beginning the element view.

Examples

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```

pandas.Index.groupby

Index.groupby(to_groupby)

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
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_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_type_compatible

Index.is_type_compatible(typ)
pandas.Index.isin

Index.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.Index.item

Index.item(*args)
Copy an element of an array to a standard Python scalar and return it.

Parameters  *args : Arguments (variable number and type)
  • none: in this case, the method only works for arrays with one element (a.size == 1),
    which element is copied into a standard Python scalar object and returned.
  • int_type: this argument is interpreted as a flat index into the array, specifying which
    element to copy and return.
  • tuple of int_types: functions as does a single int_type argument, except that the argu-
    ment is interpreted as an nd-index into the array.

Returns  z : Standard Python scalar object
  A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is
no available Python scalar that would not lose information. Void arrays return a buffer object for item(),
unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This
can be useful for speeding up access to elements of the array and doing arithmetic on elements of the
array using Python’s optimized math.

Examples

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
pandas.Index.itemset

Index.itemset(*args, **kwargs)
   This method will not function because object is immutable.

pandas.Index.join

Index.join(other, how='left', level=None, return_indexers=False)
   Internal API method. Compute join_index and indexers to conform data structures to the new index.
   Parameters
   other : Index
   how : {'left', 'right', 'inner', 'outer'}
   level :
   return_indexers : boolean, default False
   Returns
   join_index, (left_indexer, right_indexer)

pandas.Index.map

Index.map(mapper)

pandas.Index.max

Index.max(axis=None, out=None)
   Return the maximum along a given axis.
   Refer to numpy.amax for full documentation.
   See Also:
   
   numpy.amax equivalent function

pandas.Index.mean

Index.mean(axis=None, dtype=None, out=None)
   Returns the average of the array elements along given axis.
   Refer to numpy.mean for full documentation.
   See Also:
   
   numpy.mean equivalent function

pandas.Index.min

Index.min(axis=None, out=None)
   Return the minimum along a given axis.
   Refer to numpy.amin for full documentation.
   See Also:
   
   numpy.amin equivalent function
pandas.Index.newbyteorder

Index.newbyteorder(new_order=’S’)

Return the array with the same data viewed with a different byte order.

Equivalent to:

arr.view(arr.dtype.newbyteorder(new_order))

Changes are also made in all fields and sub-arrays of the array data type.

Parameters  

new_order : string, optional

Byte order to force; a value from the byte order specifications above. new_order
codes can be any of:

* ’S’ - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

The default value (‘S’) results in swapping the current byte order. The code does
a case-insensitive check on the first letter of new_order for the alternatives above.
For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

Returns  

new_arr : array

New array object with the dtype reflecting given change to the byte order.

pandas.Index.nonzero

Index.nonzero()

Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

See Also:

numpy.nonzero equivalent function

pandas.Index.order

Index.order(return_indexer=False, ascending=True)

Return sorted copy of Index

pandas.Index.prod

Index.prod(axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis

Refer to numpy.prod for full documentation.

See Also:

numpy.prod equivalent function
**pandas.Index.ptp**

Index.ptp(*axis*, *out*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to numpy.ptp for full documentation.

See Also:

numpy.ptp equivalent function

**pandas.Index.put**

Index.put(*args*, **kwargs)

This method will not function because object is immutable.

**pandas.Index.ravel**

Index.ravel(*order*)

Return a flattened array.

Refer to numpy.ravel for full documentation.

See Also:

numpy.ravel equivalent function

ndarray.flat a flat iterator on the array.

**pandas.Index.reindex**

Index.reindex(*target*, *method*, *level*, *limit*, *copy_if_needed*, *takeable*)

For Index, simply returns the new index and the results of get_indexer. Provided here to enable an interface that is amenable for subclasses of Index whose internals are different (like MultiIndex)

Returns (new_index, indexer, mask): tuple

**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(repeats, axis=None)
Repeat elements of an array.
Refer to numpy.repeat for full documentation.
See Also:

numpy.repeat equivalent function

pandas.Index.reshape

Index.reshape(shape, order='C')
Returns an array containing the same data with a new shape.
Refer to numpy.reshape for full documentation.
See Also:

numpy.reshape equivalent function

pandas.Index.resize

Index.resize(new_shape, refcheck=True)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or n ints
Shape of resized array.
refcheck : bool, optional
If False, reference count will not be checked. Default is True.

Returns
None

Raises ValueError
If a does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError
If the order keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

resize Return a new array with the specified shape.

Notes
This reallocates space for the data area if necessary.
Only contiguous arrays (data elements consecutive in memory) can be resized.
The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set refcheck to False.
Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ...
ValueError: cannot resize an array that has been referenced ...
```

Unless `refcheck` is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

pandas.Index.round

Index.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.Index.searchsorted

Index.searchsorted`(v, side='left', sorter=None)`
Find indices where elements of v should be inserted in a to maintain order.
For full documentation, see `numpy.searchsorted`
See Also:

```python
numpy.searchsorted` equivalent function
```

### pandas.Index.set_names

```python
Index.set_names(names, inplace=False)
```
Set new names on index. Defaults to returning new index.

**Parameters**

- `names` : sequence
  names to set
- `inplace` : bool
  if True, mutates in place

**Returns**
new index (of same type and class...etc) [if inplace, returns None]

### pandas.Index.set_value

```python
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.setfield

```python
Index.setfield(val, dtype, offset=0)
```
Put a value into a specified place in a field defined by a data-type.
Place `val` into `a`'s field defined by `dtype` and beginning `offset` bytes into the field.

**Parameters**

- `val` : object
  Value to be placed in field.
- `dtype` : dtype object
  Data-type of the field in which to place `val`.
- `offset` : int, optional
  The number of bytes into the field at which to place `val`.

**Returns**
None

See Also:

`getfield`

**Examples**

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [5, 5, 5],
       [5, 5, 5]])
```
```python
>>> x
array([[ 1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
       [ 1.48219694e-323, 1.00000000e+000, 1.48219694e-323],
       [ 1.48219694e-323, 1.48219694e-323, 1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
```

**pandas.Index.setflags**

Index.setflags(\(write=None, align=None, uic=None\))

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by \(a\) (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

- **Parameters**
  - **write** : bool, optional
    Describes whether or not \(a\) can be written to.
  - **align** : bool, optional
    Describes whether or not \(a\) is aligned properly for its type.
  - **uic** : bool, optional
    Describes whether or not \(a\) is a copy of another “base” array.

**Notes**

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

- WRITEABLE (W) the data area can be written to;
- ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);
- UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

**Examples**

```python
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
```
>>> y.flags
   C_CONTIGUOUS : True
   F_CONTIGUOUS : False
   OWNDATA : True
   WRITEABLE : True
   ALIGNED : True
   UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
   C_CONTIGUOUS : True
   F_CONTIGUOUS : False
   OWNDATA : True
   WRITEABLE : False
   ALIGNED : False
   UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True

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)

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

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)

For an ordered Index, compute the 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
Returns  (start, end) : (int, int)

Notes
This function assumes that the data is sorted, so use at your own peril

pandas.Index.sort
Index.sort(*args, **kwargs)

pandas.Index.squeeze
Index.squeeze(axis=None)
    Remove single-dimensional entries from the shape of a.
    Refer to numpy.squeeze for full documentation.
See Also:
    numpy.squeeze equivalent function

pandas.Index.std
Index.std(axis=None, dtype=None, out=None, ddof=0)
    Returns the standard deviation of the array elements along given axis.
    Refer to numpy.std for full documentation.
See Also:
    numpy.std equivalent function

pandas.Index.sum
Index.sum(axis=None, dtype=None, out=None)
    Return the sum of the array elements over the given axis.
    Refer to numpy.sum for full documentation.
See Also:
    numpy.sum equivalent function

pandas.Index.summary
Index.summary(name=None)
pandas: powerful Python data analysis toolkit, Release 0.13.1

pandas.Index.swapaxes
Index.swapaxes(axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.
Refer to numpy.swapaxes for full documentation.
See Also:
numpy.swapaxes equivalent function
pandas.Index.take
Index.take(indexer, axis=0)
Analogous to 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()
return a series with both index and values equal to the index keys useful with map for returning an indexer
based on an index
pandas.Index.tofile
Index.tofile(fid, sep=”“, format=”%s”)
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can
be recovered using the function fromfile().
Parameters fid : file or str
An open file object, or a string containing a filename.
sep : str
Separator between array items for text output. If “” (empty), a binary file is
written, equivalent to file.write(a.tostring()).
format : str
Format string for text file output. Each entry in the array is formatted to text by
first converting it to the closest Python type, and then using “format” % item.

28.7. Index

1035


**Notes**

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

**pandas.Index.tolist**

Index.tolist()

Overridden version of ndarray.tolist

**pandas.Index.tostring**

Index.tostring(order='C')

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

**Parameters**

- **order**: {'C', 'F', None}, optional
  
  Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

- **s** : str
  
  A Python string exhibiting a copy of a’s raw data.

**Examples**

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tolist()
'
\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'
\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

**pandas.Index.trace**

Index.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

**See Also:**

- numpy.trace equivalent function
pandas.Index.transpose

Index.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters:

- axes : None, tuple of ints, or n ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
  - n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns:

out : ndarray

View of a, with axes suitably permuted.

See Also:

ndarray.T Array property returning the array transposed.

Examples

```python
g = np.array([[1, 2], [3, 4]])
g
g[0]
array([[1, 2],
       [3, 4]])
g.transpose()
array([[1, 3],
       [2, 4]])
g.transpose((1, 0))
array([[1, 3],
       [2, 4]])
g.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

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: powerful Python data analysis toolkit, Release 0.13.1

pandas.Index.unique

Index.unique()
Return array of unique values in the Index. Significantly faster than numpy.unique

Returns uniques : ndarray

pandas.Index.var

Index.var(axis=None, dtype=None, out=None, ddof=0)
Returns the variance of the array elements, along given axis.
Refer to numpy.var for full documentation.
See Also:
numpy.var equivalent function

pandas.Index.view

Index.view(*args, **kwargs)

28.7.2 Modifying and Computations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.copy([names, name, dtype, deep])</td>
<td>Make a copy of this object. Name and dtype sets those attributes on the new object.</td>
</tr>
<tr>
<td>Index.delete(loc)</td>
<td>Make new Index with passed location deleted</td>
</tr>
<tr>
<td>Index.diff(other)</td>
<td>Compute sorted set difference of two Index objects</td>
</tr>
<tr>
<td>Index.drop(labels)</td>
<td>Make new Index with passed list of labels deleted</td>
</tr>
<tr>
<td>Index.equals(other)</td>
<td>Determines if two Index objects contain the same elements.</td>
</tr>
<tr>
<td>Index.identical(other)</td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td>Index.insert(loc, item)</td>
<td>Make new Index inserting new item at location</td>
</tr>
<tr>
<td>Index.order([return_indexer, ascending])</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>Index.reindex(target[, method, level, ...])</td>
<td>For Index, simply returns the new index and the results of</td>
</tr>
<tr>
<td>Index.repeat(repeats[, axis])</td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td>Index.set_names(names[, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>Index.unique()</td>
<td>Return array of unique values in the Index. Significantly faster than</td>
</tr>
</tbody>
</table>

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 dtypes 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 deleted

**Returns**

new_index: Index

**pandas.Index.diff**

Index.diff(other)

Compute sorted set difference of two Index objects

**Returns**

diff: Index

**Notes**

One can do either of these and achieve the same result

```python
>>> index - index2
>>> index.diff(index2)
```

**pandas.Index.drop**

Index.drop(labels)

Make new Index with passed list of labels deleted

**Parameters**

labels: array-like

**Returns**

dropped: Index

**pandas.Index.equals**

Index.equals(other)

Determines if two Index objects contain the same elements.

**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

**Parameters**

loc: int

type: object

**Returns**

new_index: Index
pandas: powerful Python data analysis toolkit, Release 0.13.1

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, copy_if_needed=False, takeable=False)
For Index, simply returns the new index and the results of get_indexer. Provided here to enable an interface that is amenable for subclasses of Index whose internals are different (like MultiIndex)
Returns (new_index, indexer, mask) : tuple

pandas.Index.repeat

Index.repeat (repeats, axis=None)
Repeat elements of an array.
Refer to numpy.repeat for full documentation.
See Also:
numpy.repeat equivalent function

pandas.Index.set_names

Index.set_names (names, inplace=False)
Set new names on index. Defaults to returning new index.
Parameters names : sequence
   names to set
inplace : bool
   if True, mutates in place
Returns new index (of same type and class...etc) [if inplace, returns None]

pandas.Index.unique

Index.unique ()
Return array of unique values in the Index. Significantly faster than numpy.unique
Returns uniques : ndarray

28.7.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.astype(dtype)</td>
<td>Overridden version of ndarray.astype</td>
</tr>
<tr>
<td>Index.tolist()</td>
<td>Overridden version of ndarray.tolist</td>
</tr>
<tr>
<td>Index.to_datetime([dayfirst])</td>
<td>For an Index containing strings or datetime.datetime objects, attempt</td>
</tr>
<tr>
<td>Index.to_series()</td>
<td>return a series with both index and values equal to the index keys</td>
</tr>
</tbody>
</table>
pandas.Index.astype
Index.astype(dtype)

pandas.Index.tolist
Index.tolist()
    Overridden version of ndarray.tolist

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()
    return a series with both index and values equal to the index keys useful with map for returning an indexer based on an index

28.7.4 Sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.argsort(*args, **kwargs)</td>
<td>See docstring for ndarray.argsort</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)
    See docstring for ndarray.argsort

pandas.Index.order
Index.order(return_indexer=False, ascending=True)
    Return sorted copy of Index

pandas.Index.sort
Index.sort(*args, **kwargs)

28.7.5 Time-specific operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.shift([periods, freq])</td>
<td>Shift Index containing datetime objects by input number of periods and</td>
</tr>
</tbody>
</table>
**pandas: powerful Python data analysis toolkit, Release 0.13.1**

### pandas.Index.shift

**Index.shift**(periods=1, freq=None)
Shift Index containing datetime objects by input number of periods and DateOffset

**Returns**  
shifted : Index

### 28.7.6 Combining / joining / merging

<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.intersection(other)</td>
<td>Form the intersection of two Index objects. Sortedness of the result is not guaranteed</td>
</tr>
<tr>
<td>Index.join(other[, how, level, return_indexers])</td>
<td>Internal API method. Compute join_index and indexers to conform data structures to the new index.</td>
</tr>
<tr>
<td>Index.union(other)</td>
<td>Form the union of two Index objects and sorts if possible</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.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.join**

**Index.join**(other[, how='left', level=None, return_indexers=False])
Internal API method. Compute join_index and indexers to conform data structures to the new index.

