
pandas: powerful Python data analysis toolkit

Release 0.13.0

Wes McKinney & PyData Development Team

January 03, 2014

CONTENTS

1	What's New	3
1.1	v0.13.0 (January 3, 2014)	3
1.2	v0.12.0 (July 24, 2013)	25
1.3	v0.11.0 (April 22, 2013)	36
1.4	v0.10.1 (January 22, 2013)	45
1.5	v0.10.0 (December 17, 2012)	50
1.6	v0.9.1 (November 14, 2012)	61
1.7	v0.9.0 (October 7, 2012)	65
1.8	v0.8.1 (July 22, 2012)	66
1.9	v0.8.0 (June 29, 2012)	67
1.10	v0.7.3 (April 12, 2012)	73
1.11	v0.7.2 (March 16, 2012)	76
1.12	v0.7.1 (February 29, 2012)	77
1.13	v0.7.0 (February 9, 2012)	78
1.14	v0.6.1 (December 13, 2011)	83
1.15	v0.6.0 (November 25, 2011)	84
1.16	v0.5.0 (October 24, 2011)	85
1.17	v0.4.3 through v0.4.1 (September 25 - October 9, 2011)	86
2	Installation	89
2.1	Python version support	89
2.2	Binary installers	89
2.3	Dependencies	90
2.4	Recommended Dependencies	90
2.5	Optional Dependencies	90
2.6	Installing from source	92
2.7	Running the test suite	92
3	Frequently Asked Questions (FAQ)	93
3.1	How do I control the way my DataFrame is displayed?	93
3.2	Adding Features to your Pandas Installation	93
3.3	Migrating from scikits.timeseries to pandas >= 0.8.0	94
3.4	Byte-Ordering Issues	98
3.5	Visualizing Data in Qt applications	99
4	Package overview	101
4.1	Data structures at a glance	101
4.2	Mutability and copying of data	102
4.3	Getting Support	102

4.4	Credits	102
4.5	Development Team	102
4.6	License	102
5	10 Minutes to Pandas	105
5.1	Object Creation	105
5.2	Viewing Data	106
5.3	Selection	108
5.4	Missing Data	113
5.5	Operations	113
5.6	Merge	116
5.7	Grouping	118
5.8	Reshaping	119
5.9	Time Series	121
5.10	Plotting	123
5.11	Getting Data In/Out	124
5.12	Gotchas	126
6	Cookbook	127
6.1	Idioms	127
6.2	Selection	127
6.3	MultiIndexing	127
6.4	Missing Data	128
6.5	Grouping	129
6.6	Timeseries	130
6.7	Merge	130
6.8	Plotting	130
6.9	Data In/Out	131
6.10	Computation	132
6.11	Miscellaneous	132
6.12	Aliasing Axis Names	133
7	Intro to Data Structures	135
7.1	Series	135
7.2	DataFrame	140
7.3	Panel	152
7.4	Panel4D (Experimental)	157
7.5	PanelND (Experimental)	159
8	Essential Basic Functionality	161
8.1	Head and Tail	161
8.2	Attributes and the raw ndarray(s)	162
8.3	Accelerated operations	163
8.4	Flexible binary operations	163
8.5	Descriptive statistics	169
8.6	Function application	176
8.7	Reindexing and altering labels	179
8.8	Iteration	186
8.9	Vectorized string methods	188
8.10	Sorting by index and value	192
8.11	Copying	194
8.12	dtypes	194
8.13	Working with package options	201
8.14	Console Output Formatting	205

9	Indexing and Selecting Data	207
9.1	Different Choices for Indexing (<code>loc</code> , <code>iloc</code> , and <code>ix</code>)	207
9.2	Deprecations	208
9.3	Basics	208
9.4	Attribute Access	210
9.5	Slicing ranges	211
9.6	Selection By Label	213
9.7	Selection By Position	215
9.8	Setting With Enlargement	217
9.9	Fast scalar value getting and setting	219
9.10	Boolean indexing	219
9.11	The <code>where()</code> Method and Masking	222
9.12	The <code>query()</code> Method (Experimental)	225
9.13	Take Methods	236
9.14	Duplicate Data	238
9.15	Dictionary-like <code>get()</code> method	239
9.16	Advanced Indexing with <code>.ix</code>	239
9.17	The <code>select()</code> Method	242
9.18	The <code>lookup()</code> Method	242
9.19	<code>Float64Index</code>	242
9.20	Returning a view versus a copy	245
9.21	Fallback indexing	247
9.22	Index objects	248
9.23	Hierarchical indexing (<code>MultiIndex</code>)	249
9.24	Setting index metadata (<code>name(s)</code> , <code>levels</code> , <code>labels</code>)	261
9.25	Adding an index to an existing <code>DataFrame</code>	261
9.26	Add an index using <code>DataFrame</code> columns	261
9.27	Remove / reset the index, <code>reset_index</code>	262
9.28	Adding an ad hoc index	263
9.29	Indexing internal details	264
10	Computational tools	265
10.1	Statistical functions	265
10.2	Moving (rolling) statistics / moments	269
10.3	Expanding window moment functions	275
10.4	Exponentially weighted moment functions	277
11	Working with missing data	279
11.1	Missing data basics	279
11.2	Datetimes	281
11.3	Calculations with missing data	281
11.4	Cleaning / filling missing data	283
11.5	Missing data casting rules and indexing	296
12	Group By: split-apply-combine	299
12.1	Splitting an object into groups	300
12.2	Iterating through groups	304
12.3	Aggregation	305
12.4	Transformation	308
12.5	Filtration	312
12.6	Dispatching to instance methods	313
12.7	Flexible <code>apply</code>	314
12.8	Other useful features	315
13	Merge, join, and concatenate	319

13.1	Concatenating objects	319
13.2	Database-style DataFrame joining/merging	328
14	Reshaping and Pivot Tables	339
14.1	Reshaping by pivoting DataFrame objects	339
14.2	Reshaping by stacking and unstacking	340
14.3	Reshaping by Melt	344
14.4	Combining with stats and GroupBy	345
14.5	Pivot tables and cross-tabulations	347
14.6	Tiling	350
14.7	Computing indicator / dummy variables	351
14.8	Factorizing values	352
15	Time Series / Date functionality	353
15.1	Time Stamps vs. Time Spans	354
15.2	Converting to Timestamps	355
15.3	Generating Ranges of Timestamps	356
15.4	DatetimeIndex	358
15.5	DateOffset objects	364
15.6	Time series-related instance methods	369
15.7	Up- and downsampling	371
15.8	Time Span Representation	373
15.9	Converting between Representations	376
15.10	Time Zone Handling	377
15.11	Time Deltas	380
15.12	Time Deltas & Reductions	383
15.13	Time Deltas & Conversions	384
16	Plotting with matplotlib	387
16.1	Basic plotting: <code>plot</code>	387
16.2	Other plotting features	399
17	Trellis plotting interface	419
17.1	Examples	419
17.2	Scales	426
18	IO Tools (Text, CSV, HDF5, ...)	429
18.1	CSV & Text files	430
18.2	JSON	450
18.3	HTML	458
18.4	Excel files	467
18.5	Clipboard	468
18.6	Pickling	469
18.7	msgpack (experimental)	470
18.8	HDF5 (PyTables)	473
18.9	SQL Queries	496
18.10	Google BigQuery (Experimental)	497
18.11	STATA Format	498
19	Remote Data Access	501
19.1	Yahoo! Finance	501
19.2	Google Finance	502
19.3	FRED	502
19.4	Fama/French	503
19.5	World Bank	503

20	Enhancing Performance	507
20.1	Cython (Writing C extensions for pandas)	507
20.2	Expression Evaluation via <code>eval()</code> (Experimental)	511
21	Sparse data structures	519
21.1	SparseArray	521
21.2	SparseList	521
21.3	SparseIndex objects	522
22	Caveats and Gotchas	523
22.1	Using If/Truth Statements with Pandas	523
22.2	NaN, Integer NA values and NA type promotions	524
22.3	Integer indexing	526
22.4	Label-based slicing conventions	526
22.5	Miscellaneous indexing gotchas	527
22.6	Timestamp limitations	529
22.7	Parsing Dates from Text Files	529
22.8	Differences with NumPy	530
22.9	Thread-safety	530
22.10	HTML Table Parsing	530
22.11	Byte-Ordering Issues	531
23	rpy2 / R interface	533
23.1	Transferring R data sets into Python	533
23.2	Converting DataFrames into R objects	534
23.3	Calling R functions with pandas objects	534
23.4	High-level interface to R estimators	534
24	Related Python libraries	535
24.1	la (larry)	535
24.2	statsmodels	535
24.3	scikits.timeseries	535
25	Comparison with R / R libraries	537
25.1	Base R	537
25.2	zoo	539
25.3	xts	539
25.4	plyr	539
25.5	reshape / reshape2	540
26	Comparison with SQL	543
26.1	SELECT	543
26.2	WHERE	544
26.3	GROUP BY	546
26.4	JOIN	548
26.5	UNION	550
26.6	UPDATE	551
26.7	DELETE	551
27	API Reference	553
27.1	Input/Output	553
27.2	General functions	576
27.3	Series	604
27.4	DataFrame	710
27.5	Panel	855

27.6	Index	925
27.7	DatetimeIndex	965
27.8	GroupBy	1005
28	Release Notes	1045
28.1	pandas 0.13.0	1045
28.2	pandas 0.12.0	1059
28.3	pandas 0.11.0	1066
28.4	pandas 0.10.1	1072
28.5	pandas 0.10.0	1074
28.6	pandas 0.9.1	1079
28.7	pandas 0.9.0	1081
28.8	pandas 0.8.1	1086
28.9	pandas 0.8.0	1089
28.10	pandas 0.7.3	1093
28.11	pandas 0.7.2	1094
28.12	pandas 0.7.1	1096
28.13	pandas 0.7.0	1097
28.14	pandas 0.6.1	1103
28.15	pandas 0.6.0	1106
28.16	pandas 0.5.0	1109
28.17	pandas 0.4.3	1113
28.18	pandas 0.4.2	1114
28.19	pandas 0.4.1	1115
28.20	pandas 0.4.0	1117
28.21	pandas 0.3.0	1122
	Python Module Index	1125

PDF Version

Zipped HTML **Date:** January 03, 2014 **Version:** 0.13.0

Binary Installers: <http://pypi.python.org/pypi/pandas>

Source Repository: <http://github.com/pydata/pandas>

Issues & Ideas: <https://github.com/pydata/pandas/issues>

Q&A Support: <http://stackoverflow.com/questions/tagged/pandas>

Developer Mailing List: <http://groups.google.com/group/pystatsmodels>

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 v0.13.0 (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](#)*

1.1.1 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 @jtratrner. 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()
True

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

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

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

```
      A  B
0    11  1
1   bbb  2
2    cc  3
[3 rows x 2 columns]
```

- **Panel.reindex** has the following call signature `Panel.reindex(items=None, major_axis=None, minor_axis=None)` to conform with other NDFrame objects. See *Internal Refactoring* for more information.
- **Series.argmax** and **Series.argmin** are now aliased to **Series.idxmax** and **Series.idxmin**. These return the index of the min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element. (GH6214)

1.1.2 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)

1.1.3 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`.

1.1.4 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
```

```
0    1
1    2
2    3
dtype: int64
```

```
In [12]: s[5] = 5.
```

```
In [13]: s
```

```
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
```

```
   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
```

```
   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
```

```
   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
```

```
<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
```

```
<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']
```

```
           Item1  Item2
2001-01-12      30     32
2001-01-13      30     32
2001-01-14      30     32
2001-01-15      30     32
[4 rows x 2 columns]
```

1.1.5 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
Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='object')
```

```
In [27]: s = Series(range(5), index=index)
```

```
In [28]: s
```

```
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]
2
```

```
In [30]: s.ix[3]
2
```

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

The only positional indexing is via `iloc`

```
In [32]: s.iloc[3]
3
```

A scalar index that is not found will raise `KeyError`

Slicing is ALWAYS on the values of the index, for `[]`, `ix`, `loc` and ALWAYS positional with `iloc`

```
In [33]: s[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [34]: s.ix[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [35]: s.loc[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [36]: s.iloc[2:4]
```

```
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats are allowed

```
In [37]: s[2.1:4.6]
```

```
3.0    2
4.5    3
dtype: int64
```

```
In [38]: s.loc[2.1:4.6]
```

```
3.0    2
4.5    3
dtype: int64
```

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

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

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

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

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

1.1.6 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']")
.....:
```

	A	B
2013-01-05	0.049355	0.632633
2013-01-06	-0.700262	-0.237922
2013-01-07	-0.419578	-0.159861
2013-01-08	1.144951	-0.726740
2013-01-09	0.309259	-0.328860
2013-01-10	1.785587	1.135875

[6 rows x 2 columns]

Use an inline column reference

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

	A	B	C	D
2013-01-01	0.713216	-0.778461	-0.661062	0.862877
2013-01-02	0.344342	0.149565	-0.626968	-0.875772
2013-01-03	-0.930687	-0.218983	0.949965	-0.442354
2013-01-05	0.049355	0.632633	-1.502767	-1.225492
2013-01-06	-0.700262	-0.237922	0.803216	0.323302
2013-01-07	-0.419578	-0.159861	0.286230	-0.311358
2013-01-08	1.144951	-0.726740	-0.178315	0.386127
2013-01-09	0.309259	-0.328860	0.056344	1.495435
2013-01-10	1.785587	1.135875	0.699592	0.245373

[9 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          (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

<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

<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
```

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

```
In [60]: store2.close()
```

```
In [61]: store2
```

```
<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

1.1.7 DataFrame repr Changes

The HTML and plain text representations of `DataFrame` now show a truncated view of the table once it exceeds a certain size, rather than switching to the short info view ([GH4886](#), [GH5550](#)). This makes the representation more consistent as small `DataFrames` get larger.

2010-03-29	13.70	13.88	13.39	13.57	158225000	12.98
2010-03-30	13.55	13.64	13.18	13.28	142055200	12.70
	...	...	...	...	...	...

771 rows × 6 columns

To get the info view, call `DataFrame.info()`. If you prefer the info view as the repr for large `DataFrames`, you can set this by running `set_option('display.large_repr', 'info')`.

1.1.8 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 handling 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])
```

```

      1  2
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)
```

```

      1  2  NaN
0  1  0  0
1  0  1  0
2  0  0  1
[3 rows x 3 columns]
```

- `timedelta64[ns]` operations. See [the docs](#).

Warning: Most of these operations require `numpy >= 1.7`

Using the new top-level `to_timedelta`, you can convert a scalar or array from the standard `timedelta` format (produced by `to_csv`) into a `timedelta` type (`np.timedelta64` in nanoseconds).