**Parameters**  
other : Index
how : {'left', 'right', 'inner', 'outer'}
level :
return_indexers : boolean, default False

**Returns**  
join_index, (left_indexer, right_indexer)

**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
28.7.7 Selecting

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.get_indexer(target[, method, limit])</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>Index.get_indexer_non_unique(target, **kwargs)</code></td>
<td>Return an indexer suitable for taking from a non unique index.</td>
</tr>
<tr>
<td><code>Index.get_level_values(level)</code></td>
<td>Return vector of label values for requested level, equal to the length</td>
</tr>
<tr>
<td><code>Index.get_loc(key)</code></td>
<td>Get integer location for requested label</td>
</tr>
<tr>
<td><code>Index.get_value(series, key)</code></td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>Index.isin(values)</code></td>
<td>Compute boolean array of whether each index value is found in the</td>
</tr>
<tr>
<td><code>Index.slice_indexer([start, end, step])</code></td>
<td>For an ordered Index, compute the slice indexer for input labels and</td>
</tr>
<tr>
<td><code>Index.slice_locs([start, end])</code></td>
<td>For an ordered Index, compute the slice locations for input labels</td>
</tr>
</tbody>
</table>

**pandas.Index.get_indexer**

```python
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. The mask determines whether labels are found or not in the current index.

**Parameters**
- `target` : Index
- `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` : ndarray

**Notes**

This is a low-level method and probably should be used at your own risk

**Examples**

```python
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

**pandas.Index.get_indexer_non_unique**

```python
Index.get_indexer_non_unique (target, **kwargs)
```

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**

```python
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)
Get integer location for requested label

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.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.Index.slice_indexer**

Index.slice_indexer(start=None, end=None, step=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

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)
For an ordered Index, compute the 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

Returns (start, end): (int, int)
Notes

This function assumes that the data is sorted, so use at your own peril

28.7.8 Properties

<table>
<thead>
<tr>
<th>pandas.Index.is_monotonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index.is_monotonic</td>
</tr>
<tr>
<td>pandas.Index.is_numeric</td>
</tr>
<tr>
<td>Index.is_numeric()</td>
</tr>
</tbody>
</table>

28.8 DatetimeIndex

<table>
<thead>
<tr>
<th>DatetimeIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

28.8.1 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)

**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>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td>as18</td>
<td></td>
</tr>
<tr>
<td>asobject</td>
<td>Convert to Index of datetime objects</td>
</tr>
<tr>
<td>base</td>
<td>Base object if memory is from some other object.</td>
</tr>
<tr>
<td>ctypes</td>
<td>An object to simplify the interaction of the array with the ctypes module.</td>
</tr>
<tr>
<td>data</td>
<td>Python buffer object pointing to the start of the array’s data.</td>
</tr>
<tr>
<td>date</td>
<td>Returns numpy array of datetime.date.</td>
</tr>
<tr>
<td>day</td>
<td></td>
</tr>
<tr>
<td>dayofweek</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>dayofyear</td>
<td></td>
</tr>
<tr>
<td>dtype</td>
<td></td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>flat</td>
<td>A 1-D iterator over the array.</td>
</tr>
<tr>
<td>freq</td>
<td></td>
</tr>
<tr>
<td>freqstr</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td></td>
</tr>
<tr>
<td>imag</td>
<td>The imaginary part of the array.</td>
</tr>
<tr>
<td>inferred_type</td>
<td></td>
</tr>
<tr>
<td>is_all_dates</td>
<td></td>
</tr>
<tr>
<td>is_monotonic</td>
<td></td>
</tr>
<tr>
<td>itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>microsecond</td>
<td></td>
</tr>
<tr>
<td>minute</td>
<td></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></td>
</tr>
<tr>
<td>nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>nlevels</td>
<td></td>
</tr>
<tr>
<td>quarter</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>The real part of the array.</td>
</tr>
<tr>
<td>second</td>
<td></td>
</tr>
<tr>
<td>shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</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></td>
</tr>
<tr>
<td>week</td>
<td></td>
</tr>
<tr>
<td>weekday</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>weekofyear</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td></td>
</tr>
</tbody>
</table>
**pandas.DatetimeIndex.T**

DatetimeIndex.T

Same as self.transpose(), except that self is returned if self.ndim < 2.

**Examples**

```python
>>> x = np.array([[1., 2.],[3., 4.]])
```  
```python
array([[ 1., 2.],
       [ 3., 4.]])
```  
```python
>>> x.T
array([[ 1., 3.],
       [ 2., 4.]])
```  
```python
>>> x = np.array([1.,2.,3.,4.])
```  
```python
array([ 1., 2., 3., 4.])
```  
```python
>>> x.T
array([ 1., 2., 3., 4.])
```

**pandas.DatetimeIndex.asi8**

DatetimeIndex.asi8

**pandas.DatetimeIndex.asobject**

DatetimeIndex.asobject

Convert to Index of datetime objects

**pandas.DatetimeIndex.base**

DatetimeIndex.base

Base object if memory is from some other object.

**Examples**

The base of an array that owns its memory is None:

```python
>>> x = np.array([1,2,3,4])
```  
```python
>>> x.base is None
True
```  
Slicing creates a view, whose memory is shared with x:

```python
>>> y = x[2:]
```  
```python
>>> y.base is x
True
```
pandas.DatetimeIndex.ctypes

DatetimeIndex.ctypes

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters None

Returns c : Python object

Possessing attributes data, shape, strides, etc.

See Also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.
Examples

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
  c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
  c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```
pandas.DatetimeIndex.flags

DatetimeIndex.flags

pandas.DatetimeIndex.flat

DatetimeIndex.flat
A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

flatten Return a copy of the array collapsed into one dimension.

flatiter

Examples

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

pandas.DatetimeIndex.freq

DatetimeIndex.freq

pandas.DatetimeIndex.freqstr

DatetimeIndex.freqstr
pandas.DatetimeIndex.hour

DatetimeIndex.hour

pandas.DatetimeIndex.imag

DatetimeIndex.imag
The imaginary part of the array.

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

pandas.DatetimeIndex.inferred_type

DatetimeIndex.inferred_type

pandas.DatetimeIndex.is_all_dates

DatetimeIndex.is_all_dates

pandas.DatetimeIndex.is_monotonic

DatetimeIndex.is_monotonic

pandas.DatetimeIndex.itemsize

DatetimeIndex.itemsize
Length of one array element in bytes.

Examples

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

pandas.DatetimeIndex.microsecond

DatetimeIndex.microsecond
pandas: powerful Python data analysis toolkit, Release 0.13.1

pandas.DatetimeIndex.minute

DatetimeIndex.minute

pandas.DatetimeIndex.month

DatetimeIndex.month

The month as January=1, December=12

pandas.DatetimeIndex.names

DatetimeIndex.names

pandas.DatetimeIndex.nanosecond

DatetimeIndex.nanosecond

pandas.DatetimeIndex.nbytes

DatetimeIndex.nbytes

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

pandas.DatetimeIndex.ndim

DatetimeIndex.ndim

Number of array dimensions.

Examples

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```
pandas.DatetimeIndex.nlevels

DatetimeIndex.nlevels

pandas.DatetimeIndex.quarter

DatetimeIndex.quarter

pandas.DatetimeIndex.real

DatetimeIndex.real
The real part of the array.

See Also:

numpy.real equivalent function

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.      , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

pandas.DatetimeIndex.second

DatetimeIndex.second

pandas.DatetimeIndex.shape

DatetimeIndex.shape
Tuple of array dimensions.

Notes
May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]]
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

**pandas.DatetimeIndex.size**

DatetimeIndex.size

Number of elements in the array.

Equivalent to np.prod(a.shape), i.e., the product of the array's dimensions.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

**pandas.DatetimeIndex.strides**

DatetimeIndex.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element (i[0], i[1], ..., i[n]) in an array a is:

\[
\text{offset} = \sum \text{np.array}(i) \times a.\text{strides}
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

**See Also:**

numpy.lib.stride_tricks.as_strided

**Notes**

Imagine an array of 32-bit integers (each 4 bytes):

```python
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array \(x\) will be \((20, 4)\).
Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=np.sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = np.sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

```python
pandas.DatetimeIndex.time

DatetimeIndex.time
Returns numpy array of datetime.time. The time part of the Timestamps.
```

```python
pandas.DatetimeIndex.tzinfo

DatetimeIndex.tzinfo
Alias for tz attribute
```

```python
pandas.DatetimeIndex.values

DatetimeIndex.values
```

```python
pandas.DatetimeIndex.week

DatetimeIndex.week
```

```python
pandas.DatetimeIndex.weekday

DatetimeIndex.weekday
The day of the week with Monday=0, Sunday=6
```
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.DatetimeIndex.weekofyear**

DatetimeIndex.weekofyear

**pandas.DatetimeIndex.year**

DatetimeIndex.year

<table>
<thead>
<tr>
<th>inferred_freq</th>
<th>is_normalized</th>
<th>is_unique</th>
<th>name</th>
<th>offset</th>
<th>resolution</th>
</tr>
</thead>
</table>

**Methods**

- `all([axis, out])` Returns True if all elements evaluate to True.
- `any([axis, out])` Returns True if any of the elements of *a* evaluate to True.
- `append(other)` Append a collection of Index options together
- `argmax([axis, out])` Return indices of the maximum values along the given axis.
- `argmin()`
- `argsort(*args, **kwargs)` See docstring for ndarray.argsort
- `asof(label)` For a sorted index, return the most recent label up to and including the past
- `asof_locs(where, mask)` where : array of timestamps
- `astype(dtype)`
- `byteswap(inplace)` Swap the bytes of the array elements
- `choose(choices[, out, mode])` Use an index array to construct a new array from a set of choices.
- `clip(a_min, a_max[, out])` Return an array whose values are limited to *[a_min, a_max]*.
- `compress(condition[, axis, out])` Return selected slices of this array along given axis.
- `conj()` Complex-conjugate all elements.
- `conjugate()` Return the complex conjugate, element-wise.
- `copy([names, name, dtype, deep])` Make a copy of this object.
- `cumprod([axis, dtype, out])` Return the cumulative product of the elements along the given axis.
- `cumsum([axis, dtype, out])` Return the cumulative sum of the elements along the given axis.
- `delete(loc)` Make new DatetimeIndex with passed location deleted
- `diagonal([offset, axis1, axis2])` Return specified diagonals.
- `diff(other)` Compute sorted set difference of two Index objects
- `dot(b[, out])` Dot product of two arrays.
- `drop(labels)` Make new Index with passed list of labels deleted
- `dump(file)` Dump a pickle of the array to the specified file.
- `dump()` Returns the pickle of the array as a string.
- `equals(other)` Determines if two Index objects contain the same elements.
- `fill(*args, **kwargs)` This method will not function because object is immutable.
- `flatten(order)` Return a copy of the array collapsed into one dimension.
- `format([name, formatter])` Render a string representation of the Index
- `get_duplicates()`
- `get_indexer(target[, method, limit])` Compute indexer and mask for new index given the current index.
- `get_indexer_non_unique(target, **kwargs)` 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
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_loc(key)</td>
<td>Get integer location for requested 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></td>
</tr>
<tr>
<td>get_values()</td>
<td></td>
</tr>
<tr>
<td>getfield(dtype[, offset])</td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td>groupby(f)</td>
<td></td>
</tr>
<tr>
<td>holds_integer()</td>
<td></td>
</tr>
<tr>
<td>identical(other)</td>
<td>Similar to equals, but check that other comparable attributes are</td>
</tr>
<tr>
<td>indexer_at_time(time[, asof])</td>
<td>Select values at particular time of day (e.g., 9:00-9:30 AM)</td>
</tr>
<tr>
<td>indexer_between_time(start_time, end_time[, ...])</td>
<td>Select values between particular times of day (e.g., 9:00-9:30 AM)</td>
</tr>
<tr>
<td>insert(loc, item)</td>
<td>Make new Index inserting new item at location</td>
</tr>
<tr>
<td>intersection(other)</td>
<td>Specialized intersection for DatetimeIndex objects. May be much faster</td>
</tr>
<tr>
<td>is_(other)</td>
<td>More flexible, faster check like is but that works through views</td>
</tr>
<tr>
<td>is_float()</td>
<td></td>
</tr>
<tr>
<td>is_integer()</td>
<td></td>
</tr>
<tr>
<td>is_lexsorted_for_tuple(tup)</td>
<td></td>
</tr>
<tr>
<td>is_numeric()</td>
<td></td>
</tr>
<tr>
<td>is_type_compatible(typ)</td>
<td></td>
</tr>
<tr>
<td>isin(values)</td>
<td>Compute boolean array of whether each index value is found in the</td>
</tr>
<tr>
<td>item(*args)</td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td>items(*args, **kwargs)</td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td>join(other[, how, level, return_indexers])</td>
<td>See Index.join</td>
</tr>
<tr>
<td>map(f)</td>
<td></td>
</tr>
<tr>
<td>max([axis])</td>
<td>Overridden ndarray.max to return a Timestamp</td>
</tr>
<tr>
<td>mean([axis, dtype, out])</td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td>min([axis])</td>
<td>Overridden ndarray.min to return a Timestamp</td>
</tr>
<tr>
<td>newbyteorder([new_order])</td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td>nonzero()</td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td>normalize()</td>
<td>Return DatetimeIndex with times to midnight. Length is unaltered</td>
</tr>
<tr>
<td>order([return_indexer, ascending])</td>
<td>Return sorted copy of Index</td>
</tr>
<tr>
<td>prod([axis, dtype, out])</td>
<td>Return the product of the array elements over the given axis</td>
</tr>
<tr>
<td>ptp([axis, out])</td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td>put(*args, **kwargs)</td>
<td>This method will not function because object is immutable.</td>
</tr>
<tr>
<td>ravel(order)</td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td>reindex(target[, method, level, limit, ...])</td>
<td>For Index, simply returns the new index and the results of</td>
</tr>
<tr>
<td>rename(name[, inplace])</td>
<td>Set new names on index</td>
</tr>
<tr>
<td>repeat(repeats[, axis])</td>
<td>Analogous to ndarray.repeat</td>
</tr>
<tr>
<td>reshape(shape[, order])</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>resize(new_shape[, refcheck])</td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td>round([decimals, out])</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>searchsorted(key[, side])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>set_names(names[, inplace])</td>
<td>Set new names on index.</td>
</tr>
<tr>
<td>set_value(arr, key, value)</td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td>setfield(val, dtype[, offset])</td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td>setflags([write, align, uic])</td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively</td>
</tr>
<tr>
<td>shift(n[, freq])</td>
<td>Specialized shift which produces a DatetimeIndex</td>
</tr>
<tr>
<td>slice_indexer([start, end, step])</td>
<td>Index.slice_indexer, customized to handle time slicing</td>
</tr>
<tr>
<td>slice_locs([start, end])</td>
<td>Index.slice_locs, customized to handle partial ISO-8601 string slicing</td>
</tr>
<tr>
<td>snap([freq])</td>
<td>Snap time stamps to nearest occurring frequency</td>
</tr>
</tbody>
</table>
### Table 28.86 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort(*args, **kwargs)</td>
<td>Remove single-dimensional entries from the shape of ( a ).</td>
</tr>
<tr>
<td>squeeze(axis)</td>
<td>Remove single-dimensional entries from the shape of ( a ).</td>
</tr>
<tr>
<td>std(axis, dtype, out, ddof)</td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td>sum(axis, dtype, out)</td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td>summary(name)</td>
<td></td>
</tr>
<tr>
<td>swapaxes(axis1, axis2)</td>
<td>Return a view of the array with ( axis1 ) and ( axis2 ) interchanged.</td>
</tr>
<tr>
<td>take(indices[, axis])</td>
<td>Analogous to ndarray.take</td>
</tr>
<tr>
<td>to_datetime(dayfirst)</td>
<td></td>
</tr>
<tr>
<td>to_native_types(slicer)</td>
<td>slice and dice then format</td>
</tr>
<tr>
<td>to_period(freq)</td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>to_pydatetime()</td>
<td>Return DatetimeIndex as object ndarray of datetime.datetime objects</td>
</tr>
<tr>
<td>to_series()</td>
<td>return a series with both index and values equal to the index keys</td>
</tr>
<tr>
<td>tofile(fid[, sep, format])</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>tolist()</td>
<td>See ndarray.tolist</td>
</tr>
<tr>
<td>tostring(order)</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>transpose(axes)</td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td>tz_convert(tz)</td>
<td>Convert DatetimeIndex from one time zone to another (using pytz)</td>
</tr>
<tr>
<td>tz_localize(tz[, infer_dst])</td>
<td>Localize tz-naive DatetimeIndex to given time zone (using pytz)</td>
</tr>
<tr>
<td>union(other)</td>
<td>Specialized union for DatetimeIndex objects. If combine</td>
</tr>
<tr>
<td>union_many(others)</td>
<td>A bit of a hack to accelerate unioning a collection of indexes</td>
</tr>
<tr>
<td>unique()</td>
<td>Index.unique with handling for DatetimeIndex metadata</td>
</tr>
<tr>
<td>var(axis, dtype, out, ddof)</td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td>view(*args, **kwargs)</td>
<td></td>
</tr>
</tbody>
</table>

#### pandas.DatetimeIndex.all

```
DatetimeIndex.all(axis=None, out=None)
```

Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See Also:

- `numpy.all` equivalent function

#### pandas.DatetimeIndex.any

```
DatetimeIndex.any(axis=None, out=None)
```

Returns True if any of the elements of \( a \) evaluate to True.

Refer to `numpy.any` for full documentation.

See Also:

- `numpy.any` equivalent function

#### 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, out=None)

Return indices of the maximum values along the given axis.

Refer to numpy.argmax for full documentation.

See Also:

numpy.argmax equivalent function

**pandas.DatetimeIndex.argmin**

DatetimeIndex.argmin()

**pandas.DatetimeIndex.argsort**

DatetimeIndex.argsort(*args, **kwargs)

See docstring for 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

**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.byteswap**

DatetimeIndex.byteswap(inplace)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters inplace: bool, optional

If True, swap bytes in-place, default is False.

Returns out: ndarray

The byteswapped array. If inplace is True, this is a view to self.
Examples

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

```
pandas.DatetimeIndex.choose

DatetimexIndex.choose(choices, out=None, mode='raise')
Use an index array to construct a new array from a set of choices.
Refer to numpy.choose for full documentation.
See Also:

numpy.choose equivalent function
```

```
pandas.DatetimeIndex.clip

DatetimexIndex.clip(a_min, a_max, out=None)
Return an array whose values are limited to [a_min, a_max].
Refer to numpy.clip for full documentation.
See Also:

numpy.clip equivalent function
```

```
pandas.DatetimeIndex.compress

DatetimexIndex.compress(condition, axis=None, out=None)
Return selected slices of this array along given axis.
Refer to numpy.compress for full documentation.
See Also:

numpy.compress equivalent function
```
pandas.DatetimeIndex.conj

DatetimeIndex.conj()
Complex-conjugate all elements.
Refer to numpy.conjugate for full documentation.
See Also:

numpy.conjugate equivalent function

pandas.DatetimeIndex.conjugate

DatetimeIndex.conjugate()
Return the complex conjugate, element-wise.
Refer to numpy.conjugate for full documentation.
See Also:

numpy.conjugate equivalent function

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.cumprod

DatetimeIndex.cumprod(axis=None, dtype=None, out=None)
Return the cumulative product of the elements along the given axis.
Refer to numpy.cumprod for full documentation.
See Also:

numpy.cumprod equivalent function
pandas.DatetimeIndex.cumsum

```
DatetimeIndex.cumsum(axis=None, dtype=None, out=None)
```

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also:

`numpy.cumsum` equivalent function

pandas.DatetimeIndex.delete

```
DatetimeIndex.delete(loc)
```

Make new DatetimeIndex with passed location deleted

Returns `new_index` : DatetimeIndex

pandas.DatetimeIndex.diagonal

```
DatetimeIndex.diagonal(offset=0, axis1=0, axis2=1)
```

Return specified diagonals.

Refer to `numpy.diagonal()` for full documentation.

See Also:

`numpy.diagonal` equivalent function

pandas.DatetimeIndex.diff

```
DatetimeIndex.diff(other)
```

Compute sorted set difference of two Index objects

Returns `diff` : Index

Notes

One can do either of these and achieve the same result

```python
>>> index - index2
>>> index.diff(index2)
```

pandas.DatetimeIndex.dot

```
DatetimeIndex.dot(b, out=None)
```

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See Also:

`numpy.dot` equivalent function
Examples

```python
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```python
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

**pandas.DatetimeIndex.drop**

DatetimeIndex.drop(labels)

Make new Index with passed list of labels deleted

- **Parameters** labels : array-like
- **Returns** dropped : Index

**pandas.DatetimeIndex.dump**

DatetimeIndex.dump(file)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

- **Parameters** file : str
  A string naming the dump file.

**pandas.DatetimeIndex.dumps**

DatetimeIndex.dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

- **Parameters** None

**pandas.DatetimeIndex.equals**

DatetimeIndex.equals(other)

Determines if two Index objects contain the same elements.

**pandas.DatetimeIndex.fill**

DatetimeIndex.fill(*args, **kwargs)

This method will not function because object is immutable.
pandas.DatetimeIndex.flatten

DatetimeIndex.\texttt{flatten}(\texttt{order}='C')

Return a copy of the array collapsed into one dimension.

\textbf{Parameters} \quad \texttt{order}: \{‘C’, ‘F’, ‘A’\}, optional

Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from \texttt{a}. The default is ‘C’.

\textbf{Returns} \quad \texttt{y}: ndarray

A copy of the input array, flattened to one dimension.

\textbf{See Also:}

\texttt{ravel} \quad \text{Return a flattened array.}

\texttt{flat} \quad \text{A 1-D flat iterator over the array.}

\textbf{Examples}

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

pandas.DatetimeIndex.format

DatetimeIndex.\texttt{format}(\texttt{name}=False, \texttt{formatter}=None, **\texttt{kwargs})

Render a string representation of the Index

pandas.DatetimeIndex.get_duplicates

DatetimeIndex.\texttt{get_duplicates}()

pandas.DatetimeIndex.get_indexer

DatetimeIndex.\texttt{get_indexer}(\texttt{target}, \texttt{method}=None, \texttt{limit}=None)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to \texttt{ndarray.take} to align the current data to the new index. The mask determines whether labels are found or not in the current index.

\textbf{Parameters} \quad \texttt{target}: Index

\texttt{method}: \{‘pad’, ‘ffill’, ‘backfill’, ‘bfill’\}

\texttt{pad / ffill}: propagate LAST valid observation forward to next valid backfill \texttt{/ bfill}: use NEXT valid observation to fill gap

\textbf{Returns} \quad \texttt{indexer}: ndarray
Notes

This is a low-level method and probably should be used at your own risk

Examples

```python
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

pandas.DatetimeIndex.get_indexer_non_unique

DatetimelIndex.get_indexer_non_unique(target, **kwargs)

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

DatetimelIndex.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

DatetimelIndex.get_loc(key)

Get integer location for requested label

Returns
- loc : int

pandas.DatetimeIndex.get_value

DatetimelIndex.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

DatetimelIndex.get_value_maybe_box(series, key)

pandas.DatetimeIndex.get_values

DatetimelIndex.get_values()
pandas.DatetimeIndex.getfield

DatetimeIndex.getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by
the given type and the offset into the current array in bytes. The offset needs to be such that the view
dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view
with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters
dtype: str or dtype

The data type of the view. The dtype size of the view can not be larger than that
of the array itself.

offset: int

Number of bytes to skip before beginning the element view.

Examples

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
       [ 0.,  4.]])
```

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
- **tz**: string or pytz.timezone
  
  Time zone for time. Corresponding timestamps would be converted to time zone
  of the TimeSeries

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, 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_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_type_compatible

DatetimeIndex.is_type_compatible(typ)

pandas.DatetimeIndex.isin

DatetimeIndex.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.DatetimeIndex.item

DatetimeIndex.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters  *args : Arguments (variable number and type)

• none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.

• int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.

• tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns  z : Standard Python scalar object
A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of \( a \) is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

`item` is very similar to \( a[\text{args}] \), except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

Examples

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

pandas.DatetimeIndex.itemset

`DatetimeIndex.itemset(*args, **kwargs)`

This method will not function because object is immutable.

pandas.DatetimeIndex.join

`DatetimeIndex.join(other, how='left', level=None, return_indexers=False)`

See Index.join

pandas.DatetimeIndex.map

`DatetimeIndex.map(f)`

pandas.DatetimeIndex.max

`DatetimeIndex.max(axis=None)`

Overridden ndarry.max to return a Timestamp
pandas.DatetimeIndex.mean

```
DatetimeIndex.mean(axis=None, dtype=None, out=None)
```

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also:

`numpy.mean` equivalent function

pandas.DatetimeIndex.min

```
DatetimeIndex.min(axis=None)
```

Overridden ndarray.min to return a Timestamp

pandas.DatetimeIndex.newbyteorder

```
DatetimeIndex.newbyteorder(new_order='S')
```

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbyteorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

- `new_order` : string, optional
  Byte order to force; a value from the byte order specifications above. `new_order` codes can be any of:

  - 'S' - swap dtype from current to opposite endian
  - {'<', 'L'} - little endian
  - {'>', 'B'} - big endian
  - {'=', 'N'} - native order
  - {'|', 'I'} - ignore (no change to byte order)

  The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of `new_order` for the alternatives above. For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

**Returns**

- `new_arr` : array
  New array object with the dtypes reflecting given change to the byte order.

pandas.DatetimeIndex.nonzero

```
DatetimeIndex.nonzero()
```

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also:

`numpy.nonzero` equivalent function
**pandas.DatetimeIndex.normalize**

```python
DatetimeIndex.normalize()
```

Return DatetimeIndex with times to midnight. Length is unaltered.

**Returns**

- `normalized` : DatetimeIndex

**pandas.DatetimeIndex.order**

```python
DatetimeIndex.order(return_indexer=False, ascending=True)
```

Return sorted copy of Index.

**pandas.DatetimeIndex.prod**

```python
DatetimeIndex.prod(axis=None, dtype=None, out=None)
```

Return the product of the array elements over the given axis.

Refer to `numpy.prod` for full documentation.

**See Also:**

- `numpy.prod` equivalent function

**pandas.DatetimeIndex.ptp**

```python
DatetimeIndex.ptp(axis=None, out=None)
```

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

**See Also:**

- `numpy.ptp` equivalent function

**pandas.DatetimeIndex.put**

```python
DatetimeIndex.put(*args, **kwargs)
```

This method will not function because object is immutable.

**pandas.DatetimeIndex.ravel**

```python
DatetimeIndex.ravel(order)
```

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

**See Also:**

- `numpy.ravel` equivalent function
- `ndarray.flat` a flat iterator on the array.
pandas.DatetimeIndex.reindex

```
DatetimeIndex.reindex(target, method=None, level=None, limit=None, copy_if_needed=False, takeable=False)
```

For Index, simply returns the new index and the results of get_indexer. Provided here to enable an interface that is amenable for subclasses of Index whose internals are different (like MultiIndex)

Returns (new_index, indexer, mask) : tuple

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.reshape

```
DatetimeIndex.reshape(shape, order='C')
```

Returns an array containing the same data with a new shape.

Refer to numpy.reshape for full documentation.

See Also:

numpy.reshape equivalent function

pandas.DatetimeIndex.resize

```
DatetimeIndex.resize(new_shape, refcheck=True)
```

Change shape and size of array in-place.

Parameters new_shape : tuple of ints, or n ints
    Shape of resized array.

refcheck : bool, optional
    If False, reference count will not be checked. Default is True.

Returns None

Raises ValueError
    If a does not own its own data or references or views to it exist, and the data memory must be changed.
SystemError
If the order keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

resize Return a new array with the specified shape.

Notes
This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set refcheck to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```python
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless refcheck is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
```
>>> c
array([[0]])

datetime_index.round
datetime_index.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

datetime_index.searchsorted
datetime_index.searchsorted(key, side='left')

datetime_index.set_names
datetime_index.set_names(names, inplace=False)

Set new names on index. Defaults to returning new index.

Parameters:

names : sequence
    names to set

inplace : bool
    if True, mutates in place

Returns
    new index (of same type and class...etc) [if inplace, returns None]

datetime_index.set_value
datetime_index.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you’re doing

datetime_index.setfield
datetime_index.setfield(val, dtype, offset=0)

Put a value into a specified place in a field defined by a data-type.

Place val into a’s field defined by dtype and beginning offset bytes into the field.

Parameters:

val : object
    Value to be placed in field.

dtype : dtype object
    Data-type of the field in which to place val.

offset : int, optional
    The number of bytes into the field at which to place val.
Returns None

See Also:

gffield

Examples

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

pandas.DatetimeIndex.setflags

```
DatetimeIndex.setflags(write=None, align=None, uic=None)
```

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

- **write**: bool, optional
  
  Describes whether or not a can be written to.

- **align**: bool, optional
  
  Describes whether or not a is aligned properly for its type.

- **uic**: bool, optional
  
  Describes whether or not a is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;
ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

**Examples**

```python
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
 C_CONTIGUOUS : True
 F_CONTIGUOUS : False
 OWNDATA : True
 WRITEABLE : True
 ALIGNED : True
 UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
 C_CONTIGUOUS : True
 F_CONTIGUOUS : False
 OWNDATA : True
 WRITEABLE : False
 ALIGNED : False
 UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

**pandas.DatetimeIndex.shift**

```python
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**

```python
DatetimeIndex.slice_indexer(start=None, end=None, step=None)
```

Index.slice_indexer, customized to handle time slicing

**pandas.DatetimeIndex.slice_locs**

```python
DatetimeIndex.slice_locs(start=None, end=None)
```

Index.slice_locs, customized to handle partial ISO-8601 string slicing
**pandas.DatetimeIndex.snap**

DatetimeIndex.snap(freq='S')

Snap time stamps to nearest occurring frequency

**pandas.DatetimeIndex.sort**

DatetimeIndex.sort(*args, **kwargs)

**pandas.DatetimeIndex.squeeze**

DatetimeIndex.squeeze(axis=None)

Remove single-dimensional entries from the shape of a.

Refer to numpy.squeeze for full documentation.

See Also:

numpy.squeeze equivalent function

**pandas.DatetimeIndex.std**

DatetimeIndex.std(axis=None, dtype=None, out=None, ddof=0)

Returns the standard deviation of the array elements along given axis.

Refer to numpy.std for full documentation.

See Also:

numpy.std equivalent function

**pandas.DatetimeIndex.sum**

DatetimeIndex.sum(axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis.

Refer to numpy.sum for full documentation.

See Also:

numpy.sum equivalent function

**pandas.DatetimeIndex.summary**

DatetimeIndex.summary(name=None)

**pandas.DatetimeIndex.swapaxes**

DatetimeIndex.swapaxes(axis1, axis2)

Return a view of the array with axis1 and axis2 interchanged.

Refer to numpy.swapaxes for full documentation.

See Also:
pandas: powerful Python data analysis toolkit, Release 0.13.1

numpy.swapaxes equivalent function
pandas.DatetimeIndex.take
DatetimeIndex.take(indices, axis=0)
Analogous to ndarray.take
pandas.DatetimeIndex.to_datetime
DatetimeIndex.to_datetime(dayfirst=False)
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()
return a series with both index and values equal to the index keys useful with map for returning an indexer
based on an index
pandas.DatetimeIndex.tofile
DatetimeIndex.tofile(fid, sep=”“, format=”%s”)
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can
be recovered using the function fromfile().
Parameters fid : file or str
An open file object, or a string containing a filename.
sep : str
Separator between array items for text output. If “” (empty), a binary file is
written, equivalent to file.write(a.tostring()).
format : str

1078

Chapter 28. API Reference


Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

pandas.DatetimeIndex.tolist

DatetimeIndex.tolist()

See ndarray.tolist

pandas.DatetimeIndex.tostring

DatetimeIndex.tostring(order='C')

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

Parameters

order : {‘C’, ‘F’, None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : str

A Python string exhibiting a copy of a’s raw data.

Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00\x00\x00\x00'
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

pandas.DatetimeIndex.trace

DatetimeIndex.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

See Also:

numpy.trace equivalent function
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.DatetimeIndex.transpose**

**DatetimeIndex.transpose(** `axes` **)**

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

- `axes`: None, tuple of ints, or `n` ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: `i` in the `j`-th place in the tuple means `a`’s `i`-th axis becomes `a.transpose()`’s `j`-th axis.
  - `n` ints: same as an `n`-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

**Returns**

- `out`: ndarray
  View of `a`, with axes suitably permuted.