```
In [64]: to_timedelta('1 days 06:05:01.00003')
numpy.timedelta64(108301000030000,'ns')
```

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

```
In [66]: to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
```

```

0    1 days, 06:05:01.000030
1              00:00:00.000016
2                      NaT
dtype: timedelta64[ns]
```

```
In [67]: to_timedelta(np.arange(5), unit='s')
```

```

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')
```

```
0      00:00:00
1    1 days, 00:00:00
2    2 days, 00:00:00
3    3 days, 00:00:00
4    4 days, 00:00:00
dtype: timedelta64[ns]
```

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

```
In [69]: from datetime import timedelta
```

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

```
In [71]: td[2] += np.timedelta64(timedelta(minutes=5, seconds=3))
```

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

```
In [73]: td
```

```
0    31 days, 00:00:00
1    31 days, 00:00:00
2    31 days, 00:05:03
3              NaT
dtype: timedelta64[ns]
```

```
# to days
```

```
In [74]: td / np.timedelta64(1, 'D')
```

```
0     31.000000
1     31.000000
2     31.003507
3           NaN
dtype: float64
```

```
In [75]: td.astype('timedelta64[D]')
```

```
0     31
1     31
2     31
3    NaN
dtype: float64
```

```
# to seconds
```

```
In [76]: td / np.timedelta64(1, 's')
```

```
0    2678400
1    2678400
2    2678703
3           NaN
```

```
dtype: float64
```

```
In [77]: td.astype('timedelta64[s]')
```

```
0    2678400
1    2678400
2    2678703
3         NaN
dtype: float64
```

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

```
In [78]: td * -1
```

```
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])
```

```
0    31 days, 00:00:00
1    62 days, 00:00:00
2    93 days, 00:15:09
3              NaT
dtype: timedelta64[ns]
```

Absolute `DateOffset` objects can act equivalently to `timedeltas`

```
In [80]: from pandas import offsets
```

```
In [81]: td + offsets.Minute(5) + offsets.Milli(5)
```

```
0    31 days, 00:05:00.005000
1    31 days, 00:05:00.005000
2    31 days, 00:10:03.005000
3              NaT
dtype: timedelta64[ns]
```

`fillna` is now supported for `timedeltas`

```
In [82]: td.fillna(0)
```

```
0    31 days, 00:00:00
1    31 days, 00:00:00
2    31 days, 00:05:03
3         00:00:00
dtype: timedelta64[ns]
```

```
In [83]: td.fillna(timedelta(days=1,seconds=5))
```

```
0    31 days, 00:00:00
1    31 days, 00:00:00
2    31 days, 00:05:03
3     1 days, 00:00:05
dtype: timedelta64[ns]
```

You can do numeric reduction operations on `timedeltas`.

```
In [84]: td.mean()
```

```
0    31 days, 00:01:41
dtype: timedelta64[ns]
```

```
In [85]: td.quantile(.1)
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.

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

```
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.

```
In [87]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
```

```
      0      1
0     a      1
1     b      2
2    NaN    NaN
[3 rows x 2 columns]
```

Elements that do not match return a row of `NaN`. Thus, a `Series` of messy strings can be *converted* into a like-indexed `Series` or `DataFrame` of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects.

Named groups like

```
In [88]: Series(['a1', 'b2', 'c3']).str.extract(
.....:         ' (?P<letter>[ab]) (?P<digit>\d) '
.....: )
```

```
      letter digit
0         a      1
1         b      2
2        NaN    NaN
[3 rows x 2 columns]
```

and optional groups can also be used.

```
In [89]: Series(['a1', 'b2', '3']).str.extract(
.....:         ' (?P<letter>[ab])? (?P<digit>\d) '
.....: )
```

```
      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.

```
In [90]: date_range('2013-01-01', periods=5, freq='5N')
```

```
<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

```
In [91]: date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
```

```
<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

```
In [92]: t = Timestamp('20130101 09:01:02')
```

```
In [93]: t + pd.datetools.Nano(123)
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:

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

```
In [95]: dfi
```

```
   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
```

```
   A      B
```

```
0    True  False
1    False False
2     True   True
3    False  False
[4 rows x 2 columns]
```

```
In [99]: dfi[mask.any(1)]
```

```
      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()
```

```
      A      B
0  1.0  0.25
1  2.1  1.50
2  3.4  2.75
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40
[6 rows x 2 columns]
```

Additionally, the method argument to `interpolate` has been expanded to include `'nearest'`, `'zero'`, `'slinear'`, `'quadratic'`, `'cubic'`, `'barycentric'`, `'krogh'`, `'piecewise_polynomial'`, `'pchip'`, `'polynomial'`, `'spline'`. The new methods require `scipy`. Consult the [Scipy reference guide](#) and [documentation](#) for more information about when the various methods are appropriate. See *the docs*.

Interpolate now also accepts a `limit` keyword argument. This works similar to `fillna`'s `limit`:

```
In [102]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])
```

```
In [103]: ser.interpolate(limit=2)
```

```
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"},
.....:                      "A1980" : {0 : "d", 1 : "e", 2 : "f"},
.....:                      "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
.....:                      "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
.....:                      "X"      : dict(zip(range(3), np.random.randn(3)))
.....:                      })
```

```
In [106]: df["id"] = df.index
```

```
In [107]: df
```

```
   A1970 A1980 B1970 B1980      X  id
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")
```

```
      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](#))

1.1.9 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: 20 ms per loop

# pure Python evaluation
In [112]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 58.4 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')

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')

   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')

      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')

[      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,
# publicdata:samples.gsod, is available on all BigQuery accounts,
# and is based on NOAA gsod data.

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

# Fetch the result set for this query

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

df = gbq.read_gbq(query, project_id = projectid)

# Use pandas to process and reshape the dataset

df2 = df.pivot(index='STATION', columns='MONTH', values='MEAN_TEMP')
df3 = pandas.concat([df2.min(), df2.mean(), df2.max()],
                    axis=1, keys=["Min Tem", "Mean Temp", "Max Temp"])
```

The resulting DataFrame is:

```
> df3
```

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

Warning: To use this module, you will need a BigQuery account. See <https://cloud.google.com/products/big-query> for details.
As of 10/10/13, there is a bug in Google's API preventing result sets from being larger than 100,000 rows. A patch is scheduled for the week of 10/14/13.

1.1.10 Internal Refactoring

In 0.13.0 there is a major refactor primarily to subclass `Series` from `NDFrame`, which is the base class currently for `DataFrame` and `Panel`, to unify methods and behaviors. `Series` formerly subclassed directly from `ndarray`. ([GH4080](#), [GH3862](#), [GH816](#))

Warning: There are two potential incompatibilities from < 0.13.0

- Using certain numpy functions would previously return a `Series` if passed a `Series` as an argument. This seems only to affect `np.ones_like`, `np.empty_like`, `np.diff` and `np.where`. These now return `ndarrays`.

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

Numpy Usage

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

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

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

Pandonic Usage

```
In [129]: Series(1,index=s.index)
```

```
0    1
1    1
2    1
3    1
dtype: int64
```

```
In [130]: s.diff()
```

```
0    NaN
1     1
2     1
3     1
dtype: float64
```

```
In [131]: s.where(s>1)
```

```
0    NaN
1     2
2     3
3     4
dtype: float64
```

- Passing a `Series` directly to a cython function expecting an `ndarray` type will no longer 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 `ipy2<=2.3.8`. an Issue has been opened against `ipy2` and a workaround is detailed in [GH5698](#). Thanks @JanSchulz.

- Pickle compatibility is preserved for pickles created prior to 0.13. These must be unpickled with `pd.read_pickle`, see [Pickling](#).

- Refactor of `series.py/frame.py/panel.py` to move common code to `generic.py`
 - added `__setup_axes` to created generic NDFrame structures
 - moved methods
 - * `from_axes, __wrap_array, axes, ix, loc, iloc, shape, empty, swapaxes, transpose, pop`
 - * `__iter__, keys, __contains__, __len__, __neg__, __invert__`
 - * `convert_objects, as_blocks, as_matrix, values`
 - * `__getstate__, __setstate__` (compat remains in frame/panel)
 - * `__getattr__, __setattr__`
 - * `_indexed_same, reindex_like, align, where, mask`
 - * `fillna, replace` (Series replace is now consistent with DataFrame)
 - * `filter` (also added axis argument to selectively filter on a different axis)
 - * `reindex, reindex_axis, take`
 - * `truncate` (moved to become part of NDFrame)
- These are API changes which make Panel more consistent with DataFrame
 - `swapaxes` on a Panel with the same axes specified now return a copy
 - support attribute access for setting
 - filter supports the same API as the original DataFrame filter
- Reindex called with no arguments will now return a copy of the input object
- `TimeSeries` is now an alias for `Series`. the property `is_time_series` can be used to distinguish (if desired)
- Refactor of Sparse objects to use BlockManager
 - Created a new block type in internals, `SparseBlock`, which can hold multi-dtypes and is non-consolidatable. `SparseSeries` and `SparseDataFrame` now inherit more methods from 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 class, courtesy of @jtratrner
- 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
2
```

```
In [134]: s.a = 5
```

```
In [135]: s
```

```
a    5
b    2
c    3
dtype: int64
```

1.1.11 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.

1.2 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.

1.2.1 API changes

- The I/O API is now much more consistent with a set of top level reader functions accessed like `pd.read_csv()` that generally return a pandas object.
 - `read_csv`
 - `read_excel`
 - `read_hdf`

```
- read_sql
- read_json
- read_html
- read_stata
- read_clipboard
```

The corresponding writer functions are object methods that are accessed like `df.to_csv()`

```
- to_csv
- to_excel
- to_hdf
- to_sql
- to_json
- to_html
- to_stata
- to_clipboard
```

- Fix modulo and integer division on Series,DataFrames to act similarly to float dtypes to return `np.nan` or `np.inf` as appropriate (GH3590). This correct a numpy bug that treats integer and float dtypes differently.

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

```
In [2]: p % 0
```

```
   first  second
0    NaN    NaN
1    NaN    NaN
2    NaN    NaN
[3 rows x 2 columns]
```

```
In [3]: p % p
```

```
   first  second
0      0    NaN
1      0    NaN
2      0      0
[3 rows x 2 columns]
```

```
In [4]: p / p
```

```
   first  second
0      1      inf
1      1      inf
2      1  1.000000
[3 rows x 2 columns]
```

```
In [5]: p / 0
```

```
   first  second
0     inf     inf
1     inf     inf
```

```
2      inf      inf
[3 rows x 2 columns]
```

- Add `squeeze` keyword to `groupby` to allow reduction from `DataFrame` -> `Series` if groups are unique. This is a Regression from 0.10.1. We are reverting back to the prior behavior. This means `groupby` will return the same shaped objects whether the groups are unique or not. Revert this issue ([GH2893](#)) with ([GH3596](#)).

```
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)
```

```
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)
```

```
      0      1      2      3
val1
1      0.5 -0.5  7.5 -7.5
[1 rows x 4 columns]
```

- Raise on `iloc` when boolean indexing with a label based indexer mask e.g. a boolean `Series`, even with integer labels, will raise. Since `iloc` is purely positional based, the labels on the `Series` are not alignable ([GH3631](#))

This case is rarely used, and there are plenty of alternatives. This preserves the `iloc` API to be *purely* positional based.

```
In [10]: df = DataFrame(lrange(5), list('ABCDE'), columns=['a'])
```

```
In [11]: mask = (df.a%2 == 0)
```

```
In [12]: mask
```

```
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]
```

```
      a
A      0
C      2
E      4
[3 rows x 1 columns]
```

```
# this will work as well
In [14]: df.iloc[mask.values]
```

```
      a
A    0
C    2
E    4
[3 rows x 1 columns]
```

`df.iloc[mask]` will raise a `ValueError`

- The `raise_on_error` argument to plotting functions is removed. Instead, plotting functions raise a `TypeError` when the dtype of the object is object to remind you to avoid object arrays whenever possible and thus you should cast to an appropriate numeric dtype if you need to plot something.
- Add `colormap` keyword to `DataFrame` plotting methods. Accepts either a matplotlib colormap object (ie, `matplotlib.cm.jet`) or a string name of such an object (ie, `'jet'`). The colormap is sampled to select the color for each column. Please see [Colormaps](#) for more information. (GH3860)
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead. (GH3582, GH3675, GH3676)
- the method and axis arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. (GH3907)
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if True, default is False (same as prior to 0.12) (GH3679)
- Implement `__nonzero__` for `NDFrame` objects (GH3691, GH3696)
- IO api

- added top-level function `read_excel` to replace the following. The original API is deprecated and will be removed in a future version

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

With

```
import pandas as pd
pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

- added top-level function `read_sql` that is equivalent to the following

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

- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Do not allow astypes on `datetime64[ns]` except to object, and `timedelta64[ns]` to object/int (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when performed on a `Series` and return an empty `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of slice objects:
 - `sum`, `prod`, `mean`, `std`, `var`, `skew`, `kurt`, `corr`, and `cov`

- `read_html` now defaults to `None` when reading, and falls back on `bs4` + `html5lib` when `lxml` fails to parse. a list of parsers to try until success is also valid
- The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). ([GH4090](#), [GH4092](#))
- New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. ([GH4090](#), [GH4092](#))

1.2.2 I/O Enhancements

- `pd.read_html()` can now parse HTML strings, files or urls and return `DataFrames`, courtesy of [@cpcloud](#). ([GH3477](#), [GH3605](#), [GH3606](#), [GH3616](#)). It works with a *single* parser backend: `BeautifulSoup4` + `html5lib` *See the docs*

You can use `pd.read_html()` to read the output from `DataFrame.to_html()` like so

```
In [15]: df = DataFrame({'a': range(3), 'b': list('abc')})

In [16]: print(df)
   a  b
0  0  a
1  1  b
2  2  c
[3 rows x 2 columns]

In [17]: html = df.to_html()

In [18]: alist = pd.read_html(html, infer_types=True, index_col=0)

In [19]: print(df == alist[0])
   a  b
0  True True
1  True True
2  True True
[3 rows x 2 columns]
```

Note that `alist` here is a Python list so `pd.read_html()` and `DataFrame.to_html()` are not inverses.

- `pd.read_html()` no longer performs hard conversion of date strings ([GH3656](#)).

Warning: You may have to install an older version of `BeautifulSoup4`, *See the installation docs*

- 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())
```

```
C0,,C_10_g0,C_10_g1,C_10_g2
C1,,C_11_g0,C_11_g1,C_11_g2
C2,,C_12_g0,C_12_g1,C_12_g2
C3,,C_13_g0,C_13_g1,C_13_g2
R0,R1,,,
R_10_g0,R_11_g0,R0C0,R0C1,R0C2
R_10_g1,R_11_g1,R1C0,R1C1,R1C2
R_10_g2,R_11_g2,R2C0,R2C1,R2C2
R_10_g3,R_11_g3,R3C0,R3C1,R3C2
R_10_g4,R_11_g4,R4C0,R4C1,R4C2
```

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

```
C0          C_10_g0 C_10_g1 C_10_g2
C1          C_11_g0 C_11_g1 C_11_g2
C2          C_12_g0 C_12_g1 C_12_g2
C3          C_13_g0 C_13_g1 C_13_g2
R0      R1
R_10_g0 R_11_g0      R0C0      R0C1      R0C2
R_10_g1 R_11_g1      R1C0      R1C1      R1C2
R_10_g2 R_11_g2      R2C0      R2C1      R2C2
R_10_g3 R_11_g3      R3C0      R3C1      R3C2
R_10_g4 R_11_g4      R4C0      R4C1      R4C2
[5 rows x 3 columns]
```

- 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
1 -0.402761 -0.246604
2 -0.288433 -0.763434
[3 rows x 2 columns]
```

```
      0      1
```

```

3  2.069526 -1.203569
4  0.591830  0.841159
5 -0.501083 -0.816561
[3 rows x 2 columns]
      0      1
6 -0.207082 -0.664112
7  0.580411 -0.965628
8 -0.038605 -0.460478
[3 rows x 2 columns]
      0      1
9 -0.310458  0.866493
[1 rows x 2 columns]

```

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

1.2.3 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*\.\s*', value=np.nan)

```

```

      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)

```

```

      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.set_option('a.b')
2

```

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

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

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

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

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)

   A  B
2  2  b
3  3  b
4  4  b
5  5  b
[4 rows x 2 columns]
```

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [40]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)

   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]
```

- `Series` and `DataFrame` `hist` methods now take a `figsize` argument ([GH3834](#))
- `DatetimeIndex`s no longer try to convert mixed-integer indexes during join operations ([GH3877](#))
- `Timestamp.min` and `Timestamp.max` now represent valid `Timestamp` instances instead of the default `date-time.min` and `datetime.max` (respectively), thanks @SleepingPills

- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

1.2.4 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.

```
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
```

1.2.5 Bug Fixes

- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have a `dtype` of `object` (GH1818, GH3572, GH3911, GH3912), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which eliminates any spurious plots from showing up.
- `fillna` methods now raise a `TypeError` if the `value` parameter is a list or tuple.
- `Series.str` now supports iteration (GH3638). You can iterate over the individual elements of each string in the `Series`. Each iteration yields a `Series` with either a single character at each index of the original `Series` or `NaN`. For example,

```
In [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
0      o
1      o
2      o
3      l
dtype: object
0      NaN
1      w
2      e
3      o
dtype: object
0      NaN
1      NaN
2      NaN
3      w
dtype: object
```

```
In [52]: s
```

```
0      NaN
1      NaN
2      NaN
3      w
dtype: object
```

```
In [53]: s.dropna().values.item() == 'w'
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.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)
- `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 were open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in `test_yahoo`, `test_google` where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN `DataFrame` would barf on a 1xN mask (GH4071)
- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where `sharex` and `sharey` were not being passed to `grouped_hist` (GH4089)
- 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.

1.3 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*

1.3.1 Selection Choices

Starting in 0.11.0, object selection has had a number of user-requested additions in order to support more explicit location based indexing. Pandas now supports three types of multi-axis indexing.

- `.loc` is strictly label based, will raise `KeyError` when the items are not found, allowed inputs are:
 - A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index. This use is **not** an integer position along the index)
 - A list or array of labels ['a', 'b', 'c']
 - A slice object with labels 'a':'f', (note that contrary to usual python slices, **both** the start and the stop are included!)
 - A boolean array

See more at [Selection by Label](#)

- `.iloc` is strictly integer position based (from 0 to `length-1` of the axis), will raise `IndexError` when the requested indicies are out of bounds. Allowed inputs are:
 - An integer e.g. 5
 - A list or array of integers [4, 3, 0]
 - A slice object with ints 1:7
 - A boolean array

See more at [Selection by Position](#)

- `.ix` supports mixed integer and label based access. It is primarily label based, but will fallback to integer positional access. `.ix` is the most general and will support any of the inputs to `.loc` and `.iloc`, as well as support for floating point label schemes. `.ix` is especially useful when dealing with mixed positional and label based hierarchical indexes.

As using integer slices with `.ix` have different behavior depending on whether the slice is interpreted as position based or label based, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at [Advanced Indexing](#), [Advanced Hierarchical](#) and [Fallback Indexing](#)

1.3.2 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.

1.3.3 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
```

```
      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
```

```
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
```

```
      A      B      C
0 -0.611328 -0.270630  255
1  1.044922 -1.685677    0
2  1.503906 -0.440747    0
3 -1.328125 -0.115070    1
4  1.024414 -0.632102    0
5  0.660156 -0.585977    0
6  1.236328 -1.444787    0
7 -2.169922 -0.201135    0
[8 rows x 3 columns]
```

```
In [6]: df2.dtypes
```

```
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
```

```
      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
```

```
A    float32
B    float64
C    float64
dtype: object
```

1.3.4 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
dtype('float64')
```

Conversion

```
In [11]: df3.astype('float32').dtypes
```

```
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
```

```
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

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')

0    2001-01-01 00:00:00
1                      NaT
2                      NaT
3                      NaT
4    2001-01-04 00:00:00
5    2001-01-05 00:00:00
dtype: datetime64[ns]
```

1.3.5 Dtype Gotchas

Platform Gotchas

Starting in 0.11.0, construction of DataFrame/Series will use default dtypes of `int64` and `float64`, *regardless of platform*. This is not an apparent change from earlier versions of pandas. If you specify dtypes, they *WILL* be respected, however ([GH2837](#))

The following will all result in `int64` dtypes

```
In [20]: DataFrame([1,2], columns=['a']).dtypes

a    int64
dtype: object

In [21]: DataFrame({'a' : [1,2]}).dtypes

a    int64
dtype: object

In [22]: DataFrame({'a' : 1 }, index=range(2)).dtypes

a    int64
dtype: object
```

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

Upcasting Gotchas

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

```
In [23]: dfi = df3.astype('int32')
```

```
In [24]: dfi['D'] = dfi['D'].astype('int64')
```

```
In [25]: dfi
```

```
   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
```

```
A      int32
B      int32
C      int32
D      int64
E      int32
dtype: object
```

```
In [27]: casted = dfi[dfi>0]
```

```
In [28]: casted
```

```
   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   1  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
```

```
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
```

```
A    float32
B    float64
C    float64
D    float16
E      int32
dtype: object
```

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

```
In [34]: casted
```

```
      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
```

```
A    float32
B    float64
C    float64
D    float16
E      int32
dtype: object
```

1.3.6 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
```

```
      A      B      timestamp
2001-01-02 -1.448835  0.153437 2001-01-03 00:00:00
2001-01-03 -1.123570 -0.791498 2001-01-03 00:00:00
2001-01-04  0.105400  1.262401 2001-01-03 00:00:00
2001-01-05 -0.721844 -0.647645 2001-01-03 00:00:00
2001-01-06 -0.830631  0.761823 2001-01-03 00:00:00
2001-01-07  0.597819  1.045558 2001-01-03 00:00:00
[6 rows x 3 columns]
```

```
# datetime64[ns] out of the box
```

```
In [39]: df.get_dtype_counts()
```

```
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
```

```
          A          B          timestamp
2001-01-02 -1.448835  0.153437 2001-01-03 00:00:00
2001-01-03 -1.123570 -0.791498 2001-01-03 00:00:00
2001-01-04      NaN  1.262401             NaT
2001-01-05      NaN -0.647645             NaT
2001-01-06 -0.830631  0.761823 2001-01-03 00:00:00
2001-01-07  0.597819  1.045558 2001-01-03 00:00:00
[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
dtype('<M8[ns]')
```

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

```
In [46]: s
```

```
0    2001-01-02 00:00:00
1                NaT
2    2001-01-02 00:00:00
dtype: datetime64[ns]
```

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

```
In [48]: s = s.astype('O')
```

```
In [49]: s
```

```
0    2001-01-02 00:00:00
1                NaN
2    2001-01-02 00:00:00
dtype: object
```

```
In [50]: s.dtype
dtype('O')
```

1.3.7 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

1.3.8 Enhancements

- Improved performance of `df.to_csv()` by up to 10x in some cases. (GH3059)
- Numexpr is now a *Recommended Dependencies*, to accelerate certain types of numerical and boolean operations
- Bottleneck is now a *Recommended Dependencies*, to accelerate certain types of nan operations
- HDFStore

- support `read_hdf/to_hdf` API similar to `read_csv/to_csv`

```
In [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'])
```

```
   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']
```

```
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']
```

```
   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
```

```
<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()
```

```
          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()
```

```
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/huynh/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.

1.4 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.

1.4.1 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)

1.4.2 New features

- MySQL support for database (contribution from Dan Allan)

1.4.3 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`

```
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
```

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

```
# on-disk operations
```

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

```
In [9]: store.select('df',[ 'B > 0', 'string == foo' ])
```

```
          A          B          C string string2
2000-01-04 -0.761218  1.768215  0.152288    foo    cool
[1 rows x 5 columns]
```

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

```
In [10]: df[(df.B > 0) & (df.string == 'foo')]
```

```
          A          B          C string string2
2000-01-04 -0.761218  1.768215  0.152288    foo    cool
[1 rows x 5 columns]
```

Retrieving unique values in an indexable or data column.

```
In [11]: import warnings
```

```
In [12]: with warnings.catch_warnings():
.....:     warnings.simplefilter('ignore', category=UserWarning)
.....:     store.unique('df','index')
.....:     store.unique('df','string')
.....:
```

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
```

```
          A          B          C string string2  datetime64
2000-01-01 -1.601262 -0.256718  0.239369    foo    cool 2001-01-02 00:00:00
2000-01-02  0.174122 -1.131794 -1.948006    foo    cool 2001-01-02 00:00:00
2000-01-03  0.980347 -0.674429 -0.361633    foo    cool 2001-01-02 00:00:00
```

```

2000-01-04      NaN      NaN  0.152288    foo    cool 2001-01-02 00:00:00
2000-01-05 -0.862613 -0.210968 -0.859278    NaN    cool 2001-01-02 00:00:00
2000-01-06  1.498195  0.462413 -0.647604    NaN    cool 2001-01-02 00:00:00
2000-01-07  1.511487 -0.727189 -0.342928    foo    cool 2001-01-02 00:00:00
2000-01-08 -0.007364  1.427674  0.104020    bar    cool 2001-01-02 00:00:00
[8 rows x 6 columns]

```

```
In [19]: df_mixed1.get_dtype_counts()
```

```

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'])
```

```

          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
2000-01-07  1.511487 -0.727189
2000-01-08 -0.007364  1.427674
[8 rows x 2 columns]

```

HDFStore now serializes multi-index dataframes when appending tables.

```

In [21]: 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 [22]: df = DataFrame(np.random.randn(10, 3), index=index,
.....:                  columns=['A', 'B', 'C'])
.....:

```

```
In [23]: df
```

```

          A          B          C
foo bar
foo one    2.052171 -1.230963 -0.019240
   two   -1.713238  0.838912 -0.637855
   three  0.215109 -1.515362  1.586924
bar one   -0.447974 -1.573998  0.630925
   two   -0.071659 -1.277640 -0.102206
baz two    0.870302  1.275280 -1.199212
   three  1.060780  1.673018  1.249874
qux one    1.458210 -0.710542  0.825392
   two    1.557329  1.993441 -0.616293
   three  0.150468  0.132104  0.580923
[10 rows x 3 columns]

```

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

```
In [25]: store.select('mi')
```

```
          A          B          C
foo bar
foo one    2.052171 -1.230963 -0.019240
   two    -1.713238  0.838912 -0.637855
   three    0.215109 -1.515362  1.586924
bar one   -0.447974 -1.573998  0.630925
   two   -0.071659 -1.277640 -0.102206
baz two    0.870302  1.275280 -1.199212
   three  1.060780  1.673018  1.249874
qux one    1.458210 -0.710542  0.825392
   two    1.557329  1.993441 -0.616293
   three  0.150468  0.132104  0.580923
[10 rows x 3 columns]
```

```
# the levels are automatically included as data columns
```

```
In [26]: store.select('mi', Term('foo=bar'))
```

```
          A          B          C
foo bar
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
```

```
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df          frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,stri
/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->[b
```

```
# individual tables were created
```

```
In [31]: store.select('df1_mt')
```

```
          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')
```

```
      C      D      E      F  foo
2000-01-01 -0.431163  0.016640  0.904578 -1.645852  bar
2000-01-02  0.800353 -0.451572  0.831767  0.228760  bar
2000-01-03  1.239198  0.185437 -0.540770 -0.370038  bar
2000-01-04 -0.040863  0.290110 -0.096145  1.717830  bar
2000-01-05 -0.134024 -0.205969  1.348944 -1.198246  bar
2000-01-06  0.059493 -0.460111 -1.565401 -0.025706  bar
2000-01-07 -0.302741  0.261551 -0.066342  0.897097  bar
2000-01-08  0.582752 -1.490764 -0.639757 -0.952750  bar
[8 rows x 5 columns]
```

```
# as a multiple
```

```
In [33]: store.select_as_multiple(['df1_mt', 'df2_mt'], where = [ 'A>0', 'B>0' ], selector = 'df1_mt')
```

```
      A      B      C      D      E      F  foo
2000-01-04  1.298390  1.662964 -0.040863  0.290110 -0.096145  1.717830  bar
2000-01-07  0.942864  2.502156 -0.302741  0.261551 -0.066342  0.897097  bar
[2 rows x 7 columns]
```

Enhancements

- HDFStore now can read native PyTables table format tables
- You can pass `nan_rep = 'my_nan_rep'` to `append`, to change the default nan representation on disk (which converts to/from `np.nan`), this defaults to `nan`.
- You can pass `index` to `append`. This defaults to `True`. This will automatically create indices on the *indexables* and *data columns* of the table
- You can pass `chunksize=an integer` to `append`, to change the writing chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=an integer` to the first `append`, to set the TOTAL number of expected rows that PyTables will expect. This will optimize read/write performance.
- `Select` now supports passing `start` and `stop` to provide selection space limiting in selection.
- Greatly improved ISO8601 (e.g., yyyy-mm-dd) date parsing for file parsers ([GH2698](#))
- Allow `DataFrame.merge` to handle combinatorial sizes too large for 64-bit integer ([GH2690](#))
- `Series` now has unary negation (`-series`) and inversion (`~series`) operators ([GH2686](#))
- `DataFrame.plot` now includes a `logx` parameter to change the x-axis to log scale ([GH2327](#))
- `Series` arithmetic operators can now handle constant and `ndarray` input ([GH2574](#))
- `ExcelFile` now takes a `kind` argument to specify the file type ([GH2613](#))
- A faster implementation for `Series.str` methods ([GH2602](#))

Bug Fixes

- `HDFStore` tables can now store `float32` types correctly (cannot be mixed with `float64` however)
- Fixed Google Analytics prefix when specifying request segment ([GH2713](#)).