**See Also:**

- `ndarray.T`: Array property returning the array transposed.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
```

**pandas.DatetimeIndex.tz_convert**

**DatetimeIndex.tz_convert**( `tz` )

Convert DatetimeIndex from one time zone to another (using pytz)

**Returns**

- `normalized`: DatetimeIndex

**See Also:**

- `tz_localize`

**pandas.DatetimeIndex.tz_localize**

**DatetimeIndex.tz_localize**( `tz`, `infer_dst=False` )

Localize tz-naive DatetimeIndex to given time zone (using pytz)
Parameters  tz : string or pytz.timezone
    Time zone for time. Corresponding timestamps would be converted to time zone
    of the TimeSeries

    infer_dst : boolean, default False
    Attempt to infer fall dst-transition hours based on order

Returns  localized : DatetimeIndex

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 metadata

Returns  result : DatetimeIndex

pandas.DatetimeIndex.var

DatetimeIndex.var(axis=None, dtype=None, out=None, ddof=0)
    Returns the variance of the array elements, along given axis.
    Refer to numpy.var for full documentation.
    See Also:

    numpy.var  equivalent function

pandas.DatetimeIndex.view

DatetimeIndex.view(*args, **kwargs)

28.8.2  Time/Date Components

<table>
<thead>
<tr>
<th>DatetimeIndex.year</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.month</td>
</tr>
</tbody>
</table>

Continued on next page
Table 28.87 – continued from previous page

<table>
<thead>
<tr>
<th>DatetimeIndex.day</th>
<th>Returns numpy array of datetime.date.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.hour</td>
<td>Returns numpy array of datetime.time.</td>
</tr>
<tr>
<td>DatetimeIndex.minute</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.second</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.microsecond</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.nanosecond</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.dayofyear</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>DatetimeIndex.weekofyear</td>
<td>The day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>DatetimeIndex.week</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.dayofweek</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.weekday</td>
<td></td>
</tr>
<tr>
<td>DatetimeIndex.quarter</td>
<td></td>
</tr>
</tbody>
</table>

pandas.DatetimeIndex.year

`DatetimeIndex.year`

pandas.DatetimeIndex.month

`DatetimeIndex.month`

The month as January=1, December=12

```
pandas.DatetimeIndex.day
```

`DatetimeIndex.day`

pandas.DatetimeIndex.hour

`DatetimeIndex.hour`

pandas.DatetimeIndex.minute

`DatetimeIndex.minute`

pandas.DatetimeIndex.second

`DatetimeIndex.second`

pandas.DatetimeIndex.microsecond

`DatetimeIndex.microsecond`

pandas.DatetimeIndex.nanosecond

`DatetimeIndex.nanosecond`
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.DatetimeIndex.dayofyear

DatetimeIndex.dayofyear

pandas.DatetimeIndex.weekofyear

DatetimeIndex.weekofyear

pandas.DatetimeIndex.week

DatetimeIndex.week

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

28.8.3 Selecting

<table>
<thead>
<tr>
<th>DatetimeIndex.indexer_at_time(time[, asof])</th>
<th>Select values at particular time of day (e.g. 9:30AM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.indexer_between_time(...[, ...])</td>
<td>Select values between particular times of day (e.g., 9:00-9:30AM)</td>
</tr>
</tbody>
</table>

pandas.DatetimeIndex.indexer_at_time

DatetimeIndex.indexer_at_time(time, asof=False)
Select values at particular time of day (e.g. 9:30AM)

Parameters
time : datetime.time or string
**tz** : string or pytz.timezone

Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries

**Returns**

**values_at_time** : TimeSeries

### pandas.DatetimeIndex.indexer_between_time

**DatetimeIndex indexer_between_time**

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, default None

**Returns**

**values_between_time** : TimeSeries

### 28.8.4 Time-specific operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.normalize()</td>
<td>Return DatetimeIndex with times to midnight. Length is unaltered</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 DatetimeIndex from one time zone to another (using pytz)</td>
</tr>
<tr>
<td>DatetimeIndex.tz_localize(tz, infer_dst)</td>
<td>Localize tz-naive DatetimeIndex to given time zone (using pytz)</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 DatetimeIndex from one time zone to another (using pytz)

**Returns**

**normalized** : DatetimeIndex
### pandas.DatetimeIndex.tz_localize

**DatetimeIndex.tz_localize**(tz, infer_dst=False)

Localize tz-naive DatetimeIndex to given time zone (using pytz)

- **Parameters**
  - `tz`: string or pytz.timezone
    - Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries
  - `infer_dst`: boolean, default False
    - Attempt to infer fall dst-transition hours based on order

- **Returns**
  - `localized`: DatetimeIndex

#### 28.8.5 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.to_datetime([dayfirst])</td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>DatetimeIndex.to_period([freq])</td>
<td>Cast to PeriodIndex at a particular frequency</td>
</tr>
<tr>
<td>DatetimeIndex.to_pydatetime()</td>
<td>Return DatetimeIndex as object ndarray of datetime.datetime objects</td>
</tr>
</tbody>
</table>

### pandas.DatetimeIndex.to_datetime

DatetimeIndex.to_datetime\(\text{(dayfirst=False)}\)

### pandas.DatetimeIndex.to_period

DatetimeIndex.to_period\(\text{(freq=None)}\)

#### 28.9 GroupBy

GroupBy objects are returned by groupby calls: pandas.DataFrame.groupby(), pandas.Series.groupby(), etc.

#### 28.9.1 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupBy.<strong>iter</strong>()</td>
<td>Groupby iterator</td>
</tr>
<tr>
<td>GroupBy.groups</td>
<td>dict {group name -&gt; group labels}</td>
</tr>
<tr>
<td>GroupBy.indices</td>
<td>dict {group name -&gt; group indices}</td>
</tr>
<tr>
<td>GroupBy.get_group(name[, obj])</td>
<td>Constructs NDFrame from group with provided name</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.core.groupby.GroupBy.**\_\_\_iter\_\_

\_\_\_iter\_\_()  

Groupby iterator  

**Returns** Generator yielding sequence of (name, subsetted object)  

for each group

**pandas.core.groupby.GroupBy.groups**

\_\_\_iter\_\_()  

**dict** [group name -> group labels]

**pandas.core.groupby.GroupBy.indices**

\_\_\_iter\_\_()  

dict [group name -> group indices]

**pandas.core.groupby.GroupBy.get_group**

\_\_\_iter\_\_()  

**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

28.9.2 Function application

**GroupBy.apply** *(func, *args, **kwargs)*  

Apply function and combine results together in an intelligent way.  

**GroupBy.aggregate** *(func, *args, **kwargs)*  

**GroupBy.transform** *(func, *args, **kwargs)*

**pandas.core.groupby.GroupBy.apply**

\_\_\_iter\_\_()  

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
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

pandas.core.groupby.GroupBy.aggregate

GroupBy.aggregate(func, *args, **kwargs)

pandas.core.groupby.GroupBy.transform

GroupBy.transform(func, *args, **kwargs)

28.9.3 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.std(ddof)</td>
<td>Compute standard deviation of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.var(ddof)</td>
<td>Compute variance of groups, excluding missing values</td>
</tr>
<tr>
<td>GroupBy.ohlc()</td>
<td>Compute sum of values, excluding missing values</td>
</tr>
</tbody>
</table>

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.std

GroupBy.std(ddof=1)
Compute standard deviation of groups, excluding missing values
For multiple groupings, the result index will be a MultiIndex
**pandas.core.groupby.GroupBy.var**

GroupBy.var(ddof=1)

Compute variance of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**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.common.isnull**

pandas.core.common.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

- **isnull**: 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.

**pandas.core.common.notnull**

pandas.core.common.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

- **isnull**: 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.

**pandas.core.reshape.get_dummies**

pandas.core.reshape.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False)

Convert categorical variable into dummy/indicator variables

Parameters

- **data**: array-like or Series
  - String to append DataFrame column names

- **prefix**: string, default None
  - If appending prefix, separator/delimiter to use

- **dummy_na**: bool, default False
  - Add a column to indicate NaNs, if False NaNs are ignored.
Returns `dummies` : DataFrame

Examples

```python
>>> s = pd.Series(list('abca'))
```
```python
>>> get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```
```python
>>> s1 = ['a', 'b', np.nan]
```
```python
>>> get_dummies(s1)
   a  b
0  1  0
1  0  1
2  0  0
```
```python
>>> get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0  0
1  0  1  0
2  0  0  1
```

See also `Series.str.get_dummies`.

**pandas.io.clipboard.read_clipboard**

`pandas.io.clipboard.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

**pandas.io.excel.ExcelFile.parse**

`ExcelFile.parse(sheetname, 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, **kwds)`

Read an Excel table into DataFrame

Parameters `sheetname` : string or integer

Name of Excel sheet or the page number of the sheet

`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)

**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”)

**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 \rightarrow 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

**Returns** parsed : DataFrame

DataFrame parsed from the Excel file

**pandas.io.excel.read_excel**

**pandas.io.excel.read_excel**(io, sheetname, **kwds)

Read an Excel table into a pandas DataFrame

**Parameters**

- **io** : string, file-like object or xld workbook
  
  If a string, expected to be a path to.xls or .xlsx file

- **sheetname** : string
  
  Name of Excel sheet

- **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)

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

gine: 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.

Returns parsed : DataFrame
   DataFrame from the passed in Excel file

pandas.io.html.read_html

pandas.io.html.read_html (io, match='.', flavor=None, header=None, index_col=None,
   skiprows=None, infer_types=None, attrs=None, parse_dates=False,
   tupleize_cols=False, thousands='', )
Read HTML tables into a list of DataFrame objects.

Parameters io : str or file-like
   A URL, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with 'https' you might try removing the 's'.

match : str or compiled regular expression, optional
The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to ‘.+’ (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

**flavor**: str or None, container of strings

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**: bool, optional

This option is deprecated in 0.13, an will have no effect in 0.14. It defaults to True.

**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. In 0.13, this parameter can sometimes interact strangely with infer_types. If you get a large number of NaT values in your results, consider passing infer_types=False and manually converting types afterwards.

**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 ‘,’.

Returns  dfs : list of DataFrames

See Also:
pandas.io.parsers.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.

pandas.io.json.read_json

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 orients ‘index’ and ‘columns’.
  – The DataFrame columns must be unique for orients ‘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
Read CSV (comma-separated) 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 ','
  Delimiter to use. If sep is None, will try to automatically determine this. Regular expressions are accepted.

- `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

- `dtype`: Type name or dict of column -> type
  Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

- `compression`: {'gzip', 'bz2', None}, default None
  For on-the-fly decompression of on-disk data

- `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 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)

**skiprows** : list-like or integer
Row numbers to skip (0-indexed) or number of rows 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 or None (default)
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.
**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. Does not support line commenting (will return empty line)

**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 line at bottom of file to skip

**converters**: dict. optional

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 (e.g. ‘utf-8’)

**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

**Returns**

**result**: DataFrame or TextParser

---

**pandas.io.parsers.read_fwf**

**pandas.io.parsers.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

dtype : Type name or dict of column -> type
    Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None
    For on-the-fly decompression of on-disk data

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 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)

skiprows : list-like or integer
    Row numbers to skip (0-indexed) or number of rows 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 or None (default)
    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.

dayfirst: boolean, default False
    DD/MM format dates, international and European format

thousands: str, default None
    Thousands separator

column: str, default None
    Indicates remainder of line should not be parsed Does not support line commenting (will return empty line)
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 line at bottom of file to skip

converters: dict, optional
    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’)

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

Returns  **result** : DataFrame or TextParser

Also, ‘delimiter’ is used to specify the filler character of the fields if it is not spaces (e.g., ‘~’).

**pandas.io.parsers.read_table**

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.

**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

**dtype** : Type name or dict of column -> type
Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

**compression** : {'gzip', 'bz2', None}, default None
For on-the-fly decompression of on-disk data

**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 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)

**skiprows** : list-like or integer
Row numbers to skip (0-indexed) or number of rows 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 or None (default)
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.parse to do the conversion.

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 Does not support line commenting (will return empty line)

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 line at bottom of file to skip
converters : dict. optional

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’)

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

Returns result : DataFrame or TextParser

pandas.io.pickle.read_pickle

pandas.io.pickle.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

pandas.io.pytables.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 represenation
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.io.pytables.HDFStore.get

HDFStore.get(key)
Retrieve pandas object stored in file

Parameters  key : object

Returns  obj : type of object stored in file
pandas: powerful Python data analysis toolkit, Release 0.13.1

**pandas.io.pytables.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

**pandas.io.pytables.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 convertible) objects, optional
- start: integer (defaults to None), row number to start selection
- stop: integer (defaults to None), row number to stop selection
- columns: a list of columns that if not None, will limit the return columns
- iterator: boolean, return an iterator, default False
- chunksize: nrows to include in iteration, return an iterator
- auto_close: boolean, should automatically close the store when finished, default is False

**Returns**
The selected object

**pandas.io.pytables.read_hdf**

pandas.io.pytables.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 convertible) 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

auto_close : optional, boolean, should automatically close the store
when finished, default is False

Returns  The selected object

pandas.io.sql.read_sql

pandas.io.sql.read_sql(sql, con, index_col=None, coerce_float=True, params=None)

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 will be 0 to
len(results) - 1.

Parameters  sql: string

SQL query to be executed

con: DB connection object, optional

index_col: string, optional
column name to use 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 or tuple, optional

List of parameters to pass to execute method.

pandas.io.stata.read_stata

pandas.io.stata.read_stata(filepath_or_buffer, convert_dates=True, convert_categoricals=True,
encoding=None, index=None)

Read Stata file into DataFrame

Parameters  filepath_or_buffer : string or file-like object

Path to .dta file or object implementing a binary read() functions

convert_dates : boolean, defaults to True

Convert date variables to DataFrame time values

convert_categoricals : boolean, defaults to True

Read value labels and convert columns to Categorical/Factor variables

encoding : string, None or encoding
Encoding used to parse the files. Note that Stata doesn’t support unicode. None defaults to cp1252.

**index** : identifier of index column

identifier of column that should be used as index of the DataFrame

### pandas.stats.moments.ewma

**pandas.stats.moments.ewma**(*arg*, *com=None*, *span=None*, *halflife=None*, *min_periods=0*,
*freq=None*, *time_rule=None*, *adjust=True*)

Exponentially-weighted moving average

**Parameters**

- **arg** : Series, DataFrame
- **com** : float, optional
  Center of mass: \( \alpha = \frac{1}{1 + \text{com}} \),
- **span** : float, optional
  Specify decay in terms of span, \( \alpha = \frac{2}{(\text{span} + 1)} \)
- **halflife** : float, optional
  Specify decay in terms of halflife, \( \alpha = 1 - \exp(\log(0.5) / \text{halflife}) \)
- **min_periods** : int, default 0
  Number of observations in sample to require (only affects beginning)
- **freq** : None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic time_rule is a legacy alias for freq
- **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)

**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 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.

### pandas.stats.moments.ewmcorr

**pandas.stats.moments.ewmcorr**(*arg1*, *arg2*, *com=None*, *span=None*, *halflife=None*,
*min_periods=0*, *freq=None*, *time_rule=None*)

Exponentially-weighted moving correlation
Parameters arg1 : Series, DataFrame, or ndarray
    arg2 : Series, DataFrame, or ndarray
    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, :math: \alpha = 1 - \exp(log(0.5) / \text{halflife})
    min_periods : int, default 0
        Number of observations in sample to require (only affects beginning)
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic time_rule is a legacy alias for freq
    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)

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 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.
**min_periods** : int, default 0
Number of observations in sample to require (only affects beginning)

**freq** : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

**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)

**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 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.

**pandas.stats.moments.ewmstd**

```
pandas.stats.moments.ewmstd(arg, com=None, span=None, halflife=None, min_periods=0, bias=False, time_rule=None)
```

Exponentially-weighted moving std

**Parameters**  
arg : Series, DataFrame

**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, :math: alpha = 1 - \exp(\log(0.5) / halflife)

**min_periods** : int, default 0
Number of observations in sample to require (only affects beginning)

**freq** : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

**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)

**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 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.

**pandas.stats.moments.ewmvar**

\[
\text{pandas.stats.moments.ewmvar}(\text{arg}, \text{com=None, span=None, halflife=None, min_periods=0, bias=False, freq=None, time_rule=None})
\]

Exponentially-weighted moving variance

**Parameters**  
\( \text{arg} \): Series, DataFrame

- \( \text{com} \): float, optional
  
  Center of mass: \( \alpha = 1/(1 + \text{com}) \),

- \( \text{span} \): float, optional
  
  Specify decay in terms of span, \( \alpha = 2/(\text{span} + 1) \)

- \( \text{halflife} \): float, optional
  
  Specify decay in terms of halflife, \( \alpha = 1 - \exp(\log(0.5)/\text{halflife}) \)

- \( \text{min_periods} \): int, default 0
  
  Number of observations in sample to require (only affects beginning)

- \( \text{freq} \): None or string alias / date offset object, default=None
  
  Frequency to conform to before computing statistic time_rule is a legacy alias for freq

- \( \text{adjust} \): boolean, default True
  
  Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

- \( \text{bias} \): 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 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.
pandas.stats.moments.expanding_apply

**pandas.stats.moments.expanding_apply** *(arg, func, min_periods=1, freq=None, center=False, time_rule=None)*

Generic expanding function application

**Parameters**
- **arg** : Series, DataFrame
- **func** : function
  - Must produce a single value from an ndarray input
- **min_periods** : int
  - Minimum number of observations in window required to have a value
- **freq** : None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **center** : boolean, default False
  - Whether the label should correspond with center of window
- **time_rule** : Legacy alias for freq

**Returns**
- **y** : type of input argument

pandas.stats.moments.expanding_corr

**pandas.stats.moments.expanding_corr** *(arg1, arg2, min_periods=1, freq=None, center=False, time_rule=None)*

Expanding sample correlation

**Parameters**
- **arg1** : Series, DataFrame, or ndarray
- **arg2** : Series, DataFrame, or ndarray
- **min_periods** : int
  - Minimum number of observations in window required to have a value
- **freq** : None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

**Returns**
- **y** : type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series -> Computes result for each column Series / Series -> Series

pandas.stats.moments.expanding_count

**pandas.stats.moments.expanding_count** *(arg, freq=None, center=False, time_rule=None)*

Expanding count of number of non-NaN observations.