- Function to reset Google Analytics token store so users can recover from improperly setup client secrets ([GH2687](#)).
- Fixed groupby bug resulting in segfault when passing in MultiIndex ([GH2706](#))
- Fixed bug where passing a Series with datetime64 values into *to_datetime* results in bogus output values ([GH2699](#))
- Fixed bug in *pattern* in *HDFStore* expressions when *pattern* is not a valid regex ([GH2694](#))
- Fixed performance issues while aggregating boolean data ([GH2692](#))
- When given a boolean mask key and a Series of new values, Series *__setitem__* will now align the incoming values with the original Series ([GH2686](#))
- Fixed MemoryError caused by performing counting sort on sorting MultiIndex levels with a very large number of combinatorial values ([GH2684](#))
- Fixed bug that causes plotting to fail when the index is a DatetimeIndex with a fixed-offset timezone ([GH2683](#))
- Corrected businessday subtraction logic when the offset is more than 5 bdays and the starting date is on a weekend ([GH2680](#))
- Fixed C file parser behavior when the file has more columns than data ([GH2668](#))
- Fixed file reader bug that misaligned columns with data in the presence of an implicit column and a specified *usecols* value
- DataFrames with numerical or datetime indices are now sorted prior to plotting ([GH2609](#))
- Fixed DataFrame.from_records error when passed columns, index, but empty records ([GH2633](#))
- Several bug fixed for Series operations when dtype is datetime64 ([GH2689](#), [GH2629](#), [GH2626](#))

See the [full release notes](#) or issue tracker on GitHub for a complete list.

1.5 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.

1.5.1 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_reccarray*)
- 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

1.5.2 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 methods 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
```

```

      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]
```

```

      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)
```

```

      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
```

```
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')
```

```
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')
```

```
2000-01-01    0
2000-01-02   21
2000-01-03   57
2000-01-04   93
2000-01-05  129
Freq: D, dtype: int64
```

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

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

```
In [12]: pd.isnull(s)
```

```
0    False
1    False
2    False
3    False
dtype: bool
```

```
In [13]: s.fillna(0)
```

```
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)
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

```
In [16]: s.fillna(0)
```

```
0    1.5
1    0.0
2    3.4
3    0.0
dtype: float64
```

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

- Methods with the `inplace` option now all return `None` instead of the calling object. E.g. code written like `df = df.fillna(0, inplace=True)` may stop working. To fix, simply delete the unnecessary variable assignment.
- `pandas.merge` no longer sorts the group keys (`sort=False`) by default. This was done for performance reasons: the group-key sorting is often one of the more expensive parts of the computation and is often unnecessary.
- The default column names for a file with no header have been changed to the integers 0 through $N - 1$. This is to create consistency with the `DataFrame` constructor with no columns specified. The v0.9.0 behavior (names `X0, X1, ...`) can be reproduced by specifying `prefix='X'`:

```
In [18]: data= 'a,b,c\n1,Yes,2\n3,No,4'
```

```
In [19]: print(data)
```

```
a,b,c
1,Yes,2
3,No,4
```

```
In [20]: pd.read_csv(StringIO(data), header=None)
```

```
   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')
```

```
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))
```

```
   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'])
```

```
   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
```

```
0    NaN
1     1
2     2
3    NaN
4     4
dtype: float64
```

```
In [27]: s.fillna(0)
```

```
0     0
1     1
2     2
3     0
4     4
dtype: float64
```

```
In [28]: s.fillna(method='pad')
```

```
0    NaN
1     1
2     2
3     2
```

```
4      4
dtype: float64
```

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

```
In [29]: s.fffll()
```

```
0      NaN
1         1
2         2
3         2
4         4
dtype: float64
```

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

```
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
```

```
0      0.013135
1      0.909855
2      0.098093
3      0.023540
4      0.141354
dtype: float64
```

```
In [33]: s.apply(f)
```

```
      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")
15
```

- `to_string()` methods now always return unicode strings ([GH2224](#)).

1.5.3 New features

1.5.4 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
```

```
      0         1         2         3         4         5         6  \
0  2.520045  1.570114 -0.360875 -0.880096  0.235532  0.207232 -1.983857
1  0.422194  0.288403 -0.487393 -0.777639  0.055865  1.383381  0.085638
2  0.585174 -0.568825 -0.719412  1.191340 -0.456362  0.089931  0.776079
3  1.218080 -0.564705 -0.581790  0.286071  0.048725  1.002440  1.276582
4 -0.376280  0.511936 -0.116412 -0.625256 -0.550627  1.261433 -0.552429
      7         8         9        10        11        12        13  \
0 -1.702547 -1.621234 -0.906840  1.014601 -0.475108 -0.358944  1.262942
1  0.246392  0.965887  0.246354 -0.727728 -0.094414 -0.276854  0.158399
2  0.752889 -1.195795 -1.425911 -0.548829  0.774225  0.740501  1.510263
3  0.054399  0.241963 -0.471786  0.314510 -0.059986 -2.069319 -1.115104
4  1.695803 -1.025917 -0.910942  0.426805 -0.131749  0.432600  0.044671
      14        15
0 -0.412451 -0.462580
1 -0.277255  1.331263
2 -1.642511  0.432560
3 -0.369325 -1.502617
4 -0.341265  1.844536
[5 rows x 16 columns]
```

The old behavior of printing out summary information can be achieved via the ‘expand_frame_repr’ print option:

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

```
In [38]: wide_frame
```

```
      0         1         2         3         4         5         6         7         8         9
0  2.520045  1.570114 -0.360875 -0.880096  0.235532  0.207232 -1.983857 -1.702547 -1.621234 -0.906840
1  0.422194  0.288403 -0.487393 -0.777639  0.055865  1.383381  0.085638  0.246392  0.965887  0.246354
2  0.585174 -0.568825 -0.719412  1.191340 -0.456362  0.089931  0.776079  0.752889 -1.195795 -1.425911
3  1.218080 -0.564705 -0.581790  0.286071  0.048725  1.002440  1.276582  0.054399  0.241963 -0.471786
4 -0.376280  0.511936 -0.116412 -0.625256 -0.550627  1.261433 -0.552429  1.695803 -1.025917 -0.910942
[5 rows x 16 columns]
```

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

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

```
In [40]: wide_frame
```

```
      0         1         2  \
0  2.520045  1.570114 -0.360875
1  0.422194  0.288403 -0.487393
2  0.585174 -0.568825 -0.719412
3  1.218080 -0.564705 -0.581790
4 -0.376280  0.511936 -0.116412
      3         4         5  \
0 -0.880096  0.235532  0.207232
```



```

1 -0.777639  0.055865  1.383381
2  1.191340 -0.456362  0.089931
3  0.286071  0.048725  1.002440
4 -0.625256 -0.550627  1.261433
      6      7      8  \
0 -1.983857 -1.702547 -1.621234
1  0.085638  0.246392  0.965887
2  0.776079  0.752889 -1.195795
3  1.276582  0.054399  0.241963
4 -0.552429  1.695803 -1.025917
      9     10     11  \
0 -0.906840  1.014601 -0.475108
1  0.246354 -0.727728 -0.094414
2 -1.425911 -0.548829  0.774225
3 -0.471786  0.314510 -0.059986
4 -0.910942  0.426805 -0.131749
      12     13     14  \
0 -0.358944  1.262942 -0.412451
1 -0.276854  0.158399 -0.277255
2  0.740501  1.510263 -1.642511
3 -2.069319 -1.115104 -0.369325
4  0.432600  0.044671 -0.341265
      15
0 -0.462580
1  1.331263
2  0.432560
3 -1.502617
4  1.844536
[5 rows x 16 columns]

```

1.5.5 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
```

```

              A              B              C
2000-01-01 -2.036047  0.000830 -0.955697
2000-01-02 -0.898872 -0.725411  0.059904
2000-01-03 -0.449644  1.082900 -1.221265
2000-01-04  0.361078  1.330704  0.855932
2000-01-05 -1.216718  1.488887  0.018993
2000-01-06 -0.877046  0.045976  0.437274
2000-01-07 -0.567182 -0.888657 -0.556383
2000-01-08  0.655457  1.117949 -2.782376
[8 rows x 3 columns]

```

```
# 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

<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')

           A          B          C
2000-01-01 -2.036047  0.000830 -0.955697
2000-01-02 -0.898872 -0.725411  0.059904
2000-01-03 -0.449644  1.082900 -1.221265
2000-01-04  0.361078  1.330704  0.855932
2000-01-05 -1.216718  1.488887  0.018993
2000-01-06 -0.877046  0.045976  0.437274
2000-01-07 -0.567182 -0.888657 -0.556383
2000-01-08  0.655457  1.117949 -2.782376
[8 rows x 3 columns]

In [50]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
....:               major_axis=date_range('1/1/2000', periods=5),
....:               minor_axis=['A', 'B', 'C', 'D'])
....:

In [51]: wp

<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']) ])
....:

<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'))
8

In [55]: store.select('wp')

<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
```

```
<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
```

```
<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
```

```
<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
```

	A	B	C	string	int
2000-01-01	-2.036047	0.000830	-0.955697	string	1
2000-01-02	-0.898872	-0.725411	0.059904	string	1
2000-01-03	-0.449644	1.082900	-1.221265	string	1

```
2000-01-04  0.361078  1.330704  0.855932  string  1
2000-01-05 -1.216718  1.488887  0.018993  string  1
2000-01-06 -0.877046  0.045976  0.437274  string  1
2000-01-07 -0.567182 -0.888657 -0.556383  string  1
2000-01-08  0.655457  1.117949 -2.782376  string  1
[8 rows x 5 columns]
```

```
In [69]: dfl.get_dtype_counts()
```

```
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.

1.5.6 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),
.....:    minor_axis=['A', 'B', 'C', 'D'])
.....:
```

```
In [71]: p4d

<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

See the [full release notes](#) or issue tracker on GitHub for a complete list.

1.6 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.

1.6.1 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])
```

```
   A  B  C
2  0  1  1
3  0  1  1
4  0  0  1
0  1  1  0
1  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()
```

```
   A  B  C
0  3  2  1
1  2  1  3
2 NaN NaN NaN
3 NaN NaN NaN
4 NaN NaN NaN
5  1  3  2
[6 rows x 3 columns]
```

```
In [6]: df.rank(na_option='top')
```

```
   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')
```

```
   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
```

```
   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]
```

```
   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]
```

```
   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]

```

```
In [12]: df.where(df>0)
```

```

      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]

```

```
In [13]: df.where(df>0,-df)
```

```

      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]

```

Furthermore, *where* now aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via *.ix* (but on the contents rather than the axis labels)

```
In [14]: df2 = df.copy()
```

```
In [15]: df2[ df2[1:4] > 0 ] = 3
```

```
In [16]: df2
```

```

      A      B      C
0  0.706220 -1.130744 -0.690308
1 -0.885387  3.000000  3.000000
2  3.000000 -1.189832 -0.344258
3  3.000000 -1.514102  3.000000
4  0.089527  0.576687 -0.737750
[5 rows x 3 columns]

```

DataFrame.mask is the inverse boolean operation of *where*.

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

```

      A      B      C
0  0.706220  NaN  NaN
1     NaN  0.246004  1.986687
2  0.212595  NaN  NaN
3  0.816335  NaN  1.298184
4  0.089527  0.576687  NaN
[5 rows x 3 columns]

```

- Enable referencing of Excel columns by their column names ([GH1936](#))

```
In [18]: xl = ExcelFile('data/test.xls')
```

```
In [19]: xl.parse('Sheet1', index_col=0, parse_dates=True,
.....:           parse_cols='A:D')
.....:
```

```
           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)

1.6.2 API changes

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

```
In [20]: prng = period_range('2012Q1', periods=2, freq='Q')
```

```
In [21]: s = Series(np.random.randn(len(prng)), prng)
```

```
In [22]: s.resample('M')
```

```
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
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\n00001,001,5\n00002,002,6'
```

```
In [26]: from cStringIO import StringIO
```



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

```
      A  B  C
0  00001  1  5
1  00002  2  6
[2 rows x 3 columns]
```

See the [full release notes](#) or issue tracker on GitHub for a complete list.

1.7 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.

1.7.1 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)

1.7.2 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\n1,1,0\n0,1,0'
```

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

```
In [4]: df
```

```
      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
```

```
0    1
1    2
2    3
dtype: int64
```

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

```
In [8]: s2
```

```
foo    NaN
bar    NaN
baz    NaN
dtype: float64
```

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

See the [full release notes](#) or issue tracker on GitHub for a complete list.

1.8 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.

1.8.1 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)

1.8.2 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

1.9 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. Lingerin 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.

1.9.1 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)

1.9.2 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.

1.9.3 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 local time. Time zone conversions are therefore essentially free. User needs to know very little about `pytz` library now; only time zone names as strings are required. Time zone-aware timestamps are equal if and only if their UTC timestamps match. Operations between time zone-aware time series with different time zones will result in a UTC-indexed time series.
- Time series **string indexing conveniences** / shortcuts: slice years, year and month, and index values with strings
- Enhanced time series **plotting**; adaptation of `scikits.timeseries` matplotlib-based plotting code
- New `date_range`, `bdate_range`, and `period_range` *factory functions*
- Robust **frequency inference** function `infer_freq` and `inferred_freq` property of `DatetimeIndex`, with option to infer frequency on construction of `DatetimeIndex`
- `to_datetime` function efficiently **parses array of strings** to `DatetimeIndex`. `DatetimeIndex` will parse array or list of strings to `datetime64`
- **Optimized** support for `datetime64-dtype` data in `Series` and `DataFrame` columns
- New `NaT` (Not-a-Time) type to represent **NA** in timestamp arrays
- Optimize `Series.asof` for looking up “as of” values for arrays of timestamps
- Milli, Micro, Nano date offset objects
- Can index time series with `datetime.time` objects to select all data at particular **time of day** (`TimeSeries.at_time`) or **between two times** (`TimeSeries.between_time`)
- Add *tshift* method for leading/lagging using the frequency (if any) of the index, as opposed to a naive lead/lag using `shift`

1.9.4 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>` 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`

1.9.5 New plotting methods

`Series.plot` now supports a `secondary_y` option:

```
In [1]: plt.figure()
<matplotlib.figure.Figure at 0x21837f50>

In [2]: fx['FR'].plot(style='g')
<matplotlib.axes.AxesSubplot at 0x21a64550>
```

```
In [3]: fx['IT'].plot(style='k--', secondary_y=True)
<matplotlib.axes.AxesSubplot at 0x21a7fe50>
```



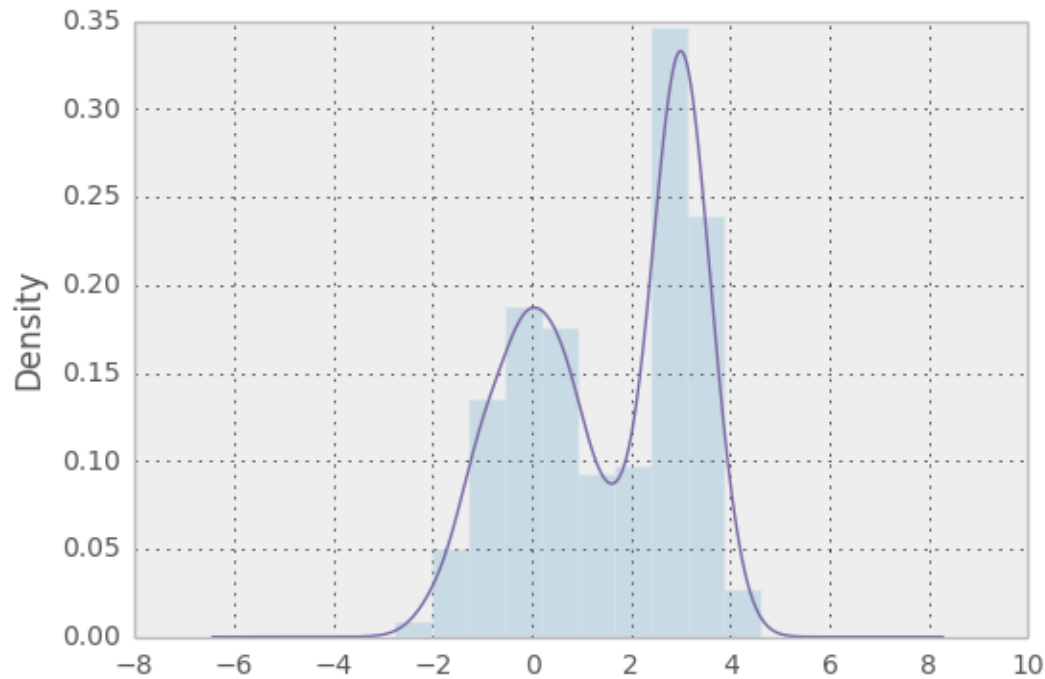
Vytautas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, 'kde' is a new option:

```
In [4]: s = Series(np.concatenate((np.random.randn(1000),
...:                               np.random.randn(1000) * 0.5 + 3)))
...:
```

```
In [5]: plt.figure()
<matplotlib.figure.Figure at 0x21837d50>
```

```
In [6]: s.hist(normed=True, alpha=0.2)
<matplotlib.axes.AxesSubplot at 0x20799190>
```

```
In [7]: s.plot(kind='kde')
<matplotlib.axes.AxesSubplot at 0x20799190>
```



See [the plotting page](#) for much more.

1.9.6 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.

1.9.7 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]
Timestamp('2000-01-06 00:00:00', tz=None)

In [11]: isinstance(rng[5], datetime.datetime)
True

In [12]: rng_asarray = np.asarray(rng)

In [13]: scalar_val = rng_asarray[5]

In [14]: type(scalar_val)
numpy.datetime64
```

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

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

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

```
In [16]: stamp_array
```

```
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]
```

```
Timestamp('2000-01-06 00:00:00', tz=None)
```

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

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

```
In [19]: dt_array
```

```
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]
```

```
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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-10 00:00:00]
Length: 10, Freq: D, Timezone: None
```

```
In [23]: np.asarray(rng)
```

```
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]
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.

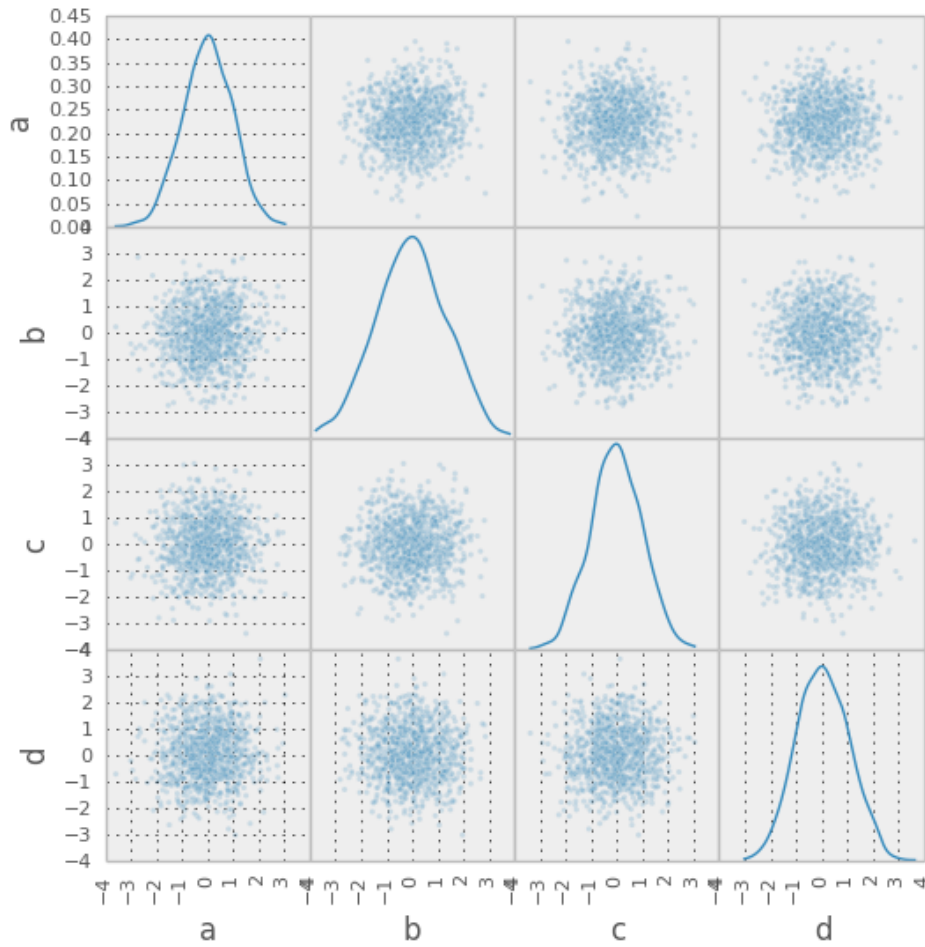
1.10 v.0.7.3 (April 12, 2012)

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.

1.10.1 New features

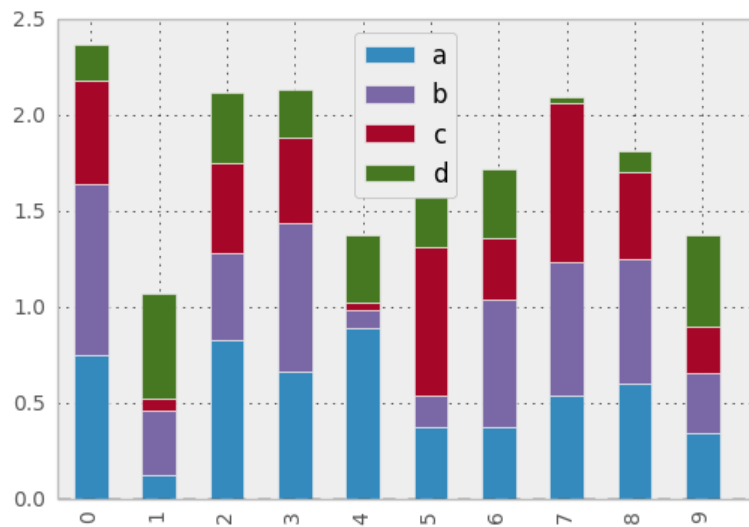
- New *fixed width file reader*, `read_fwf`
- New *scatter_matrix* function for making a scatter plot matrix

```
from pandas.tools.plotting import scatter_matrix
scatter_matrix(df, alpha=0.2)
```

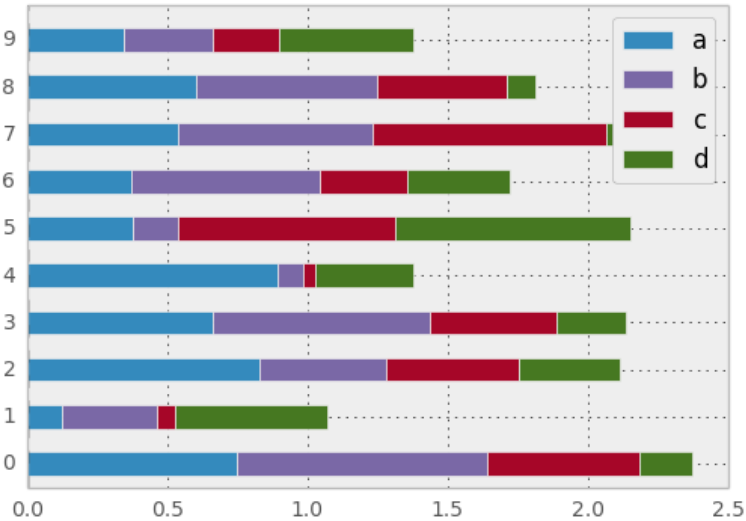


- Add stacked argument to Series and DataFrame's plot method for *stacked bar plots*.

```
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

1.10.2 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'
```

```
0    True
1   False
2   False
dtype: bool
```

```
In [3]: series != 'Steve'
```

```
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()]
```

```
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.

1.10.3 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
```

```
   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()
```

```
A
bar  count      3.000000
     mean     -0.900483
     std       0.782652
     min     -1.607906
     25%     -1.320859
...
foo   std       0.619410
     min     -1.275249
     25%     -0.054460
     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
```

```
A
bar  1    -1.033812
     3    -0.059730
foo  0     0.144909
     2     0.197333
dtype: float64
```

1.11 v.0.7.2 (March 16, 2012)

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

1.11.1 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)

1.11.2 Performance improvements

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

1.12 v.0.7.1 (February 29, 2012)

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

1.12.1 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`

1.12.2 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)

1.13 v.0.7.0 (February 9, 2012)

1.13.1 New features

- New unified *merge function* for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains (GH220, GH249, GH267)
- New *unified concatenation function* for concatenating Series, DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of `Series.append` and `DataFrame.append` (GH468, GH479, GH273)
- *Can* pass multiple DataFrames to `DataFrame.append` to concatenate (stack) and multiple Series to `Series.append` too
- *Can* pass list of dicts (e.g., a list of JSON objects) to DataFrame constructor (GH526)
- You can now *set multiple columns* in a DataFrame via `__getitem__`, useful for transformation (GH342)
- Handle differently-indexed output values in `DataFrame.apply` (GH498)

```
In [1]: df = DataFrame(randn(10, 4))
```

```
In [2]: df.apply(lambda x: x.describe())
```

```
count    0    1    2    3
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.

1.13.2 API Changes to integer indexing

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

```
In [3]: s = Series(randn(10), index=range(0, 20, 2))
```

```
In [4]: s
```

```
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
```

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

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

```
In [7]: s[4]
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
      0         1         2         3
0  0.88427  0.3363 -0.1787  0.03162
2  0.14451 -0.1415  0.2504  0.58374
4 -1.44779 -0.9186 -1.4996  0.27163
6 -0.26598 -2.4184 -0.2658  0.11503
8 -0.58776  0.3144 -0.8566  0.61941
10  0.10940 -0.7175 -1.0108  0.47990
12 -1.16919 -0.3087 -0.6049 -0.43544
14 -0.07337  0.3410  0.0424 -0.16037
```

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

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

Method	Description
Series.iget_value(i)	Retrieve value stored at location i
Series.iget(i)	Alias for iget_value
DataFrame.irow(i)	Retrieve the i-th row
DataFrame.icol(j)	Retrieve the j-th column
DataFrame.iget_value(i, j)	Retrieve the value at row i and column j

1.13.3 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
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']

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
```

```
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']
```

```
c    0.562726
e   -2.371548
g    1.269713
dtype: float64
```

1.13.4 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
```

```
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']]
```

```
m    0.945588
a    2.031757
c    0.851077
e    0.660056
dtype: float64
```

```
In [17]: s['b':'l']
```

```
c    0.851077
e    0.660056
g   -1.662471
k    0.571380
dtype: float64
```

```
In [18]: s['c':'k']
```

```
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]]
```

```
4    -1.263534
0    -0.414691
2     2.108285
dtype: float64
```

```
In [21]: s[1:5]
```

```
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`.

1.13.5 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](#))

1.13.6 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)

1.14 v.0.6.1 (December 13, 2011)

1.14.1 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*

1.14.2 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)

1.15 v.0.6.0 (November 25, 2011)

1.15.1 New Features

- *Added* `melt` function to `pandas.core.reshape`
- *Added* `level` parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- *Added* `head` and `tail` methods to Series, analogous to to DataFrame (GH296)
- *Added* `Series.isin` function which checks if each value is contained in a passed sequence (GH289)
- *Added* `float_format` option to `Series.to_string`
- *Added* `skip_footer` (GH291) and `converters` (GH343) options to `read_csv` and `read_table`
- *Added* `drop_duplicates` and `duplicated` functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- *Implemented* operators `&`, `|`, `^`, `-` on DataFrame (GH347)
- *Added* `Series.mad`, mean absolute deviation
- *Added* `QuarterEnd` `DateOffset` (GH321)
- *Added* `dot` to DataFrame (GH65)
- *Added* `orient` option to `Panel.from_dict` (GH359, GH301)
- *Added* `orient` option to `DataFrame.from_dict`
- *Added* passing list of tuples or list of lists to `DataFrame.from_records` (GH357)
- *Added* multiple levels to `groupby` (GH103)
- *Allow* multiple columns in `by` argument of `DataFrame.sort_index` (GH92, GH362)
- *Added* `fast_get_value` and `put_value` methods to DataFrame (GH360)
- *Added* `cov` instance methods to Series and DataFrame (GH194, GH362)
- *Added* `kind='bar'` option to `DataFrame.plot` (GH348)
- *Added* `idxmin` and `idxmax` to Series and DataFrame (GH286)
- *Added* `read_clipboard` function to parse DataFrame from clipboard (GH300)
- *Added* `nunique` function to Series for counting unique elements (GH297)
- *Made* DataFrame constructor use Series name if no columns passed (GH373)
- *Support* regular expressions in `read_table/read_csv` (GH364)
- *Added* `DataFrame.to_html` for writing DataFrame to HTML (GH387)
- *Added* support for `MaskedArray` data in DataFrame, masked values converted to `NaN` (GH396)
- *Added* `DataFrame.boxplot` function (GH368)
- *Can* pass extra args, kwds to `DataFrame.apply` (GH376)
- *Implement* `DataFrame.join` with vector on argument (GH312)
- *Added* `legend` boolean flag to `DataFrame.plot` (GH324)

- *Can* pass multiple levels to `stack` and `unstack` (GH370)
- *Can* pass multiple values columns to `pivot_table` (GH381)
- *Use* Series name in `GroupBy` for result index (GH363)
- *Added* `raw` option to `DataFrame.apply` for performance if only need ndarray (GH309)
- Added proper, tested weighted least squares to standard and panel OLS (GH303)

1.15.2 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)

1.16 v.0.5.0 (October 24, 2011)

1.16.1 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

1.16.2 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

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

1.17.1 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)

1.17.2 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

INSTALLATION

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](#)

Preliminary builds and installers on the [Pandas download page](#) .

2.2.2 Overview

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

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/xlwt**
 - 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

```
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>, it can be checked out using git and compiled / installed like so:

```
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 than a tarball or pypi.