**Parameters**
- **arg** : DataFrame or numpy ndarray-like
- **freq** : None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **center** : boolean, default False
  - Whether the label should correspond with center of window
time_rule : Legacy alias for freq

Returns  expanding_count : type of caller

pandas.stats.moments.expanding_cov

pandas.stats.moments.expanding_cov(arg1, arg2, min_periods=1, freq=None, center=False, time_rule=None)

Unbiased expanding covariance

Parameters  arg1 : Series, DataFrame, or ndarray
arg2 : Series, DataFrame, or ndarray
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic

Returns  y : type depends on inputs
DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
Computes result for each column Series / Series -> Series

pandas.stats.moments.expanding_kurt

pandas.stats.moments.expanding_kurt(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Unbiased expanding kurtosis

Parameters  arg : Series, DataFrame
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic

Returns  y : type of input argument

pandas.stats.moments.expanding_mean

pandas.stats.moments.expanding_mean(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Expanding mean

Parameters  arg : Series, DataFrame
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic

Returns  y : type of input argument
**pandas.stats.moments.expanding_median**

`pandas.stats.moments.expanding_median(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)`

O(N log(window)) implementation using skip list

Expanding median

**Parameters**

- **arg**: Series, DataFrame
- **min_periods**: int
  
  Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  
  Frequency to conform to before computing statistic

**Returns**

- **y**: type of input argument

**pandas.stats.moments.expanding_quantile**

`pandas.stats.moments.expanding_quantile(arg, quantile, min_periods=1, freq=None, center=False, time_rule=None)`

Expanding quantile

**Parameters**

- **arg**: Series, DataFrame
- **quantile**: 0 <= quantile <= 1
- **min_periods**: int
  
  Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  
  Frequency to conform to before computing statistic
- **center**: boolean, default False
  
  Whether the label should correspond with center of window
- **time_rule**: Legacy alias for freq

**Returns**

- **y**: Legacy alias for freq

**pandas.stats.moments.expanding_skew**

`pandas.stats.moments.expanding_skew(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)`

Unbiased expanding skewness

**Parameters**

- **arg**: Series, DataFrame
- **min_periods**: int
  
  Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  
  Frequency to conform to before computing statistic

**Returns**

- **y**: type of input argument
**pandas.stats.moments.expanding_std**

**pandas.stats.moments.expanding_std**(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Unbiased expanding standard deviation

- **Parameters**
  - **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

- **Returns**
  - **y**: type of input argument

**pandas.stats.moments.expanding_sum**

**pandas.stats.moments.expanding_sum**(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Expanding sum

- **Parameters**
  - **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

- **Returns**
  - **y**: type of input argument

**pandas.stats.moments.expanding_var**

**pandas.stats.moments.expanding_var**(arg, min_periods=1, freq=None, center=False, time_rule=None, **kwargs)

Unbiased expanding variance

- **Parameters**
  - **arg**: Series, DataFrame
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic

- **Returns**
  - **y**: type of input argument

**pandas.stats.moments.rolling_apply**

**pandas.stats.moments.rolling_apply**(arg, window, func, min_periods=None, freq=None, center=False, time_rule=None)

Generic moving function application
Parameters

arg : Series, DataFrame
    window : Number of observations used for calculating statistic
    func : function
        Must produce a single value from an ndarray input
    min_periods : int
        Minimum number of observations in window required to have a value
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
    center : boolean, default False
        Whether the label should correspond with center of window
    time_rule : Legacy alias for freq

Returns

y : type of input argument

pandas.stats.moments.rolling_corr

pandas.stats.moments.rolling_corr(arg1, arg2, window, min_periods=None, freq=None, center=False, time_rule=None)

Moving sample correlation

Parameters

arg1 : Series, DataFrame, or ndarray
    arg2 : Series, DataFrame, or ndarray
    window : Number of observations used for calculating statistic
    min_periods : int
        Minimum number of observations in window required to have a value
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
time_rule is a legacy alias for freq
    center : boolean, default False
        Whether the label should correspond with center of window

Returns

y : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
Computes result for each column Series / Series -> Series

pandas.stats.moments.rolling_count

pandas.stats.moments.rolling_count(arg, window, freq=None, center=False, time_rule=None)

Rolling count of number of non-NaN observations inside provided window.

Parameters

arg : DataFrame or numpy ndarray-like
    window : Number of observations used for calculating statistic
    freq : None or string alias / date offset object, default=None
        Frequency to conform to before computing statistic
    center : boolean, default False
        Whether the label should correspond with center of window
time_rule: Legacy alias for freq

Returns rolling_count: type of caller

pandas.stats.moments.rolling_cov

pandas.stats.moments.rolling_cov(arg1, arg2, window, min_periods=None, freq=None, center=False, time_rule=None)

Unbiased moving covariance

Parameters arg1: Series, DataFrame, or ndarray
arg2: Series, DataFrame, or ndarray
window: Number of observations used for calculating statistic
min_periods: int
Minimum number of observations in window required to have a value
freq: None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns y: type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series ->
Computes result for each column Series / Series -> Series

pandas.stats.moments.rolling_kurt

pandas.stats.moments.rolling_kurt(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Unbiased moving kurtosis

Parameters arg: Series, DataFrame
window: Number of observations used for calculating statistic
min_periods: int
Minimum number of observations in window required to have a value
freq: None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns y: type of input argument

pandas.stats.moments.rolling_mean

pandas.stats.moments.rolling_mean(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Moving mean

Parameters arg: Series, DataFrame
window: Number of observations used for calculating statistic
min_periods: int
Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic **time_rule** is a legacy alias for **freq**

**Returns**  
y : type of input argument

### pandas.stats.moments.rolling_median

**pandas.stats.moments.rolling_median**(arg, window, **min_periods=None**, **freq=None**, **cen-**

ter=False, **time_rule=None**, **kwargs**)  

O(N log(window)) implementation using skip list

Moving median

**Parameters**  
**arg** : Series, DataFrame

**window** : Number of observations used for calculating statistic

**min_periods** : int

Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic **time_rule** is a legacy alias for **freq**

**Returns**  
y : type of input argument

### pandas.stats.moments.rolling_quantile

**pandas.stats.moments.rolling_quantile**(arg, window, **quantile**, **min_periods=None**, **freq=None**, **cen-**

ter=False, **time_rule=None**)  

Moving quantile

**Parameters**  
**arg** : Series, DataFrame

**window** : Number of observations used for calculating statistic

**quantile** : 0 <= quantile <= 1

**min_periods** : int

Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

**center** : boolean, default False

Whether the label should correspond with center of window

**time_rule** : Legacy alias for **freq**

**Returns**  
y : type of input argument
pandas.stats.moments.rolling_skew

pandas.stats.moments.rolling_skew(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Unbiased moving skewness

Parameters

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
  - time_rule is a legacy alias for freq

Returns

- **y**: type of input argument

pandas.stats.moments.rolling_std

pandas.stats.moments.rolling_std(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Unbiased moving standard deviation

Parameters

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
  - time_rule is a legacy alias for freq

Returns

- **y**: type of input argument

pandas.stats.moments.rolling_sum

pandas.stats.moments.rolling_sum(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Moving sum

Parameters

- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
  - time_rule is a legacy alias for freq

Returns

- **y**: type of input argument
pandas.stats.moments.rolling_var

**pandas.stats.moments.rolling_var** *(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)*

Unbiased moving variance

**Parameters**

- **arg**: Series, DataFrame
  - 
  - **window**: Number of observations used for calculating statistic
  - **min_periods**: int
    - Minimum number of observations in window required to have a value
  - **freq**: None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic
  - **time_rule** is a legacy alias for **freq**

**Returns**

- **y**: type of input argument

pandas.tools.merge.concat

**pandas.tools.merge.concat** *(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False)*

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**: 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). Any None objects will be dropped silently unless they are all None in which case an Exception 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.

**Returns**  
**concatenated** : type of objects

**Notes**

The keys, levels, and names arguments are all optional

```
import pandas as pd

# Example

df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['a', 'b', 'c'])
df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]}, index=['c', 'a', 'b'])

# Concatenate
result = pd.concat([df1, df2], ignore_index=True)
```

### pandas.tools.merge.merge

```
pandas.tools.merge.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

Examples

```python
>>> A
<table>
<thead>
<tr>
<th>lkey</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
</tr>
<tr>
<td>baz</td>
<td>3</td>
</tr>
<tr>
<td>foo</td>
<td>4</td>
</tr>
</tbody>
</table>

>>> B
<table>
<thead>
<tr>
<th>rkey</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>bar</td>
<td>6</td>
</tr>
<tr>
<td>qux</td>
<td>7</td>
</tr>
</tbody>
</table>

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
<table>
<thead>
<tr>
<th>lkey</th>
<th>value_x</th>
<th>rkey</th>
<th>value_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>2</td>
<td>bar</td>
<td>6</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
<td>bar</td>
<td>8</td>
</tr>
<tr>
<td>baz</td>
<td>3</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>foo</td>
<td>1</td>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>foo</td>
<td>4</td>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>qux</td>
<td>7</td>
</tr>
</tbody>
</table>
```

pandas.tools.pivot.pivot_table

pandas.tools.pivot.pivot_table(data, values=None, rows=None, cols=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
rows : list of column names or arrays to group on
    Keys to group on the x-axis of the pivot table
cols : list of column names or arrays to group on
    Keys to group on the y-axis of the pivot table
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', rows=['A', 'B'],
                      cols=['C'], aggfunc=np.sum)
```

```python
>>> table
     small  large
   foo  one  1   4
       two  6   NaN
   bar  one  5   4
       two  6   7
```

**pandas.tseries.tools.to_datetime**

**pandas.tseries.tools.to_datetime**(arg, errors='ignore', dayfirst=False, utc=None, box=True, format=None, coerce=False, 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 ob-
  jects 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”
coerce : force errors to NaT (False by default)

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

Examples

Take separate series and convert to datetime

>>> 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')

Or from strings

>>> df = df.astype(str)
>>> pd.to_datetime(df.day + df.month + df.year, format='%d%m%Y')
CONTRIBUTING TO PANDAS

See the following links:

- The developer pages on the website
- Guidelines on bug reports and pull requests
- Some extra tips on using git

29.1 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.

29.1.1 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 (whatsnew, 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:

  ```python
  x = 2
  x**3
  ```

  will be rendered as

  In [1]: x = 2
  
  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.

## 29.1.2 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.

Furthermore, it is recommended to have all **optional dependencies** installed. This is not needed, 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_version()` to get an overview of the installed version of all dependencies.

**Warning:** Building the docs with Sphinx version 1.2 is broken. Use the latest stable version (1.2.1) or the older 1.1.3.

### Building pandas

For a step-by-step overview on how to set up your environment, to work with the pandas code and git, see the developer pages. When you start to work on some docs, be sure to update your code to the latest development version (‘master’):

```bash
git fetch upstream
git rebase upstream/master
```

Often it will be necessary to rebuild the C extension after updating:

```bash
python setup.py build_ext --inplace
```
Building the documentation

So how do you build the docs? Navigate to your local the folder `pandas/doc/` directory in the console and run:

```
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:

```
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 turn-around time for checking your changes. You will be prompted to delete unrequired `.rst` files, since the last committed version can always be restored from git.

```
# 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 doc build may take 10 minutes. a `--no-api` build may take 3 minutes and a single section may take 15 seconds.

### 29.1.3 Where to start?

There are a number of issues listed under Docs and Good as first PR where you could start out.

Or maybe you have an idea of you own, by using pandas, looking for something in the documentation and thinking ‘this can be improved’, let’s do something about that!

Feel free to ask questions on mailing list or submit an issue on Github.
RELEAS NOTES

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

30.1 pandas 0.13.1

Release date: (February 3, 2014)

30.1.1 New features

• Added date_format and datetime_format attribute to ExcelWriter. (GH4133)

30.1.2 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).

30.1.3 Experimental Features

30.1.4 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)

30.1.5 Bug Fixes

• Bug in io.wb.get_countries not including all countries (GH6008)
• Bug in Series replace with timestamp dict (GH5797)
• read_csv/read_table now respects the prefix kwarg (GH5732).
• Bug in selection with missing values via .ix from a duplicate indexed DataFrame failing (GH5835)
- Fix issue of boolean comparison on empty DataFrames (GH5808)
- Bug in isnull handling NaT in an object array (GH5443)
- Bug in to_datetime when passed a np.nan or integer datelike and a format string (GH5863)
- Bug in groupby dtype conversion with datetimelike (GH5869)
- Regression in handling of empty Series as indexers to Series (GH5877)
- Bug in internal caching, related to (GH5727)
- Testing bug in reading json/msgpack from a non-filepath on windows under py3 (GH5874)
- Bug when assigning to .ix[tuple(...)] (GH5896)
- Bug in fully reindexing a Panel (GH5905)
- Bug in idxmin/max with object dtypes (GH5914)
- Bug in BusinessDay when adding n days to a date not on offset when n>5 and n%5==0 (GH5890)
- Bug in assigning to chained series with a series via ix (GH5928)
- Bug in creating an empty DataFrame, copying, then assigning (GH5932)
- Bug in DataFrame.tail with empty frame (GH5846)
- Bug in propagating metadata on resample (GH5862)
- Fixed string-representation of NaT to be "NaT" (GH5708)
- Fixed string-representation for Timestamp to show nanoseconds if present (GH5912)
- pd.match not returning passed sentinel
- Panel.to_frame() no longer fails when major_axis is a MultiIndex (GH5402).
- Bug in pd.read_msgpack with inferring a DateTimeIndex frequency incorrectly (GH5947)
- Fixed to_datetime for array with both Tz-aware datetimes and NaT's (GH5961)
- Bug in rolling skew/kurtosis when passed a Series with bad data (GH5749)
- Bug in scipy interpolate methods with a datetime index (GH5975)
- Bug in NaT comparison if a mixed datetime/np.datetime64 with NaT were passed (GH5968)
- Fixed bug with pd.concat losing dtype information if all inputs are empty (GH5742)
- Recent changes in IPython cause warnings to be emitted when using previous versions of pandas in QTConsole, now fixed. If you’re using an older version and need to suppress the warnings, see (GH5922).
- Bug in merging timedelta dtypes (GH5695)
- Bug in plotting.scatter_matrix function. Wrong alignment among diagonal and off-diagonal plots, see (GH5497).
- Regression in Series with a multi-index via ix (GH6018)
- Bug in Series.xs with a multi-index (GH6018)
- Bug in Series construction of mixed type with datelike and an integer (which should result in object type and not automatic conversion) (GH6028)
- Possible segfault when chained indexing with an object array under numpy 1.7.1 (GH6026, GH6056)
- Bug in setting using fancy indexing a single element with a non-scalar (e.g. a list), (GH6043)
- to_sql did not respect if_exists (GH4110 GH4304)
• Regression in .get(None) indexing from 0.12 (GH5652)
• Subtle iloc indexing bug, surfaced in (GH6059)
• Bug with insert of strings into DatetimeIndex (GH5818)
• Fixed unicode bug in to_html/HTML repr (GH6098)
• Fixed missing arg validation in get_options_data (GH6105)
• Bug in assignment with duplicate columns in a frame where the locations are a slice (e.g. next to each other) (GH6120)
• Bug in propagating _ref_locs during construction of a DataFrame with dups index/columns (GH6121)
• Bug in DataFrame.apply when using mixed datelike reductions (GH6125)
• Bug in DataFrame.append when appending a row with different columns (GH6129)
• Bug in DataFrame construction with recarray and non-ns datetime dtype (GH6140)
• Bug in .loc setitem indexing with a dataframe on rhs, multiple item setting, and a datetimelike (GH6152)
• Fixed a bug in query/eval during lexicographic string comparisons (GH6155).
• Fixed a bug in query where the index of a single-element Series was being thrown away (GH6148).
• 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)

30.2 pandas 0.13.0

Release date: January 3, 2014

30.2.1 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)

30.2.2 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 qt pandas DataFrameModel and DataFrameWidget.
• Added `pandas.io.gbq` for reading from (and writing to) Google BigQuery into a DataFrame. (GH4140)

30.2.3 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”, “iInf”, etc.) to infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
• Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)
• Significant table writing performance improvements in `HDFStore`
• `JSON` date serialization now performed in low-level C code.
• `JSON` support for encoding `datetime.time`
• Expanded JSON docs, more info about `orient` options and the use of the `numpy` param when decoding.
• Add `drop_level` argument to `xs` (GH4180)
• Can now resample a `DataFrame` with `ohlc` (GH2320)
Index.copy() and MultiIndex.copy() now accept keyword arguments to change attributes (i.e., names, levels, labels) (GH4039)

Add rename and set_names methods to Index as well as set_names, set_levels, set_labels to MultiIndex. (GH4039) with improved validation for all (GH4039, GH4794)

A Series of dtype timedelta64[ns] can now be divided/multiplied by an integer series (GH4521)

A Series of dtype timedelta64[ns] can now be divided by another timedelta64[ns] object to yield a float64 dtyped Series. This is frequency conversion; astyping is also supported.

Timedelta64 support fillna/ffill/bfill with an integer interpreted as seconds, or a timedelta (GH3371)

Box numeric ops on timedelta Series (GH4984)

Datetime64 support fillna/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 xls writer module. (GH4542)

allow DataFrame constructor to accept more list-like objects, e.g. list of collections.Sequence and array.Array objects (GH3783, GH4297, GH4851), thanks @lgautier

DataFrame constructor now accepts a numpy masked record array (GH3478), thanks @jnothman

__getitem__ with tuple key (e.g., [:, 2]) on Series without MultiIndex raises ValueError (GH4759, GH4837)

read_json now raises a (more informative) ValueError when the dict contains a bad key and orient=’split’ (GH4730, GH4838)

read_stata now accepts Stata 13 format (GH4291)

ExcelWriter and ExcelFile can be used as contextmanagers. (GH3441, GH4933)

pandas is now tested with two different versions of statsmodels (0.4.3 and 0.5.0) (GH4981).

Better string representations of MultiIndex (including ability to roundtrip via repr). (GH3347, GH4935)

Both ExcelFile and read_excel to accept an xld.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 (GH4759)
- **Python csv parser** now supports usecols (GH4335)
- **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)

### 30.2.4 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 datet ime.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 unserialisable objects. (GH5138)

• Index and MultiIndex changes (GH4039):
  - Setting levels and labels directly on MultiIndex is now deprecated. Instead, you can use the set_levels() and set_labels() methods.
  - levels, labels and names properties no longer return lists, but instead return containers that do not allow setting of items (‘mostly immutable’)
  - levels, labels and names are validated upon setting and are either copied or shallow-copied.
  - inplace setting of levels or labels now correctly invalidates the cached properties. (GH5238).
  - __deepcopy__ now returns a shallow copy (currently: a view) of the data - allowing metadata changes.
  - MultiIndex.astype() now only allows np.object_-like dtypes and now returns a MultiIndex rather than an Index. (GH4039)
  - Added is__ method to Index that allows fast equality comparison of views (similar to np.may_share_memory but no false positives, and changes on levels and labels setting on MultiIndex). (GH4859, GH4909)
  - Aliased __iadd__ to __add__. (GH4996)
  - Added is__ method to Index that allows fast equality comparison of views (similar to np.may_share_memory but no false positives, and changes on levels and labels setting on MultiIndex). (GH4859, GH4909)

• Infer and downcast dtype if downcast='infer' is passed to fillna/ffill/bfill (GH4604)
• __nonzero__ for all NDFrame objects, will now raise a ValueError, this reverts back to (GH1073, GH4633) behavior. Add .bool() method to NDFrame objects to facilitate evaluating of single-element boolean Series
• DataFrame.update() no longer raises a DataConflictError, it now will raise a ValueError instead (if necessary) (GH4732)
• Series.isin() and DataFrame.isin() now raise a TypeError when passed a string (GH4763). Pass a list of one element (containing the string) instead.
• Remove undocumented/unused kind keyword argument from read_excel, and ExcelFile. (GH4713, GH4712)
• The method argument of NDFrame.replace() is valid again, so that a a list can be passed to to_replace (GH4743).
• provide automatic dtype conversions on _reduce operations (GH3371)
• exclude non-numerics if mixed types with datelike in _reduce operations (GH3371)
• default for tupleize_cols is now False for both to_csv and read_csv. Fair warning in 0.12 (GH3604)
• moved timedeltas support to pandas.tseries.timedeltas.py; add timedeltas string parsing, add top-level to_timedelta function
• NDFrame now is compatible with Python’s toplevel abs() function (GH4821).
• raise a TypeError on invalid comparison ops on Series/DataFrame (e.g. integer/datetime) (GH4968)

  Added a new index type, Float64Index. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes [], ix, loc for scalar indexing and slicing work exactly the same. Indexing on other index types are preserved (and positional fallback for [], ix), with the exception, that floating point slicing on indexes on non Float64Index will raise a TypeError, e.g. Series(range(5))[3.5:4.5] (GH263; issue:5375)

• Make Categorical repr nicer (GH4368)
• Remove deprecated Factor (GH3650)
• Remove deprecated set_printoptions/reset_printoptions (issue:3046)
• Remove deprecated _verbose_info (GH3215)
• Begin removing methods that don’t make sense on GroupBy objects (GH4887).
• Remove deprecated read_clipboard/to_clipboard/ExcelFile/ExcelWriter from pandas.io.parsers (GH3177)
• 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)

### 30.2.5 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 propogate 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)
• Refactor of _get_numeric_data/_get_bool_data to core/generic.py, allowing Series/Panel functionality
• Refactor of Series arithmetic with time-like objects (datetime/timedelta/time etc.) into a separate, cleaned up wrapper class. (GH4613)
• Complex compat for Series with ndarray. (GH4819)
• Removed unnecessary rwproperty from codebase in favor of builtin property. (GH4843)
• Refactor object level numeric methods (mean/sum/min/max...) from object level modules to core/generic.py (GH4435).
• Refactor cum objects to core/generic.py (GH4435), note that these have a more numpy-like function signature.
• read_html() now uses TextParser to parse HTML data from bs4/lxml (GH4770).
• Removed the keep_internal keyword parameter in pandas/core/groupby.py because it wasn’t being used (GH5102).
• Base DateOffsets are no longer all instantiated on importing pandas, instead they are generated and cached on the fly. The internal representation and handling of DateOffsets has also been clarified. (GH5189, related GH5004)
• MultiIndex constructor now validates that passed levels and labels are compatible. (GH5213, GH5214)
• Unity dropna for Series/DataFrame signature (GH5250), tests from GH5234, courtesy of @rockg
• Rewrite assert_almost_equal() in cython for performance (GH4398)
• Added an internal _update_inplace method to facilitate updating NDFrame wrappers on inplace ops (only is for convenience of caller, doesn’t actually prevent copies). (GH5247)

30.2.6 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 serialised 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 cparsor with the usecols parameter (GH3192)
• Fix an issue in merging blocks where the resulting DataFrame had partially set _ref_locs (GH4403)
• Fixed an issue where hist subplots were being overwritten when they were called using the top level matplotlib API (GH4408)
• Fixed a bug where calling Series.astype(str) would truncate the string (GH4405, GH4437)
• Fixed a py3 compat issue where bytes were being repr’d as tuples (GH4455)
• Fixed Panel attribute naming conflict if item is named ‘a’ (GH3440)
• Fixed an issue where duplicate indexes were raising when plotting (GH4486)
• Fixed an issue where cumsum and cumprod didn’t work with bool dtypes (GH4170, GH4440)
• Fixed Panel slicing issued in xs that was returning an incorrect dimmed object (GH4016)
• Fix resampling bug where custom reduce function not used if only one group (GH3849, GH4494)
• Fixed Panel assignment with a transposed frame (GH3830)
• Raise on set indexing with a Panel and a Panel as a value which needs alignment (GH3777)
• frozenset objects now raise in the Series constructor (GH4482, GH4480)
• Fixed issue with sorting a duplicate multi-index that has multiple dtypes (GH4516)
• Fixed bug in DataFrame.set_values which was causing name attributes to be lost when expanding the index. (GH3742, GH4039)
• Fixed issue where individual names, levels and labels could be set on MultiIndex without validation (GH3714, GH4039)
- Fixed (GH3334) in pivot_table. Margins did not compute if values is the index.
- Fix bug in having a rhs of np.timedelta64 or np.offsets.DateOffset when operating with datetimes (GH4532)
- Fix arithmetic with series/datetimeindex and np.timedelta64 not working the same (GH4134) and buggy timedelta in numpy 1.6 (GH4135)
- Fix bug in pd.read_clipboard on windows with PY3 (GH4561); not decoding properly
- tslib.get_period_field() and tslib.get_period_field_arr() now raise if code argument out of range (GH4519, GH4520)
- Fix boolean indexing on an empty series loses index names (GH4235), infer_dtype works with empty arrays.
- Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue (GH3317)
- Fixed issue where DataFrame.apply was rerasing exceptions incorrectly (causing the original stack trace to be truncated).
- Fix selection with ix/loc and non_unique selectors (GH4619)
- Fix assignment with iloc/loc involving a dtype change in an existing column (GH4312, GH5702) have internal setitem_with_indexer in core/indexing to use Block.setitem
- Fixed bug where thousands operator was not handled correctly for floating point numbers in csv_import (GH4322)
- Fix an issue with CacheableOffset not properly being used by many DateOffset; this prevented the DateOffset from being cached (GH4609)
- Fix boolean comparison with a DataFrame on the lhs, and a list/tuple on the rhs (GH4576)
- Fix error/dtype conversion with setitem of None on Series/DataFrame (GH4667)
- Fix decoding based on a passed in non-default encoding in pd.read_stata (GH4626)
- Fix DataFrame.from_records with a plain-vanilla ndarray. (GH4727)
- Fix some inconsistencies with Index.rename and MultiIndex.rename, etc. (GH4718, GH4628)
- Bug in using iloc/loc with a cross-sectional and duplicate indices (GH4726)
- Bug with using QUOTE_NONE with to_csv causing Exception. (GH4328)
- Bug with Series indexing not raising an error when the right-hand-side has an incorrect length (GH2702)
- Bug in multi-indexing with a partial string selection as one part of a MultiIndex (GH4758)
- Bug with reindexing on the index with a non-unique index will now raise ValueError (GH4746)
- Bug in setting with loc/ix a single indexer with a multi-index axis and a numpy array, related to (GH3777)
- Bug in concatenation with duplicate columns across dtypes not merging with axis=0 (GH4771, GH4975)
- Bug in iloc with a slice index failing (GH4771)
- Incorrect error message with no colspeccs 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 comparions 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)
pandas: powerful Python data analysis toolkit, Release 0.13.1

- 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)

30.3 pandas 0.12.0

Release date: 2013-07-24

30.3.1 New features

- pd.read_html() can now parse HTML strings, files or urls and returns a list of DataFrame s courtesy of @cpcloud. (GH3477, GH3605, GH3606)
- Support for reading Amazon S3 files. (GH3504)
• Added module for reading and writing JSON strings/files: pandas.io.json includes **to_json** DataFrame/Series method, and a **read_json** top-level reader various issues (GH1226, GH3804, GH3876, GH3867, GH1305)

• Added module for reading and writing Stata files: pandas.io.stata (GH1512) includes **to_stata** DataFrame method, and a **read_stata** top-level reader

• Added support for writing in **to_csv** and reading in **read_csv**, multi-index columns. The **header** option in **read_csv** now accepts a list of the rows from which to read the index. Added the option, **tupleize_cols** to provide compatibility for the pre 0.12 behavior of writing and reading multi-index columns via a list of tuples. The default in 0.12 is to write lists of tuples and **not** interpret list of tuples as a multi-index column. Note: The default value will change in 0.12 to make the default **to** write and read multi-index columns in the new format. (GH3571, GH1651, GH3141)

• Add iterator to **Series.str** (GH3638)
• **pd.set_option()** now allows N option, value pairs (GH3667).

• Added keyword parameters for different types of **scatter_matrix** subplots

• A **filter** method on grouped Series or DataFrames returns a subset of the original (GH3680, GH919)

• Access to historical Google Finance data in pandas.io.data (GH3814)

• DataFrame plotting methods can sample column colors from a Matplotlib colormap via the **colormap** keyword. (GH3860)

### 30.3.2 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 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 date-time.min and datetime.max (respectively), thanks @SleepingPills
• `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

### 30.3.3 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 (GH3907)
• Deprecated display.height, display.width is now only a formatting option does not control triggering of summary, similar to < 0.11.0.
• Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if `True`, default is `False` (same as prior to 0.12) (GH3679)
• io API changes
  – added `pandas.io.api` for i/o imports
  – removed `Excel` support to `pandas.io.excel`
  – added top-level `pd.read_sql` and `to_sql` DataFrame methods
  – removed `clipboard` support to `pandas.io.clipboard`
  – replace top-level and instance methods `save` and `load` with top-level `read_pickle` and `to_pickle` instance method, `save` and `load` will give deprecation warning.
• the method and axis arguments of `DataFrame.replace()` are deprecated
• set FutureWarning to require data_source, and to replace year/month with expiry date in pandas.io options. This is in preparation to add options data from google (GH3822)
• the method and axis arguments of `DataFrame.replace()` are deprecated
• Implement `__nonzero__` for `NDFrame` objects (GH3691, GH3696)
• `as_matrix` with mixed signed and unsigned dtypes will result in 2 x the lcd of the unsigned as an int, maxing with `int64`, to avoid precision issues (GH3733)
• `na_values` in a list provided to `read_csv/read_excel` will match string and numeric versions e.g. `na_values=['99']` will match 99 whether the column ends up being int, float, or string (GH3611)
• `read_html` now defaults to `None` when reading, and falls back on `bs4 + html5lib` when lxml fails to parse. a list of parsers to try until success is also valid
• more consistency in the to_datetime return types (give string/array of string inputs) (GH3888)
• The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)
• New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)
30.3.4 Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. (GH2301)

30.3.5 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 simialry 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 item-by-item not upcasting correctly (GH3740)
- Incorrectly read a HDFStore multi-index Frame with a column specification (GH3748)
- **read_html** now correctly skips tests (GH3741)
- PandasObjects raise TypeError when trying to hash (GH3882)
- Fix incorrect arguments passed to concat that are not list-like (e.g. `concat(df1,df2)`) (GH3481)
- Correctly parse when passed the `dtype=str` (or other variable-len string dtypes) in `read_csv` (GH3795)
- Fix index name not propagating when using `loc/ix` (GH3880)
- Fix groupby when applying a custom function resulting in a returned DataFrame was not converting dtypes (GH3911)
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn’t working (GH3907)
- Fixed `__truediv__` in Python 2.7 with `numexpr` installed to actually do true division when dividing two integer arrays with at least 10000 cells total (GH3764)
- Indexing with a string with seconds resolution not selecting from a time index (GH3925)
- csv parsers would loop infinitely if `iterator=True` but no `chunksize` was specified (GH3967), python parser failing with `chunksize=1`
- Fix index name not propagating when using `shift`
- Fixed dropna=False being ignored with multi-index stack (GH3997)
- Fixed flattening of columns when renaming MultiIndex columns DataFrame (GH4004)
- Fix Series.clip for datetime series. NA/NaN threshold values will now throw ValueError (GH3996)
- Fixed insertion issue into DataFrame, after rename (GH4032)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- **Series.hist** will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of tox under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fix bug where HDFStore will fail to append because of a different block ordering on-disk (GH4096)
- Better error messages on inserting incompatible columns to a frame (GH4107)
- Fixed bug in **DataFrame.replace** where a nested dict wasn’t being iterated over when regex=False (GH4115)
- Fixed bug in convert_objects(convert_numeric=True) where a mixed numeric and object Series/Frame was not converting properly (GH4119)
- Fixed bugs in multi-index selection with column multi-index and duplicates (GH4145, GH4146)
- Fixed bug in the parsing of microseconds when using the format argument in to_datetime (GH4152)
- Fixed bug in PandasAutoDateLocator where invert_xaxis triggered incorrectly 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)

30.4 pandas 0.11.0

Release date: 2013-04-22

30.4.1 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 indexeres (GH3275)
30.4.2 Improvements to existing features

- Improved performance of df.to_csv() by up to 10x in some cases. (GH3059)
- added blocks attribute to DataFrames, to return a dict of dtypes to homogeneously dtyped DataFrames
- added keyword convert_numeric to convert_objects() to try to convert object dtypes to numeric types (default is False)
- convert_dates in convert_objects can now be coerce which will return a datetime64[ns] dtype with non-convertibles set as NaT; will preserve an all-nan object (e.g. strings), default is True (to perform soft-conversion)
- Series print output now includes the dtype by default
- Optimize internal reindexing routines (GH2819, GH2867)
- describe_option() now reports the default and current value of options.
- Add format option to pandas.to_datetime with faster conversion of strings that can be parsed with datetime.strptime
- Add axes property to Series for compatibility
- Add xs function to Series for compatibility
- Allow setitem in a frame where only mixed numerics are present (e.g. int and float), (GH3037)
- HDFStore
  - Provide dotted attribute access to get from stores (e.g. store.df == store['df'])
  - New keywords iterator=boolean, and chunksize=number_in_a_chunk are provided to support iteration on select and select_as_multiple (GH3076)
  - support read_hdf/to_hdf API similar to read_csv/to_csv (GH3222)
- Add squeeze method to possibly remove length 1 dimensions from an object.