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

```
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:

```
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:

```
$ nosetests pandas
.....
.....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 its 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.

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:

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

In [2]: 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]
...:

In [3]: pd.DataFrame.just_foo_cols = just_foo_cols # monkey-patch the DataFrame class

In [4]: df = pd.DataFrame([list(range(4))], columns=["A", "foo", "foozball", "bar"])

In [5]: df.just_foo_cols()
['foo', 'foozball']

In [6]: 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` $\geq$ 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:

```
In [16]: dt = ts.Date('Q', '1984Q3')

# sic
In [17]: dt
Out[17]: <Q-DEC : 1984Q1>

In [18]: dt.asfreq('D', 'start')
Out[18]: <D : 01-Jan-1984>

In [19]: dt.asfreq('D', 'end')
Out[19]: <D : 31-Mar-1984>

In [20]: dt + 3
Out[20]: <Q-DEC : 1984Q4>
```

`Date` and `DateArray` from `scikits.timeseries` have been reincarnated in `pandas` `Period` and `PeriodIndex`:

```
In [7]: pnow('D') # scikits.timeseries.now()
Period('2014-01-03', 'D')

In [8]: Period(year=2007, month=3, day=15, freq='D')
Period('2007-03-15', 'D')
```

```

In [9]: p = Period('1984Q3')

In [10]: p
Period('1984Q3', 'Q-DEC')

In [11]: p.asfreq('D', 'start')
Period('1984-07-01', 'D')

In [12]: p.asfreq('D', 'end')
Period('1984-09-30', 'D')

In [13]: (p + 3).asfreq('T') + 6 * 60 + 30
Period('1985-07-01 06:29', 'T')

In [14]: rng = period_range('1990', '2010', freq='A')

In [15]: rng

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

In [16]: rng.asfreq('B', 'end') - 3

<class 'pandas.tseries.period.PeriodIndex'>
freq: B
[1990-12-26, ..., 2010-12-28]
length: 21

```

scikits.timeseries	pandas	Notes
Date	Period	A span of time, from yearly through to secondly
DateArray	PeriodIndex	An array of timespans
convert	resample	Frequency conversion in scikits.timeseries
convert_to_annual	pivot_annual	currently supports up to daily frequency, see GH736

3.3.1 PeriodIndex / DateArray properties and functions

The scikits.timeseries DateArray had a number of information properties. Here are the pandas equivalents:

scikits.timeseries	pandas	Notes
get_steps	<code>np.diff(idx.values)</code>	
has_missing_dates	<code>not idx.is_full</code>	
is_full	<code>idx.is_full</code>	
is_valid	<code>idx.is_monotonic</code> and <code>idx.is_unique</code>	
is_chronological	<code>is_monotonic</code>	
<code>arr.sort_chronologically()</code>	<code>idx.order()</code>	

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 with scikits.timeseries:

```
In [17]: import scikits.timeseries as ts
```

```
In [18]: data = ts.time_series(np.random.randn(50), start_date='Jan-2000', freq='M')
```

```
In [19]: data
```

```
timeseries([ 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  0.4047  0.577 -1.715
 -1.0393 -0.3706 -1.1579 -1.3443  0.8449  1.0758 -0.109  1.6436 -1.4694
  0.357  -0.6746 -1.7769 -0.9689 -1.2945  0.4137  0.2767 -0.472 -0.014
 -0.3625 -0.0062 -0.9231  0.8957  0.8052],
 dates = [Jan-2014 ... Feb-2018],
 freq = M)
```

```
In [20]: data.convert('A', func=np.mean)
```

```
timeseries([-0.3945096205751429 -0.24462765889025218 -0.22163251299635775
 -0.4537726933838235  0.8504806638002349],
 dates = [2014 ... 2018],
 freq = A-DEC)
```

Here is the equivalent pandas code:

```
In [21]: rng = period_range('Jan-2000', periods=50, freq='M')
```

```
In [22]: data = Series(np.random.randn(50), index=rng)
```

```
In [23]: data
```

```
2000-01    -1.206412
2000-02     2.565646
2000-03     1.431256
2000-04     1.340309
2000-05    -1.170299
...
2003-09    -2.179861
2003-10    -1.369849
2003-11    -0.954208
2003-12     1.462696
2004-01    -1.743161
2004-02    -0.826591
Freq: M, Length: 50
```

```
In [24]: data.resample('A', how=np.mean)
```

```
2000     0.166630
2001    -0.114581
2002    -0.205961
2003    -0.235802
2004    -1.284876
Freq: A-DEC, dtype: float64
```

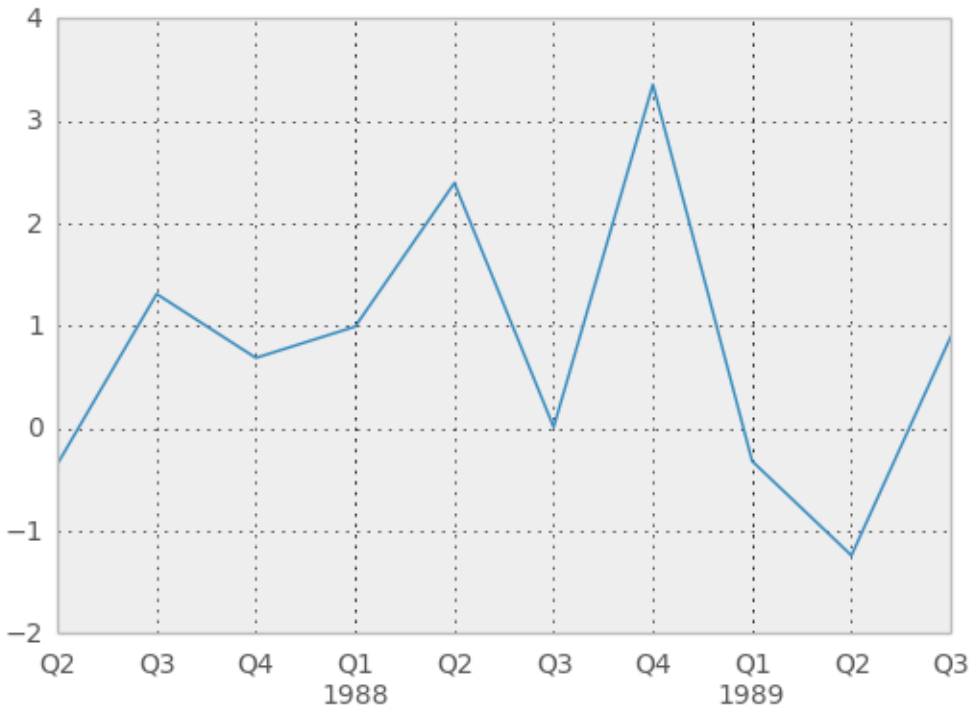
3.3.3 Plotting

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


```
In [25]: rng = period_range('1987Q2', periods=10, freq='Q-DEC')
```

```
In [26]: data = Series(np.random.randn(10), index=rng)
```

```
In [27]: plt.figure(); data.plot()
<matplotlib.axes.AxesSubplot at 0x8982a10>
```



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 [28]: rng = date_range('1/1/2000', periods=200, freq='D')
```

```
In [29]: data = Series(np.random.randn(200), index=rng)
```

```
In [30]: data[:10]
```

```
2000-01-01    -0.487602
2000-01-02    -0.082240
2000-01-03    -2.182937
2000-01-04     0.380396
2000-01-05     0.084844
2000-01-06     0.432390
2000-01-07     1.519970
2000-01-08    -0.493662
2000-01-09     0.600178
2000-01-10     0.274230
Freq: D, dtype: float64
```

In [31]: data.index

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-07-18 00:00:00]
Length: 200, Freq: D, Timezone: None
```

In [32]: data.resample('M', kind='period')

```
2000-01     0.163775
2000-02     0.026549
2000-03    -0.089563
2000-04    -0.079405
2000-05     0.160348
2000-06     0.101725
2000-07    -0.708770
Freq: M, dtype: float64
```

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

In [33]: rng = period_range('Jan-2000', periods=50, freq='M')

In [34]: data = Series(np.random.randn(50), index=rng)

In [35]: resampled = data.resample('A', kind='timestamp', convention='end')

In [36]: resampled.index

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-12-31 00:00:00, ..., 2004-12-31 00:00:00]
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 [37]: x = np.array(list(range(10)), '>i4') # big endian

In [38]: newx = x.byteswap().newbyteorder() # force native byteorder

In [39]: 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 will use the DataFrameModel class that handles the access to the DataFrame, and the DataFrameWidget, which is just a thin layer around the QTableView.

```
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)

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

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
1	Time-Series	Series with index containing datetimes
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns
3	Panel	General 3D labeled, also size-mutable array

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-contiguosness 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:

```
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

=====
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*. Customarily, 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
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
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00, ..., 2013-01-06 00:00:00]
Length: 6, Freq: D, Timezone: None
In [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
In [9]: df
```

```
           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
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
[6 rows x 4 columns]
```

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
```

```
           A           B  C  D  E
0  1 2013-01-02 00:00:00  1  3  foo
1  1 2013-01-02 00:00:00  1  3  foo
2  1 2013-01-02 00:00:00  1  3  foo
3  1 2013-01-02 00:00:00  1  3  foo
[4 rows x 5 columns]
```

Having specific *dtypes*

```
In [12]: df2.dtypes
```

```
A           float64
B    datetime64[ns]
C           float32
D           int32
E           object
dtype: object
```

If you're using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here's a subset of the attributes that will be completed:

```
In [13]: df2.<TAB>
```

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()
```

```
           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)
```

```
          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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00, ..., 2013-01-06 00:00:00]
Length: 6, Freq: D, Timezone: None
```

```
In [17]: df.columns
```

```
Index([u'A', u'B', u'C', u'D'], dtype='object')
```

```
In [18]: df.values
```

```
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()
```

```
          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
```

```
          2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A          0.469112   1.212112  -0.861849   0.721555  -0.424972  -0.673690
B          -0.282863  -0.173215  -2.104569  -0.706771   0.567020   0.113648
C          -1.509059   0.119209  -0.494929  -1.039575   0.276232  -1.478427
D          -1.135632  -1.044236   1.071804   0.271860  -1.087401   0.524988
[4 rows x 6 columns]
```

Sorting by an axis

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

```
          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')
```

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

```
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.

```
In [24]: df[0:3]
```

```
          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]
```

```
In [25]: df['20130102':'20130104']
```

```

           A           B           C           D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
[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

```
In [26]: df.loc[dates[0]]
```

```

A      0.469112
B     -0.282863
C     -1.509059
D     -1.135632
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label

```
In [27]: df.loc[:, ['A', 'B']]
```

```

           A           B
2013-01-01  0.469112 -0.282863
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
2013-01-06 -0.673690  0.113648
[6 rows x 2 columns]
```

Showing label slicing, both endpoints are *included*

```
In [28]: df.loc['20130102':'20130104', ['A', 'B']]
```

```

           A           B
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
[3 rows x 2 columns]
```

Reduction in the dimensions of the returned object

```
In [29]: df.loc['20130102', ['A', 'B']]
```

```

A      1.212112
B     -0.173215
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value

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

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

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

A    0.721555
B   -0.706771
C   -1.039575
D    0.271860
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python

```
In [33]: df.iloc[3:5, 0:2]

          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]]

          A          C
2013-01-02  1.212112  0.119209
2013-01-03 -0.861849 -0.494929
2013-01-05 -0.424972  0.276232
[3 rows x 2 columns]
```

For slicing rows explicitly

```
In [35]: df.iloc[1:3, :]

          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
[2 rows x 4 columns]
```

For slicing columns explicitly

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

          B          C
2013-01-01 -0.282863 -1.509059
2013-01-02 -0.173215  0.119209
2013-01-03 -2.104569 -0.494929
2013-01-04 -0.706771 -1.039575
```

```
2013-01-05    0.567020    0.276232
2013-01-06    0.113648   -1.478427
[6 rows x 2 columns]
```

For getting a value explicitly

```
In [37]: df.iloc[1,1]
-0.17321464905330858
```

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

```
In [38]: df.iat[1,1]
-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]
['e', 'f']
```

```
In [41]: x[8:10]
[]
```

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]
```

```
          A          B          C          D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
[3 rows x 4 columns]
```

A where operation for getting.

```
In [43]: df[df > 0]
```

```
          A          B          C          D
2013-01-01  0.469112      NaN      NaN      NaN
2013-01-02  1.212112      NaN  0.119209      NaN
2013-01-03      NaN      NaN      NaN  1.071804
2013-01-04  0.721555      NaN      NaN  0.271860
2013-01-05      NaN  0.567020  0.276232      NaN
2013-01-06      NaN  0.113648      NaN  0.524988
[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
```

```
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64
```

```
In [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
```

```
          A          B          C  D  F
2013-01-01  0.000000  0.000000 -1.509059  5 NaN
2013-01-02  1.212112 -0.173215  0.119209  5  1
2013-01-03 -0.861849 -2.104569 -0.494929  5  2
2013-01-04  0.721555 -0.706771 -1.039575  5  3
2013-01-05 -0.424972  0.567020  0.276232  5  4
2013-01-06 -0.673690  0.113648 -1.478427  5  5
[6 rows x 5 columns]
```

A where operation with setting.

```
In [51]: df2 = df.copy()
```

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

```
In [53]: df2
```

```
          A          B          C  D  F
2013-01-01  0.000000  0.000000 -1.509059 -5 NaN
2013-01-02 -1.212112 -0.173215 -0.119209 -5 -1
2013-01-03 -0.861849 -2.104569 -0.494929 -5 -2
2013-01-04 -0.721555 -0.706771 -1.039575 -5 -3
2013-01-05 -0.424972 -0.567020 -0.276232 -5 -4
2013-01-06 -0.673690 -0.113648 -1.478427 -5 -5
[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
```

```

           A          B          C  D  F  E
2013-01-01  0.000000  0.000000 -1.509059  5 NaN  1
2013-01-02  1.212112 -0.173215  0.119209  5  1  1
2013-01-03 -0.861849 -2.104569 -0.494929  5  2 NaN
2013-01-04  0.721555 -0.706771 -1.039575  5  3 NaN
[4 rows x 6 columns]
```

To drop any rows that have missing data.