```
In [1]: p = 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.reindex(items=['ItemA']).squeeze()
```

```
A    B    C    D
2001-01-02 0.469112 -0.282863 -1.509059 -1.135632
2001-01-03 1.212112 -0.173215  0.119209 -1.044236
2001-01-04 -0.861849 -2.104569 -0.494929  1.071804
2001-01-05  0.721555 -0.706771 -1.039575  0.271860
[4 rows x 4 columns]
```

```
In [3]: p.reindex(items=['ItemA'],minor=['B']).squeeze()
```

```
A    B    C    D
2001-01-02 0.469112 -0.282863 -1.509059 -1.135632
2001-01-03 1.212112 -0.173215  0.119209 -1.044236
2001-01-04 -0.861849 -2.104569 -0.494929  1.071804
2001-01-05  0.721555 -0.706771 -1.039575  0.271860
[4 rows x 4 columns]
```
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)

```python
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
[3 rows x 1 columns]
```

- 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)

### 30.4.3 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.
30.4.4 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 isin 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 inidices 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 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 similary to an ordered timeseries (GH2437).
• Fix implemented .xs when called with axes=1 and a level parameter (GH2903)
• Timestamp now supports the class method fromordinal similar to datetimes (GH3042)
• Fix issue with indexing a series with a boolean key and specifying a 1-len list on the rhs (GH2745) or a list on the rhs (GH3235)
• Fixed bug in groupby apply when kernel generate list of arrays having unequal len (GH1738)
• fixed handling of rolling_corr with center=True which could produce corr>1 (GH3155)
• Fixed issues where indices can be passed as ‘index/column’ in addition to 0/1 for the axis parameter
• PeriodIndex.tolist now boxes to Period (GH3178)
• PeriodIndex.get_loc KeyError now reports Period instead of ordinal (GH3179)
• df.to_records bug when handling MultiIndex (GH3189)
• Fix Series.__getitem__ segfault when index less than -length (GH3168)
• Fix bug when using Timestamp as a date parser (GH2932)
• Fix bug creating date range from Timestamp with time zone and passing same time zone (GH2926)
• Add comparison operators to Period object (GH2781)
• Fix bug when concatenating two Series into a DataFrame when they have the same name (GH2797)
• Fix automatic color cycling when plotting consecutive timeseries without color arguments (GH2816)
• fixed bug in the pickling of PeriodIndex (GH2891)
• Upcast/split blocks when needed in a mixed DataFrame when setitem with an indexer (GH3216)
• Invoking df.applymap on a dataframe with dupe cols now raises a ValueError (GH2786)
• Apply with invalid returned indices raise correct Exception (GH2808)
• Fixed a bug in plotting log-scale bar plots (GH3247)
• df.plot() grid on/off now obeys the mpl default style, just like series.plot(). (GH3233)
• Fixed a bug in the legend of plotting.andrews_curves() (GH3278)
• Produce a series on apply if we only generate a singular series and have a simple index (GH2893)
• Fix Python ascii file parsing when integer falls outside of floating point spacing (GH3258)
• fixed pretty printing of sets (GH3294)
• Panel() and Panel.from_dict() now respects ordering when give OrderedDict (GH3303)
• DataFrame where with a datetimelike incorrectly selecting (GH3311)
• Ensure index casts work even in Int64Index
• Fix set_index segfault when passing MultiIndex (GH3308)
• Ensure pickles created in py2 can be read in py3
• Insert ellipsis in MultiIndex summary repr (GH3348)
• Groupby will handle mutation among an input groups columns (and fallback to non-fast apply) (GH3380)
• Eliminated unicode errors on FreeBSD when using MPL GTK backend (GH3360)
• Period.strftime should return unicode strings always (GH3363)
• Respect passed read_* chunksize in get_chunk function (GH3406)

30.5 pandas 0.10.1

Release date: 2013-01-22

30.5.1 New features

• Add data interface to World Bank WDI pandas.io.wb (GH2592)

30.5.2 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
30.5.3 Improvements to existing features

- **HDFStore**
  - enables storing of multi-index dataframes (closes GH1277)
  - support data column indexing and selection, via `data_columns` keyword in append
  - support write chunking to reduce memory footprint, via `chunksize` keyword to append
  - support automagic indexing via `index` keyword to append
  - support `expectedrows` keyword in append to inform PyTables about the expected 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)

30.5.4 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 occurring 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)

30.6 pandas 0.10.0

Release date: 2012-12-17

30.6.1 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)
pandas: powerful Python data analysis toolkit, Release 0.13.1

- 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)
30.6.2 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

30.6.3 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/ffill 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

30.6.4 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 hierarchial 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)
### 30.6.5 Bug Fixes

- Fix major performance regression in DataFrame.iteritems (GH2273)
- Fixes bug when negative period passed to Series/DataFrame.diff (GH2266)
- Escape tabs in console output to avoid alignment issues (GH2038)
- Properly box datetime64 values when retrieving cross-section from mixed-dtype DataFrame (GH2272)
- Fix concatenation bug leading to GH2057, GH2257
- Fix regression in Index console formatting (GH2319)
- Box Period data when assigning PeriodIndex to frame column (GH2243, GH2281)
- Raise exception on calling reset_index on Series with inplace=True (GH2277)
- Enable setting multiple columns in DataFrame with hierarchical columns (GH2295)
- Respect dtype=object in DataFrame constructor (GH2291)
- Fix DatetimeIndex.join bug with tz-aware indexes and how='outer' (GH2317)
- pop(...) and del works with DataFrame with duplicate columns (GH2349)
- Treat empty strings as NA in date parsing (rather than let dateutil do something weird) (GH2263)
- Prevent uint64 -> int64 overflows (GH2355)
- Enable joins between MultiIndex and regular Index (GH2024)
- Fix time zone metadata issue when unioning non-overlapping DatetimeIndex objects (GH2367)
- Raise/handle int64 overflows in parsers (GH2247)
- Deleting of consecutive rows in HDFStore tables is much faster than before
- Appending on a HDFStore would fail if the table was not first created via put
- Use col_space argument as minimum column width in DataFrame.to_html (GH2328)
- Fix tz-aware DatetimeIndex.to_period (GH2232)
- Fix DataFrame row indexing case with MultiIndex (GH2314)
- Fix to_excel exporting issues with Timestamp objects in index (GH2294)
- Fixes assigning scalars and array to hierarchical column chunk (GH1803)
- Fixed a UnicodeDecodeError with series tidy_repr (GH2225)
- Fixed issued with duplicate keys in an index (GH2347, GH2380)
- Fixed issues re: Hash randomization, default on starting w/ py3.3 (GH2331)
- Fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- Fix Timestamp formatting with tzoffset time zone in dateutil 2.1 (GH2443)
- Fix GroupBy.apply issue when using BinGrouper to do ts binning (GH2300)
- Fix issues resulting from datetime.datetime columns being converted to datetime64 when calling DataFrame.apply. (GH2374)
- Raise exception when calling to_panel on non uniquely-indexed frame (GH2441)
- Improved detection of console encoding on IPython zmq frontends (GH2458)
- Preserve time zone when append-ing two time series (GH2260)
• Box timestamps when calling reset_index on time-zone-aware index rather than creating a tz-less datetime64 column (GH2262)
• Enable searching non-string columns in DataFrame.filter(like=...) (GH2467)
• Fixed issue with losing nanosecond precision upon conversion to DatetimeIndex(GH2252)
• Handle timezones in Datetime.normalize (GH2338)
• Fix test case where dtype specification with endianness causes failures on big endian machines (GH2318)
• Fix plotting bug where upsampling causes data to appear shifted in time (GH2448)
• Fix read_csv failure for UTF-16 with BOM and skiprows(GH2298)
• read_csv with names arg not implicitly setting header=None(GH2459)
• Unrecognized compression mode causes segfault in read_csv(GH2474)
• In read_csv, header=0 and passed names should discard first row(GH2269)
• Correctly route to stdout/stderr in read_table (GH2071)
• Fix exception when Timestamp.to_datetime is called on a Timestamp with tzoffset (GH2471)
• Fixed unintentional conversion of datetime64 to long in groupby.first() (GH2133)
• Union of empty DataFrames now return empty with concatenated index (GH2307)
• DataFrame.sort_index raises more helpful exception if sorting by column with duplicates (GH2488)
• DataFrame.to_string formatters can be list, too (GH2520)
• DataFrame.combine_first will always result in the union of the index and columns, even if one DataFrame is length-zero (GH2525)
• Fix several DataFrame.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)

30.7 pandas 0.9.1

Release date: 2012-11-14

30.7.1 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)
30.7.2 API Changes

- Upsampling period index “spans” intervals. Example: annual periods upscaled 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)

30.7.3 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)

30.7.4 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)
30.8 pandas 0.9.0

Release date: 10/7/2012

30.8.1 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)

30.8.2 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)
30.8.3 API Changes

- Change default header names in read_* functions to more Pythonic X0, X1, etc. instead of X.1, X.2. (GH2000)
- Deprecated day_of_year API removed from PeriodIndex, use dayofyear (GH1723)
- Don’t modify NumPy suppress printoption at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by HDFStore (GH1834, GH1824)
- Legacy cruft removed: pandas.stats.misc.quantileTS
- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)
- Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)
- Setting parts of DataFrame/Panel using ix now aligns input Series/DataFrame (GH1630)
- first and last methods in GroupBy no longer drop non-numeric columns (GH1809)
- Resolved inconsistencies in specifying custom NA values in text parser. na_values of type dict no longer override default NAs unless keep_default_na is set to false explicitly (GH1657)
- Enable skipfooter parameter in text parsers as an alias for skip_footer

30.8.4 Bug Fixes

- Perform arithmetic column-by-column in mixed-type DataFrame to avoid type upcasting issues. Caused downstream DataFrame.diff bug (GH1896)
- Fix matplotlib auto-color assignment when no custom spectrum passed. Also respect passed color keyword argument (GH1711)
- Fix resampling logical error with closed='left’ (GH1726)
- Fix critical DatetimeIndex.union bugs (GH1730, GH1719, GH1745, GH1702, GH1753)
- Fix critical DatetimeIndex.intersection bug with unanchored offsets (GH1708)
- Fix MM-YYYY time series indexing case (GH1672)
- Fix case where Categorical group key was not being passed into index in GroupBy result (GH1701)
- Handle Ellipsis in Series.__getitem__/__setitem__ (GH1721)
- Fix some bugs with handling datetime64 scalars of other units in NumPy 1.6 and 1.7 (GH1717)
- Fix performance issue in MultiIndex.format (GH1746)
- Fixed GroupBy bugs interacting with DatetimeIndex asof / map methods (GH1677)
- Handle factors with NAs in pandas.rpy (GH1615)
- Fix statsmodels import in pandas.stats.var (GH1734)
- Fix DataFrame repr/info summary with non-unique columns (GH1700)
- Fix Series.iget_value for non-unique indexes (GH1694)
- Don’t lose tzinfo when passing DatetimeIndex as DataFrame column (GH1682)
- Fix tz conversion with time zones that haven’t had any DST transitions since first date in the array (GH1673)
- Fix field access with UTC->local conversion on unsorted arrays (GH1756)
- Fix isnull handling of array-like (list) inputs (GH1755)
- Fix regression in handling of Series in Series constructor (GH1671)
- Fix comparison of Int64Index with DatetimeIndex (GH1681)
- Fix min_periods handling in new rolling_max/min at array start (GH1695)
- Fix errors with how='median' and generic NumPy resampling in some cases caused by SeriesBinGrouper (GH1648, GH1688)
- When grouping by level, exclude unobserved levels (GH1697)
- Don’t lose tzinfo in DatetimeIndex when shifting by different offset (GH1683)
- Hack to support storing data with a zero-length axis in HDFStore (GH1707)
- Fix DatetimeIndex tz-aware range generation issue (GH1674)
- Fix method='time' interpolation with intraday data (GH1698)
- Don’t plot all-NA DataFrame columns as zeros (GH1696)
- Fix bug in scatter_plot with by option (GH1716)
- Fix performance problem in infer_freq with lots of non-unique stamps (GH1686)
- Fix handling of PeriodIndex as argument to create MultiIndex (GH1705)
- Fix re: unicode MultiIndex level names in Series/DataFrame repr (GH1736)
- Handle PeriodIndex in to_datetime instance method (GH1703)
- Support StaticTzInfo in DatetimeIndex infrastructure (GH1692)
- Allow MultiIndex setops with length-0 other type indexes (GH1727)
- Fix handling of DatetimeIndex in DataFrame.to_records (GH1720)
- Fix handling of general objects in isnull on which bool(...) fails (GH1749)
- Fix .ix indexing with MultiIndex ambiguity (GH1678)
- Fix .ix setting logic error with non-unique MultiIndex (GH1750)
- Basic indexing now works on MultiIndex with > 1000000 elements, regression from earlier version of pandas (GH1757)
- Handle non-float64 dtypes in fast DataFrame.corr/cov code paths (GH1761)
- Fix DatetimeIndex.isin to function properly (GH1763)
- Fix conversion of array of tz-aware datetime.datetime to DatetimeIndex with right time zone (GH1777)
- Fix DST issues with generating anchxored date ranges (GH1778)
- Fix issue calling sort on result of Series.unique (GH1807)
- Fix numerical issue leading to square root of negative number in rolling_std (GH1840)
- Let Series.str.split accept no arguments (like str.split) (GH1859)
- Allow user to have dateutil 2.1 installed on a Python 2 system (GH1851)
- Catch ImportError less aggressively in pandas/__init__.py (GH1845)
- Fix pip source installation bug when installing from GitHub (GH1805)
- Fix error when window size > array size in rolling_apply (GH1850)
• Fix pip source installation issues via SSH from GitHub
• Fix OLS.summary when column is a tuple (GH1837)
• Fix bug in __doc__ patching when -OO passed to interpreter (GH1792 GH1741 GH1774)
• Fix unicode console encoding issue in IPython notebook (GH1782, GH1768)
• Fix unicode formatting issue with Series.name (GH1782)
• Fix bug in DataFrame.duplicated with datetime64 columns (GH1833)
• Fix bug in Panel internals resulting in error when doing fillna after truncate not changing size of panel (GH1823)
• Prevent segfault due to MultiIndex not being supported in HDFStore table format (GH1848)
• Fix UnboundLocalError in Panel.__setitem__ and add better error (GH1826)
• Fix to_csv issues with list of string entries. Isnull works on list of strings now too (GH1791)
• Fix Timestamp comparisons with datetime values outside the nanosecond range (1677-2262)
• Revert to prior behavior of normalize_date with datetime.date objects (return datetime)
• Fix broken interaction between np.nansum and Series.any/all
• Fix bug with multiple column date parsers (GH1866)
• DatetimeIndex.union(Int64Index) was broken
• Make plot x vs y interface consistent with integer indexing (GH1842)
• set_index inplace modified data even if unique check fails (GH1831)
• Only use Q-OCT/NOV/DEC in quarterly frequency inference (GH1789)
• Upcast to dtype=object when unstacking boolean DataFrame (GH1820)
• Fix float64/float32 merging bug (GH1849)
• Fixes to Period.start_time for non-daily frequencies (GH1857)
• Fix failure when converter used on index_col in read_csv (GH1835)
• Implement PeriodIndex.append so that pandas.concat works correctly (GH1815)
• Avoid Cython out-of-bounds access causing segfault sometimes in pad_2d, backfill_2d
• Fix resampling error with intraday times and anchored target time (like AS-DEC) (GH1772)
• Fix .ix indexing bugs with mixed-integer indexes (GH1799)
• Respect passed color keyword argument in Series.plot (GH1890)
• Fix rolling_min/max when the window is larger than the size of the input array. Check other malformed inputs (GH1899, GH1897)
• Rolling variance / standard deviation with only a single observation in window (GH1884)
• Fix unicode sheet name failure in to_excel (GH1828)
• Override DatetimeIndex.min/max to return Timestamp objects (GH1895)
• Fix column name formatting issue in length-truncated column (GH1906)
• Fix broken handling of copying Index metadata to new instances created by view(...) calls inside the NumPy infrastructure
• Support datetime.date again in DateOffset.rollback/rollforward
• Raise Exception if set passed to Series constructor (GH1913)
• Add TypeError when appending HDFStore table w/ wrong index type (GH1881)
• Don’t raise exception on empty inputs in EW functions (e.g. ewma) (GH1900)
• Make asof work correctly with PeriodIndex (GH1883)
• Fix extlinks in doc build
• Fill boolean DataFrame with NaN when calling shift (GH1814)
• Fix setuptools bug causing pip not to Cythonize .pyx files sometimes
• Fix negative integer indexing regression in .ix from 0.7.x (GH1888)
• Fix error while retrieving timezone and utc offset from subclasses of datetime.tzinfo without .zone and ._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)

30.9 pandas 0.8.1

Release date: July 22, 2012

30.9.1 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)

30.9.2 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)

30.9.3 Bug Fixes

• Fix NA handling in DataFrame.to_panel (GH1582)
• Handle TypeError issues inside PyObject_RichCompareBool calls in khash (GH1318)
• Fix resampling bug to lower case daily frequency (GH1588)
• Fix kendall/spearman DataFrame.corr bug with no overlap (GH1595)
• Fix bug in DataFrame.set_index (GH1592)
• Don’t ignore axes in boxplot if by specified (GH1565)
• Fix Panel .ix indexing with integers bug (GH1603)
• Fix Partial indexing bugs (years, months, ...) with PeriodIndex (GH1601)
• Fix MultiIndex console formatting issue (GH1606)
• Unordered index with duplicates doesn’t yield scalar location for single entry (GH1586)
• Fix resampling of tz-aware time series with “anchored” freq (GH1591)
• Fix DataFrame.rank error on integer data (GH1589)
• Selection of multiple SparseDataFrame columns by list in __getitem__ (GH1585)