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

```

           A          B          C  D  F  E
2013-01-02  1.212112 -0.173215  0.119209  5  1  1
[1 rows x 6 columns]
```

Filling missing data

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

```

           A          B          C  D  F  E
2013-01-01  0.000000  0.000000 -1.509059  5  5  1
2013-01-02  1.212112 -0.173215  0.119209  5  1  1
2013-01-03 -0.861849 -2.104569 -0.494929  5  2  5
2013-01-04  0.721555 -0.706771 -1.039575  5  3  5
[4 rows x 6 columns]
```

To get the boolean mask where values are nan

```
In [59]: pd.isnull(df1)
```

```

           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  False   True
2013-01-04  False  False  False  False  False   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()
```

```
A    -0.004474
B    -0.383981
C    -0.687758
D     5.000000
F     3.000000
dtype: float64
```

Same operation on the other axis

```
In [61]: df.mean(1)
```

```
2013-01-01    0.872735
2013-01-02    1.431621
2013-01-03    0.707731
2013-01-04    1.395042
2013-01-05    1.883656
2013-01-06    1.592306
Freq: D, dtype: float64
```

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

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

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

```
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')
```

```
          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)
```

```

      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())
```

```

A      2.073961
B      2.671590
C      1.785291
D      0.000000
F      NaN
dtype: float64
```

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
```

```

0      4
1      2
2      1
3      2
4      6
5      4
6      4
7      6
8      4
9      4
dtype: int64
```

```
In [69]: s.value_counts()
```

```

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()
```

```
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
```

```
      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]
```

```
# break it into pieces
```

```
In [74]: pieces = [df[:3], df[3:7], df[7:]]
```

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

```
      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
```

```
   key  lval
0  foo     1
1  foo     2
[2 rows x 2 columns]
```

```
In [79]: right
```

```
   key  rval
0  foo     4
1  foo     5
[2 rows x 2 columns]
```

```
In [80]: pd.merge(left, right, on='key')
```

```
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
[4 rows x 3 columns]
```

5.6.3 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
```

```
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758
[8 rows x 4 columns]
```

```
In [83]: s = df.iloc[3]
```

```
In [84]: df.append(s, ignore_index=True)
```

```
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
```

```
2 -1.577585    0.396823 -0.105381 -0.532532
3  1.453749    1.208843 -0.080952 -0.264610
4 -0.727965   -0.589346  0.339969 -0.693205
5 -0.339355    0.593616  0.884345  1.591431
6  0.141809    0.220390  0.435589  0.192451
7 -0.096701    0.803351  1.715071 -0.708758
8  1.453749    1.208843 -0.080952 -0.264610
[9 rows x 4 columns]
```

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
```

```
   A      B      C      D
0  foo  one -1.202872 -0.055224
1  bar  one -1.814470  2.395985
2  foo  two  1.018601  1.552825
3  bar three -0.595447  0.166599
4  foo  two  1.395433  0.047609
5  bar  two -0.392670 -0.136473
6  foo  one  0.007207 -0.561757
7  foo three  1.928123 -1.623033
[8 rows x 4 columns]
```

Grouping and then applying a function `sum` to the resulting groups.

```
In [87]: df.groupby('A').sum()
```

```
      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()
```

```
      C      D
A  B
bar one -2.802588  2.42611
foo two  3.146492 -0.63958
```

```

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

```

```

              A          B
first second
bar   one    0.029399 -0.542108
      two    0.282696 -0.087302
baz   one   -1.575170  1.771208
      two    0.816482  1.100230
[4 rows x 2 columns]

```

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

```

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

```

```

In [95]: stacked

```

```

first second A          B
bar   one    A    0.029399
      one    B   -0.542108
      two    A    0.282696
      two    B   -0.087302
baz   one    A   -1.575170
      one    B    1.771208
      two    A    0.816482
      two    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()
```

```
           A      B
first second
bar  one  0.029399 -0.542108
     two  0.282696 -0.087302
baz   one -1.575170  1.771208
     two  0.816482  1.100230
[4 rows x 2 columns]
```

```
In [97]: stacked.unstack(1)
```

```
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)
```

```
first      bar      baz
second
one  A  0.029399 -1.575170
     B -0.542108  1.771208
two  A  0.282696  0.816482
     B -0.087302  1.100230
[4 rows x 2 columns]
```

5.8.2 Pivot Tables

See the section on *Pivot Tables*.

```
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
```

```
      A  B    C      D      E
0  one  A  foo  1.418757 -0.179666
1  one  B  foo -1.879024  1.291836
2  two  C  foo  0.536826 -0.009614
3 three  A  bar  1.006160  0.392149
4  one  B  bar -0.029716  0.264599
5  one  C  bar -1.146178 -0.057409
6  two  A  foo  0.100900 -1.425638
7 three  B  foo -1.035018  1.024098
8  one  C  foo  0.314665 -0.106062
9  one  A  bar -0.773723  1.824375
10 two  B  bar -1.170653  0.595974
11 three  C  bar  0.648740  1.167115
[12 rows x 5 columns]
```


We can produce pivot tables from this data very easily:

```
In [101]: pd.pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
```

```
C          bar      foo
A  B
one A -0.773723  1.418757
    B -0.029716 -1.879024
    C -1.146178  0.314665
three A  1.006160      NaN
      B      NaN -1.035018
      C  0.648740      NaN
two  A      NaN  0.100900
      B -1.170653      NaN
      C      NaN  0.536826
[9 rows x 2 columns]
```

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')
```

```
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
```

```
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
```

```
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')
```

```
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
```

```
2012-01-31    -1.134623
2012-02-29    -1.561819
2012-03-31    -0.260838
2012-04-30     0.281957
2012-05-31     1.523962
Freq: M, dtype: float64
```

```
In [114]: ps = ts.to_period()
```

```
In [115]: ps
```

```
2012-01    -1.134623
2012-02    -1.561819
2012-03    -0.260838
2012-04     0.281957
2012-05     1.523962
Freq: M, dtype: float64
```

```
In [116]: ps.to_timestamp()
```

```
2012-01-01    -1.134623
2012-02-01    -1.561819
2012-03-01    -0.260838
2012-04-01     0.281957
2012-05-01     1.523962
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [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
```

```
In [120]: ts.head()
```

```
1990-03-01 09:00    -0.902937
```

```

1990-06-01 09:00    0.068159
1990-09-01 09:00   -0.057873
1990-12-01 09:00   -0.368204
1991-03-01 09:00   -1.144073
Freq: H, dtype: float64

```

5.10 Plotting

Plotting docs.

```
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()
<matplotlib.axes.AxesSubplot at 0x7697dd0>
```

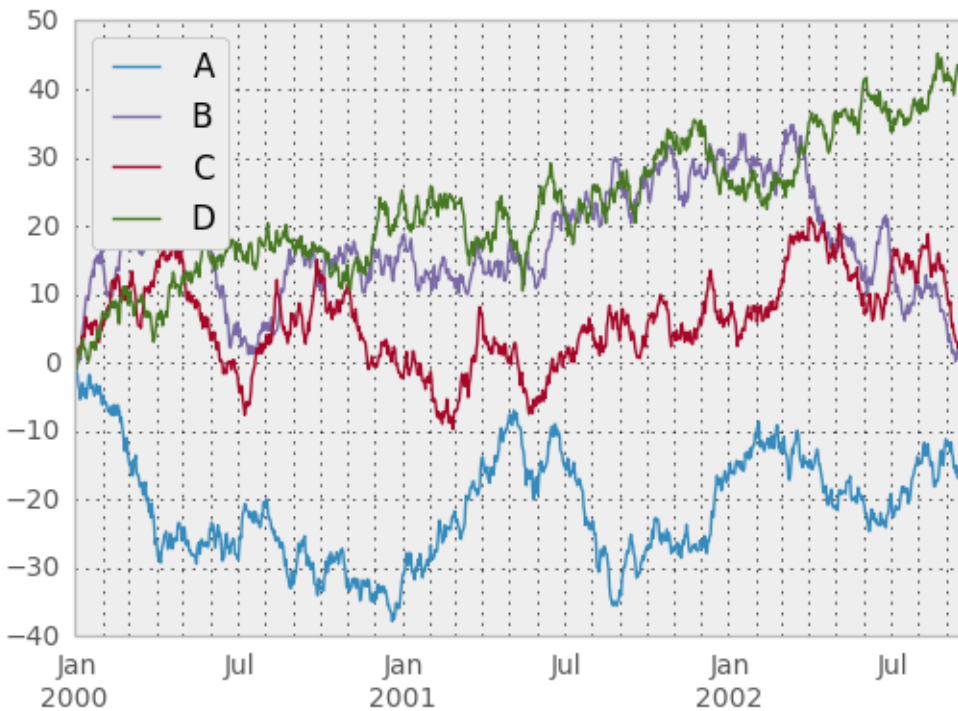


On DataFrame, plot is a convenience to plot all of the columns with labels:

```
In [124]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
.....:                      columns=['A', 'B', 'C', 'D'])
.....:
```

```
In [125]: df = df.cumsum()
```

```
In [126]: plt.figure(); df.plot(); plt.legend(loc='best')
<matplotlib.legend.Legend at 0x7b2c710>
```



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')
```

```

      Unnamed: 0      A      B      C      D
0  2000-01-01  0.266457 -0.399641 -0.219582  1.186860
1  2000-01-02 -1.170732 -0.345873  1.653061 -0.282953
2  2000-01-03 -1.734933  0.530468  2.060811 -0.515536
3  2000-01-04 -1.555121  1.452620  0.239859 -1.156896
4  2000-01-05  0.578117  0.511371  0.103552 -2.428202
5  2000-01-06  0.478344  0.449933 -0.741620 -1.962409
6  2000-01-07  1.235339 -0.091757 -1.543861 -1.084753
7  2000-01-08 -1.318492  0.003142 -3.863379 -0.791151
8  2000-01-09 -1.552842  1.292518 -4.772843 -0.471664
9  2000-01-10 -1.621025  0.074253 -5.866093 -0.162509
10 2000-01-11 -2.418239 -0.640980 -5.895733 -0.362802
11 2000-01-12 -3.350633 -0.358341 -5.917620 -0.444849
12 2000-01-13 -3.268737 -0.795976 -6.240176  0.497327
13 2000-01-14 -2.786138 -1.017654 -4.260442  0.631441
14 2000-01-15 -4.261365 -0.721180 -3.918269 -0.118960
...
[1000 rows x 5 columns]
```

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')
```

	A	B	C	D
2000-01-01	0.266457	-0.399641	-0.219582	1.186860
2000-01-02	-1.170732	-0.345873	1.653061	-0.282953
2000-01-03	-1.734933	0.530468	2.060811	-0.515536
2000-01-04	-1.555121	1.452620	0.239859	-1.156896
2000-01-05	0.578117	0.511371	0.103552	-2.428202
2000-01-06	0.478344	0.449933	-0.741620	-1.962409
2000-01-07	1.235339	-0.091757	-1.543861	-1.084753
2000-01-08	-1.318492	0.003142	-3.863379	-0.791151
2000-01-09	-1.552842	1.292518	-4.772843	-0.471664
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]

5.11.3 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'])
```

	A	B	C	D
2000-01-01	0.266457	-0.399641	-0.219582	1.186860
2000-01-02	-1.170732	-0.345873	1.653061	-0.282953
2000-01-03	-1.734933	0.530468	2.060811	-0.515536
2000-01-04	-1.555121	1.452620	0.239859	-1.156896
2000-01-05	0.578117	0.511371	0.103552	-2.428202
2000-01-06	0.478344	0.449933	-0.741620	-1.962409
2000-01-07	1.235339	-0.091757	-1.543861	-1.084753
2000-01-08	-1.318492	0.003142	-3.863379	-0.791151
2000-01-09	-1.552842	1.292518	-4.772843	-0.471664
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:

```
>>> 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.

COOKBOOK

This is a respository 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)

6.1 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

6.2 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`

6.3 MultiIndexing

The *multindexing* docs.

Creating a multi-index from a labeled frame

6.3.1 Arithmetic

Performing arithmetic with a multi-index that needs broadcastin

6.3.2 Slicing

Slicing a multi-index with xs

Slicing a multi-index with xs #2

Setting portions of a multi-index with xs

6.3.3 Sorting

Multi-index sorting

Partial Selection, the need for sortedness

6.3.4 Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

6.3.5 panelnd

The *panelnd* docs.

Construct a 5D panelnd

6.4 Missing Data

The *missing data* docs.

Fill forward a reversed timeseries

```
In [1]: df = pd.DataFrame(np.random.randn(6,1), index=pd.date_range('2013-08-01', periods=6, freq='B'))
```

```
In [2]: df.ix[3,'A'] = np.nan
```

```
In [3]: df
```

```

              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
[6 rows x 1 columns]
```

```
In [4]: df.reindex(df.index[::-1]).ffill()
```



```
                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
[6 rows x 1 columns]
```

cumsum reset at NaN values

6.4.1 Replace

Using replace with backrefs

6.5 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

6.5.1 Expanding Data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

6.5.2 Splitting

Splitting a frame

6.5.3 Pivot

The *Pivot* docs.

Partial sums and subtotals

Frequency table like plyr in R

6.5.4 Apply

Turning embeded lists into a multi-index frame

6.6 Timeseries

Between times

Using indexer between time

Vectorized Lookup

Turn a matrix with hours in columns and days in rows into a continous row sequence in the form of a time series. [How to rearrange a python pandas dataframe?](#)

6.6.1 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

6.7 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

6.8 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

6.9 Data In/Out

Performance comparison of SQL vs HDF5

6.9.1 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

6.9.2 SQL

The *SQL* docs

Reading from databases with SQL

6.9.3 Excel

The *Excel* docs

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

6.9.4 HDFStore

The *HDFStores* docs

Simple Queries with a Timestamp Index

Managing heterogeneous data using a linked multiple table hierarchy

Merging on-disk tables with millions of rows

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  
{ 'A': 10 }
```

6.10 Computation

Numerical integration (sample-based) of a time series

6.11 Miscellaneous

The *Timedeltas* docs.

Operating with timedeltas

Create timedeltas with date differences

Adding days to dates in a dataframe

6.12 Aliasing Axis Names

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

```
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
....:

In [11]: def clear_axis_alias(cls, axis, alias):
....:     if axis not in cls._AXIS_NUMBERS:
....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
....:     cls._AXIS_ALIASES.pop(alias, None)
....:

In [12]: set_axis_alias(DataFrame, 'columns', 'myaxis2')

In [13]: df2 = DataFrame(randn(3,2), columns=['c1', 'c2'], index=['i1', 'i2', 'i3'])

In [14]: df2.sum(axis='myaxis2')

i1    -0.499427
i2     0.966720
i3     0.174175
dtype: float64

In [15]: clear_axis_alias(DataFrame, 'columns', 'myaxis2')
```


INTRO TO DATA STRUCTURES

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import numpy and load pandas into your namespace:

```
In [1]: import numpy as np

# will use a lot in examples
In [2]: randn = np.random.randn

In [3]: from pandas import *
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

When using pandas, we recommend the following import convention:

```
import pandas as pd
```

7.1 Series

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

`Series` is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a `Series` is to call:

```
>>> s = Series(data, index=index)
```

Here, `data` can be many different things:

- a Python dict
- an `ndarray`
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data** is:

From ndarray

If data is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [4]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [5]: s
```

```
a    0.357
b   -0.675
c   -1.777
d   -0.969
e   -1.295
dtype: float64
```

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

```
In [7]: Series(randn(5))
```

```
0    0.414
1    0.277
2   -0.472
3   -0.014
4   -0.363
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)
```

```
a    0
b    1
c    2
dtype: float64
```

```
In [10]: Series(d, index=['b', 'c', 'd', 'a'])
```

```
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'])
```

```
a      5
b      5
c      5
d      5
e      5
dtype: float64
```

7.1.1 Series is ndarray-like

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

```
In [12]: s[0]
0.35702056413309086
```

```
In [13]: s[:3]
```

```
a      0.357
b     -0.675
c     -1.777
dtype: float64
```

```
In [14]: s[s > s.median()]
```

```
a      0.357
b     -0.675
dtype: float64
```

```
In [15]: s[[4, 3, 1]]
```

```
e     -1.295
d     -0.969
b     -0.675
dtype: float64
```

```
In [16]: np.exp(s)
```

```
a      1.429
b      0.509
c      0.169
d      0.379
e      0.274
dtype: float64
```

We will address array-based indexing in a separate [section](#).

7.1.2 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']
0.35702056413309086
```

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

```
In [19]: s

a      0.357
b     -0.675
c     -1.777
d     -0.969
e     12.000
dtype: float64
```

```
In [20]: 'e' in s
True
```

```
In [21]: 'f' in s
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)
nan
```

See also the [section on attribute access](#).