• Override Index.tolist for compatibility with MultiIndex (GH1576)
• Fix hierarchical summing bug with MultiIndex of length 1 (GH1568)
• Work around numpy.concatenate use/bug in Series.set_value (GH1561)
• Ensure Series/DataFrame are sorted before resampling (GH1580)
• Fix unhandled IndexError when indexing very large time series (GH1562)
• Fix DatetimeIndex intersection logic error with irregular indexes (GH1551)
• Fix unit test errors on Python 3 (GH1550)
• Fix .ix indexing bugs in duplicate DataFrame index (GH1201)
• Better handle errors with non-existing objects in HDFStore (GH1254)
• Don’t copy int64 array data in DatetimeIndex when copy=False (GH1624)
• Fix resampling of conforming periods quarterly to annual (GH1622)
• Don’t lose index name on resampling (GH1631)
• Support python-dateutil version 2.1 (GH1637)
• Fix broken scatter_matrix axis labeling, esp. with time series (GH1625)
• Fix cases where extra keywords weren’t being passed on to matplotlib from Series.plot (GH1636)
• Fix BusinessMonthBegin logic for dates before 1st bday of month (GH1645)
• Ensure string alias converted (valid in DatetimeIndex.get_loc) in DataFrame.xs / __getitem__ (GH1644)
• Fix use of string alias timestamps with tz-aware time series (GH1647)
• Fix Series.max/min and Series.describe on len-0 series (GH1650)
• Handle None values in dict passed to concat (GH1649)
• Fix Series.interpolate with method='values' and DatetimeIndex (GH1646)
• Fix IndexError in left merges on a DataFrame with 0-length (GH1628)
• Fix DataFrame column width display with UTF-8 encoded characters (GH1620)
• Handle case in pandas.io.data.get_data_yahoo where Yahoo! returns duplicate dates for most recent business day
• Avoid downsampling when plotting mixed frequencies on the same subplot (GH1619)
• Fix read_csv bug when reading a single line (GH1553)
• Fix bug in C code causing monthly periods prior to December 1969 to be off (GH1570)

30.10 pandas 0.8.0

Release date: 6/29/2012

30.10.1 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 type (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

30.10.2 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.rename to work with arrays of timestamp inputs
- Cython implementation of DataFrame.corr speeds up by > 100x (GH1349, GH1354)
- Exclude “nuisance” columns automatically in GroupBy.transform (GH1364)
- Support functions-as-strings in GroupBy.transform (GH1362)
- Use index name as xlabel/ylabel in plots (GH1415)
- Add convert_dtype option to Series.apply to be able to leave data as dtype=object (GH1414)
- Can specify all index level names in concat (GH1419)
- Add dialect keyword to parsers for quoting conventions (GH1363)
- Enable DataFrame[bool_DataFrame] += value (GH1366)
- Add retries argument to get_data_yahoo to try to prevent Yahoo! API 404s (GH826)
- Improve performance of reshaping by using O(N) categorical sorting
- Series names will be used for index of DataFrame if no index passed (GH1494)
- Header argument in DataFrame.to_csv can accept a list of column names to use instead of the object’s columns (GH921)
- Add raise_conflict argument to DataFrame.update (GH1526)
- Support file-like objects in ExcelFile (GH1529)

30.10.3 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)

30.10.4 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)

30.11 pandas 0.7.3

Release date:  April 12, 2012

30.11.1 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)

30.11.2 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)

30.11.3 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 gourby results (GH995)
• DataFrame.from_records no longer mutate input columns (GH975)
• Use Index name when grouping by it (GH1313)

30.12 pandas 0.7.2

Release date: March 16, 2012

30.12.1 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

30.12.2 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)

30.12.3 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)

30.12.4 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 inconsistent 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)

30.13 pandas 0.7.1

Release date: February 29, 2012

30.13.1 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

30.13.2 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)
30.13.3 Bug Fixes

- Fix memory leak when inserting large number of columns into a single DataFrame (GH790)
- Appending length-0 DataFrame with new columns would not result in those new columns being part of the resulting concatenated DataFrame (GH782)
- Fixed groupby corner case when passing dictionary grouper and as_index is False (GH819)
- Fixed bug whereby bool array sometimes had object dtype (GH820)
- Fix exception thrown on np.diff (GH816)
- Fix to_records where columns are non-strings (GH822)
- Fix Index.intersection where indices have incomparable types (GH811)
- Fix ExcelFile throwing an exception for two-line file (GH837)
- Add clearer error message in csv parser (GH835)
- Fix loss of fractional seconds in HDFStore (GH513)
- Fix DataFrame join where columns have datetimes (GH787)
- Work around numpy performance issue in take (GH817)
- Improve comparison operations for NA-friendliness (GH801)
- Fix indexing operation for floating point values (GH780, GH798)
- Fix groupby case resulting in malformed dataframe (GH814)
- Fix behavior of reindex of Series dropping name (GH812)
- Improve on redundant groupby computation (GH775)
- Catch possible NA assignment to int/bool series with exception (GH839)

30.14 pandas 0.7.0

Release date: 2/9/2012

30.14.1 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
30.14.2 API Changes

- Label-indexing with integer indexes now raises KeyError if a label is not found instead of falling back on location-based indexing (GH700)
- Label-based slicing via `ix` or `[]` on Series will now only work if exact matches for the labels are found or if the index is monotonic (for range selections)
- Label-based slicing and sequences of labels can be passed to `[]` on a Series for both getting and setting (GH86)
- `[]` operator (`__getitem__` and `__setitem__`) will raise KeyError with integer indexes when an index is not contained in the index. The prior behavior would fall back on position-based indexing if a key was not found in the index which would lead to subtle bugs. This is now consistent with the behavior of `.ix` on DataFrame and friends (GH328)
- Rename `DataFrame.delevel` to `DataFrame.reset_index` and add deprecation warning
- `Series.sort` (an in-place operation) called on a Series which is a view on a larger array (e.g. a column in a DataFrame) will generate an Exception to prevent accidentally modifying the data source (GH316)
- Refactor to remove deprecated LongPanel class (GH552)
- Deprecated `Panel.to_long`, renamed to `to_frame`
- Deprecated `colSpace` argument in `DataFrame.to_string`, renamed to `col_space`
- Rename `precision` to `accuracy` in engineering float formatter (GH 395)
- The default delimiter for `read_csv` is comma rather than letting `csv.Sniffer` infer it
- Rename `col_or_columns` argument in `DataFrame.drop_duplicates` (GH 734)

30.14.3 Improvements to existing features

- Better error message in DataFrame constructor when passed column labels don’t match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don’t print length by default in `Series.to_string`, add `length` option (GH 489)
- 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)
• 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(iclo) 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)

30.14.4 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 (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) (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)

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

30.15 pandas 0.6.1

Release date: 12/13/2011
### 30.15.1 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)

### 30.15.2 New features

- Can pass Series to DataFrame.append with ignore_index=True for appending a single row (GH430)
- Add Spearman and Kendall correlation options to Series.corr and DataFrame.corr (GH428)
- Add new `get_value` and `set_value` methods to Series, DataFrame, and Panel to very low-overhead access to scalar elements. `df.get_value(row, column)` is about 3x faster than `df[column][row]` by handling fewer cases (GH437, GH438). Add similar methods to sparse data structures for compatibility.
- Add Qt table widget to sandbox (GH435)
- DataFrame.align can accept Series arguments, add axis keyword (GH461)
- Implement new SparseList and SparseArray data structures. SparseSeries now derives from SparseArray (GH463)
- `max_columns / max_rows` options in set_printoptions (GH453)
- Implement Series.rank and DataFrame.rank, fast versions of scipy.stats.rankdata (GH428)
- Implement DataFrame.from_items alternate constructor (GH444)
- DataFrame.convert_objects method for inferring better dtypes for object columns (GH302)
- Add `rolling_corr_pairwise` function for computing Panel of correlation matrices (GH189)
- Add `margins` option to `pivot_table` for computing subgroup aggregates (GH GH114)
- Add `Series.from_csv` function (GH482)

### 30.15.3 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

30.15.4 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)

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

30.16 pandas 0.6.0

Release date: 11/25/2011

30.16.1 API Changes

• Arithmetic methods like \texttt{sum} will attempt to sum \texttt{dtype=object} values by default instead of excluding them (GH382)

30.16.2 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 \texttt{Series.mad}, mean absolute deviation, matching DataFrame
• Add \texttt{QuarterEnd} DateOffset (GH321)
• Add matrix multiplication function \texttt{dot} to DataFrame (GH65)
• Add \texttt{orient} option to \texttt{Panel.from_dict} to ease creation of mixed-type Panels (GH359, GH301)
• Add \texttt{DataFrame.from_dict} with similar \texttt{orient} option
• Can now pass list of tuples or list of lists to \texttt{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)

30.16.3 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

### 30.16.4 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)
• Fixed `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 \texttt{asfreq} (GH390)
• Pass level names to \texttt{swaplavel} (GH379)
• Don’t lose index names in \texttt{MultiIndex.droplevel} (GH394)
• Infer more proper return type in \texttt{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 \texttt{.ix} / advanced indexing (GH397)
• Handle mixed-type DataFrames correctly in unstack, do not lose type information (GH403)
• Fix integer name formatting bug in \texttt{Index.format} and in \texttt{Series.__repr__}
• Handle label types other than string passed to groupby (GH405)
• Fix bug in \texttt{.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 \texttt{GroupBy.apply} (GH416)

\textbf{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
• Ted Square
• Aman Thakral
• Chris Uga
• Dieter Vandenbussche
• carljv
• rsamson
### 30.17 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 as attributes (a new feature).

In addition to the changes listed here from 0.4.3 to 0.5.0, the minor releases 0.4.1, 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.

#### 30.17.1 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`

• Other outstanding deprecations have been 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

### 30.17.2 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

30.17.3 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)

30.17.4 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
pandas: powerful Python data analysis toolkit, Release 0.13.1

- 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

Thanks
- Thomas Kluyver
- Daniel Fortunov
- Aman Thakral
- Luca Beltrame
- Wouter Overmeire

30.18 pandas 0.4.3

Release date: 10/9/2011
This is largely a bugfix release from 0.4.2 but also includes a handful of new and enhanced features. Also, pandas can now be installed and used on Python 3 (thanks Thomas Kluyver!).
30.18.1 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)

30.18.2 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

30.18.3 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`

30.18.4 Bug Fixes

- Fix broken interaction between `Index` and `Int64Index` when calling intersection. Implement `Int64Index.intersection`
- `MultiIndex.sortlevel` discarded the level names (GH202)
- Fix bugs in groupby, join, and append due to improper concatenation of `MultiIndex` objects (GH201)
- Fix regression from 0.4.1, `isnull` and `notnull` ceased to work on other kinds of Python scalar objects like `datetime.datetime`
- Raise more helpful exception when attempting to write empty DataFrame or LongPanel to `HDFStore` (GH204)
- Use stdlib csv module to properly escape strings with commas in `DataFrame.to_csv` (GH206, Thomas Kluyver)
- Fix Python ndarray access in Cython code for sparse blocked index integrity check
- Fix bug writing Series to CSV in Python 3 (GH209)
- Miscellaneous Python 3 bugfixes

Thanks

- Thomas Kluyver
- rsamson
30.19 pandas 0.4.2

Release date: 10/3/2011

This is a performance optimization release with several bug fixes. The new Int64Index and new merging / joining Cython code and related Python infrastructure are the main new additions

30.19.1 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

30.19.2 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

30.19.3 API Changes

None
30.19.4 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

Thanks
- Uri Laserson
- Scott Sinclair

30.20 pandas 0.4.1

Release date: 9/25/2011

This is primarily a bug fix release but includes some new features and improvements

30.20.1 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

30.20.2 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
30.20.3 API Changes

None

30.20.4 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)
- stack 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

Thanks

- Yaroslav Halchenko
- Jeff Reback
- Skipper Seabold
- Dan Lovell
- Nick Pentreath

30.21 pandas 0.4.0

Release date: 9/12/2011

30.21.1 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__`, `__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

30.21.2 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).

30.21.3 API Changes

• The `DataMatrix` variable now refers to `DataFrame`, will be removed within two releases
• *WidePanel* is now known as *Panel*. The *WidePanel* variable in the pandas namespace now refers to the renamed *Panel* class

• *LongPanel* and *Panel / WidePanel* now no longer have a common subclass. *LongPanel* is now a subclass of *DataFrame* having a number of additional methods and a hierarchical index instead of the old *LongPanelIndex* object, which has been removed. Legacy *LongPanel* pickles may not load properly

• Cython is now required to build *pandas* from a development branch. This was done to avoid continuing to check in cythonized C files into source control. Builds from released source distributions will not require Cython

• Cython code has been moved up to a top level *pandas/src* directory. Cython extension modules have been renamed and promoted from the *lib* subpackage to the top level, i.e.

  - `pandas.lib.tseries` -> `pandas._tseries`
  - `pandas.lib.sparse` -> `pandas._sparse`

• *DataFrame* pickling format has changed. Backwards compatibility for legacy pickles is provided, but it’s recommended to consider PyTables-based *HDFStore* for storing data with a longer expected shelf life

• A copy argument has been added to the *DataFrame* constructor to avoid unnecessary copying of data. Data is no longer copied by default when passed into the constructor

• Handling of boolean dtype in *DataFrame* has been improved to support storage of boolean data with NA / NaN values. Before it was being converted to float64 so this should not (in theory) cause API breakage

• To optimize performance, Index objects now only check that their labels are unique when uniqueness matters (i.e. when someone goes to perform a lookup). This is a potentially dangerous tradeoff, but will lead to much better performance in many places (like *groupby*).

• Boolean indexing using Series must now have the same indices (labels)

• Backwards compatibility support for begin/end/nPeriods keyword arguments in *DateRange* class has been removed

• More intuitive / shorter filling aliases *ffill* (for *pad*) and *bfill* (for *backfill*) have been added to the functions that use them: *reindex, asfreq, fillna*.

• *pandas.core.mixins* code moved to *pandas.core.generic*

• *buffer* keyword arguments (e.g. *DataFrame.toString*) renamed to *buf* to avoid using Python built-in name

• *DataFrame.rows()* removed (use *DataFrame.index*)

• Added deprecation warning to *DataFrame.cols()* (to be removed in next release)

• *DataFrame* deprecations and de-camelCasing: *merge, asMatrix, toDataMatrix, _firstTimeWithValue, _lastTimeWithValue, toRecords, fromRecords, tggroupby, 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
pandas: powerful Python data analysis toolkit, Release 0.13.1

- `Series.combineFunc` renamed to `Series.combine` and made a bit more general with a `fill_value` keyword argument defaulting to NaN
- Removed `pandas.core.pytools` module. Code has been moved to `pandas.core.common`
- Tacked on `groupName` attribute for groups in GroupBy renamed to `name`
- Panel/LongPanel `dims` attribute renamed to `shape` to be more conformant
- Slicing a `Series` returns a view now
- More Series deprecations / renaming: `toCSV` to `to_csv`, `asOf` to `asof`, `merge` to `map`, `applymap` to `apply`, `toDict` to `to_dict`, `combineFirst` to `combine_first`. Will print `FutureWarning`.
- `DataFrame.to_csv` does not write an “index” column label by default anymore since the output file can be read back without it. However, there is a new `index_label` argument. So you can do `index_label='index'` to emulate the old behavior
- `datetools.Week` argument renamed from `dayOfWeek` to `weekday`
- `timeRule` argument in `shift` has been deprecated in favor of using the `offset` argument for everything. So you can still pass a time rule string to `offset`
- Added optional `encoding` argument to `read_csv`, `read_table`, `to_csv`, `from_csv` to handle unicode in python 2.x

30.21.4 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

Thanks

- Joon Ro
- Michael Pennington
30.22 pandas 0.3.0

Release date: February 20, 2011

30.22.1 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
30.22.2 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.)

30.22.3 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

30.22.4 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