7.1.3 Vectorized operations and label alignment with Series

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

```
In [24]: s + s
```

```
a      0.714
b     -1.349
c     -3.554
d     -1.938
e     24.000
dtype: float64
```

```
In [25]: s * 2
```

```
a      0.714
b     -1.349
c     -3.554
d     -1.938
e     24.000
dtype: float64
```

```
In [26]: np.exp(s)
```

```
a      1.429
b      0.509
c      0.169
d      0.379
e    162754.791
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [27]: s[1:] + s[:-1]
```

```
a      NaN
b    -1.349
c    -3.554
d    -1.938
e      NaN
dtype: float64
```

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

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

7.1.4 Name attribute

Series can also have a name attribute:

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

```
In [29]: s
0    -0.006
1    -0.923
2     0.896
3     0.805
4    -1.206
Name: something, dtype: float64
```

```
In [30]: s.name
'something'
```

The Series name will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

7.2 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.

7.2.1 From dict of Series or dicts

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

```
In [31]: d = {'one' : Series([1., 2., 3.], index=['a', 'b', 'c']),
.....:       'two' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
.....:
```

```
In [32]: df = DataFrame(d)
```

```
In [33]: df
```

```
   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'])
```

```
   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'])
```

```
   two three
d    4   NaN
b    2   NaN
a    1   NaN
[3 rows x 2 columns]
```

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

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

7.2.2 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)
```

```
   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'])
```

```
   one  two
a    1    4
b    2    3
c    3    2
d    4    1
[4 rows x 2 columns]
```

7.2.3 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)
```

```
   A  B      C
0  1  2  Hello
1  2  3  World
[2 rows x 3 columns]
```

```
In [44]: DataFrame(data, index=['first', 'second'])
```

```
   A  B      C
```

```
first    1  2  Hello
second   2  3  World
[2 rows x 3 columns]
```

```
In [45]: DataFrame(data, columns=['C', 'A', 'B'])
```

```
      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.

7.2.4 From a list of dicts

```
In [46]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
```

```
In [47]: DataFrame(data2)
```

```
      a  b  c
0    1  2 NaN
1    5 10 20
[2 rows x 3 columns]
```

```
In [48]: DataFrame(data2, index=['first', 'second'])
```

```
      a  b  c
first  1  2 NaN
second 5 10 20
[2 rows x 3 columns]
```

```
In [49]: DataFrame(data2, columns=['a', 'b'])
```

```
      a  b
0    1  2
1    5 10
[2 rows x 2 columns]
```

7.2.5 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 `np.nan` for those values which are missing. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

7.2.6 Alternate Constructors

DataFrame.from_dict

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

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. Works analogously to the normal `DataFrame` constructor, except that index maybe be a specific field of the structured dtype to use as the index. For example:

```
In [50]: data
```

```
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

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

```
      A  B
C
Hello  1  2
World  2  3
[2 rows x 2 columns]
```

`DataFrame.from_items`

`DataFrame.from_items` works analogously to the form of the `dict` constructor that takes a sequence of (key, value) pairs, where the keys are column (or row, in the case of `orient='index'`) names, and the value are the column values (or row values). This can be useful for constructing a `DataFrame` with the columns in a particular order without having to pass an explicit list of columns:

```
In [52]: DataFrame.from_items([( 'A', [1, 2, 3]), ( 'B', [4, 5, 6])])
```

```
      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:

```
In [53]: DataFrame.from_items([( 'A', [1, 2, 3]), ( 'B', [4, 5, 6])],
.....:                        orient='index', columns=['one', 'two', 'three'])
.....:
```

```
      one  two  three
A      1   2     3
B      4   5     6
[2 rows x 3 columns]
```

7.2.7 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']
```

```
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
```

```
   one  two  three  flag
a    1    1      1  False
b    2    2      4  False
c    3    3      9   True
d   NaN    4     NaN  False
[4 rows x 4 columns]
```

Columns can be deleted or popped like with a dict:

```
In [58]: del df['two']
```

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

```
In [60]: df
```

```
   one  flag
a    1  False
b    2  False
c    3   True
d   NaN  False
[4 rows x 2 columns]
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [61]: df['foo'] = 'bar'
```

```
In [62]: df
```

```
   one  flag  foo
a    1  False  bar
b    2  False  bar
c    3   True  bar
d   NaN  False  bar
[4 rows x 3 columns]
```

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
```

```
   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
[4 rows x 4 columns]
```

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
```

```
   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
[4 rows x 5 columns]
```

7.2.8 Indexing / Selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	<code>df[col]</code>	Series
Select row by label	<code>df.loc[label]</code>	Series
Select row by integer location	<code>df.iloc[loc]</code>	Series
Slice rows	<code>df[5:10]</code>	DataFrame
Select rows by boolean vector	<code>df[bool_vec]</code>	DataFrame

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

```
In [67]: df.loc['b']
```

```
one          2
bar          2
flag        False
foo          bar
one_trunc     2
Name: b, dtype: object
```

```
In [68]: df.iloc[2]
```

```
one          3
bar          3
flag         True
foo          bar
one_trunc    NaN
Name: c, dtype: object
```

For a more exhaustive treatment of more sophisticated label-based indexing and slicing, see the [section on indexing](#). We will address the fundamentals of reindexing / conforming to new sets of labels in the [section on reindexing](#).

7.2.9 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
```

```
      A      B      C  D
0  2.969  1.276  1.642 NaN
1 -2.406 -0.959 -0.140 NaN
2  0.635 -1.820 -2.014 NaN
3 -1.782 -0.099  2.298 NaN
4  1.565  3.273 -2.197 NaN
5  1.351 -0.727 -1.315 NaN
6 -0.323 -1.714  0.688 NaN
7    NaN    NaN    NaN NaN
8    NaN    NaN    NaN NaN
9    NaN    NaN    NaN NaN
[10 rows x 4 columns]
```

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]
```

```
      A      B      C      D
0  0.000  0.000  0.000  0.000
1 -2.792 -1.020 -0.526  1.302
2 -3.393 -1.508 -2.528  2.300
3 -4.002 -2.845  0.268  2.194
4 -1.996 -0.555 -3.552  2.145
5 -4.572 -1.841 -1.419  1.716
6 -3.785 -2.658 -0.571 -0.111
7 -3.293 -1.553 -1.438  1.866
8 -2.224 -0.472 -2.451  0.550
9 -2.416 -2.164 -0.653  1.347
[10 rows x 4 columns]
```

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
```

```
      A      B      C
2000-01-01 -2.183  0.380  0.085
2000-01-02  0.432  1.520 -0.494
2000-01-03  0.600  0.274  0.133
2000-01-04 -0.024  2.410  1.451
2000-01-05  0.206 -0.252 -2.214
2000-01-06  1.063  1.266  0.299
2000-01-07 -0.864  0.408 -1.048
2000-01-08 -0.026 -0.988  0.094
[8 rows x 3 columns]
```

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

```
In [77]: df - df['A']
```

```

      A      B      C
2000-01-01  0  2.563  2.268
2000-01-02  0  1.088 -0.926
2000-01-03  0 -0.326 -0.467
2000-01-04  0  2.434  1.474
2000-01-05  0 -0.458 -2.420
2000-01-06  0  0.203 -0.764
2000-01-07  0  1.272 -0.184
2000-01-08  0 -0.963  0.120
[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
```

```

      A      B      C
2000-01-01 -8.915  3.902  2.424
2000-01-02  4.162  9.600 -0.468
2000-01-03  5.001  3.371  2.664
2000-01-04  1.882 14.051  9.253
2000-01-05  3.030  0.740 -9.068
2000-01-06  7.317  8.331  3.497
2000-01-07 -2.319  4.041 -3.240
2000-01-08  1.871 -2.942  2.470
[8 rows x 3 columns]

```

```
In [79]: 1 / df
```

```

      A      B      C
2000-01-01 -0.458  2.629 11.786
2000-01-02  2.313  0.658 -2.026
2000-01-03  1.666  3.647  7.525
2000-01-04 -42.215  0.415  0.689
2000-01-05  4.853 -3.970 -0.452
2000-01-06  0.940  0.790  3.340
2000-01-07 -1.158  2.450 -0.954
2000-01-08 -38.839 -1.012 10.632
[8 rows x 3 columns]

```

```
In [80]: df ** 4
```

```

      A      B      C
2000-01-01  2.271e+01  0.021  5.182e-05
2000-01-02  3.495e-02  5.338  5.939e-02
2000-01-03  1.298e-01  0.006  3.118e-04
2000-01-04  3.149e-07 33.744  4.427e+00
2000-01-05  1.803e-03  0.004  2.401e+01
2000-01-06  1.278e+00  2.570  8.032e-03

```

```
2000-01-07  5.568e-01  0.028  1.207e+00
2000-01-08  4.395e-07  0.954  7.826e-05
[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
```

```
      a      b
0  False False
1  False  True
2   True False
[3 rows x 2 columns]
```

```
In [84]: df1 | df2
```

```
      a      b
0   True  True
1   True  True
2   True  True
[3 rows x 2 columns]
```

```
In [85]: df1 ^ df2
```

```
      a      b
0   True  True
1   True False
2  False  True
[3 rows x 2 columns]
```

```
In [86]: ~df1
```

```
      a      b
0  False  True
1   True False
2  False False
[3 rows x 2 columns]
```

7.2.10 Transposing

To transpose, access the `T` attribute (also the `transpose` function), similar to an `ndarray`:

```
# only show the first 5 rows
```

```
In [87]: df[:5].T
```

```
      2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A      -2.183      0.432      0.600      -0.024      0.206
B       0.380      1.520      0.274      2.410     -0.252
C       0.085     -0.494      0.133      1.451     -2.214
[3 rows x 5 columns]
```

7.2.11 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:

```
In [88]: np.exp(df)
```

```

      A      B      C
2000-01-01  0.113  1.463  1.089
2000-01-02  1.541  4.572  0.610
2000-01-03  1.822  1.316  1.142
2000-01-04  0.977 11.136  4.265
2000-01-05  1.229  0.777  0.109
2000-01-06  2.896  3.547  1.349
2000-01-07  0.422  1.504  0.351
2000-01-08  0.975  0.372  1.099
[8 rows x 3 columns]
```

```
In [89]: np.asarray(df)
```

```

array([[ -2.1829,  0.3804,  0.0848],
       [ 0.4324,  1.52   , -0.4937],
       [ 0.6002,  0.2742,  0.1329],
       [-0.0237,  2.4102,  1.4505],
       [ 0.2061, -0.2519, -2.2136],
       [ 1.0633,  1.2661,  0.2994],
       [-0.8638,  0.4082, -1.0481],
       [-0.0257, -0.9884,  0.0941]])
```

The dot method on DataFrame implements matrix multiplication:

```
In [90]: df.T.dot(df)
```

```

      A      B      C
A  7.233  0.902  0.412
B  0.902 11.149  3.230
C  0.412  3.230  8.469
[3 rows x 3 columns]
```

Similarly, the dot method on Series implements dot product:

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

```
In [92]: s1.dot(s1)
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.

7.2.12 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  schilcu01  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 values
year        100  non-null values
stint       100  non-null values
team        100  non-null values
lg          100  non-null values
g           100  non-null values
ab          100  non-null values
r           100  non-null values
h           100  non-null values
X2b         100  non-null values
X3b         100  non-null values
hr          100  non-null values
rbi         100  non-null values
sb          100  non-null values
cs          100  non-null values
bb          100  non-null values
so          100  non-null values
ibb         100  non-null values
hbp         100  non-null values
sh          100  non-null values
sf          100  non-null values
gidp        100  non-null values
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  finlest01  2007     1  COL  NL   43  94  9  17   3   0   1
89480  embreal01  2007     1  OAK  AL    4   0  0   0   0   0   0
89481  edmonji01  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     2  NYN  NL   21  41  2   8   2   0   0
89495  coninje01  2007     1  CIN  NL   80 215 23  57  11   1   6
89497  clemereo02 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  cirilje01  2007     2  ARI  NL   28  40  6   8   4   0   0
89502  cirilje01  2007     1  MIN  AL   50 153 18  40   9   2   2
89521  bondsba01  2007     1  SFN  NL  126 340 75  94  14   0  28
89523  biggicr01  2007     1  HOU  NL  141 517 68 130  31   3  10
89525  benitar01  2007     2  FLO  NL   34   0  0   0   0   0   0
89526  benitar01  2007     1  SFN  NL   19   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))
```

```

      0      1      2      3      4      5      6  \
0  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374
1  1.126203 -0.977349  1.474071 -0.064034 -1.282782  0.781836 -1.071357
2  0.758527  1.729689 -0.964980 -0.845696 -1.340896  1.846883 -1.328865
      7      8      9     10     11
0 -0.034571 -2.484478 -0.281461  0.030711  0.109121
1  0.441153  2.353925  0.583787  0.221471 -0.744471
2  1.682706 -1.717693  0.888782  0.228440  0.901805
[3 rows x 12 columns]

```

You can change how much to print on a single row by setting the `line_width` option:

```
In [98]: set_option('line_width', 40) # default is 80
```

```
In [99]: DataFrame(randn(3, 12))
```

```

      0      1      2  \
0  1.171216  0.520260 -1.197071
1  0.473424 -0.242861 -0.014805
2 -0.496922  0.306389 -2.290613
      3      4      5  \
0 -1.066969 -0.303421 -0.858447
1 -0.284319  0.650776 -1.461665
2 -1.134623 -1.561819 -0.260838
      6      7      8  \
0  0.306996 -0.028665  0.384316
1 -1.137707 -0.891060 -0.693921
2  0.281957  1.523962 -0.902937
      9     10     11
0  1.574159  1.588931  0.476720
1  1.613616  0.464000  0.227371
2  0.068159 -0.057873 -0.368204
[3 rows x 12 columns]

```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

7.2.13 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:

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

```
In [101]: df
```

```

      foo1      foo2
0 -1.144073 -1.099248
1  0.861209  0.255269
2  0.800193  0.009750

```

```
3  0.782098  0.661084
4 -1.069094  0.379319
[5 rows x 2 columns]
```

```
In [102]: df.foo1
```

```
0    -1.144073
1     0.861209
2     0.800193
3     0.782098
4    -1.069094
Name: foo1, dtype: float64
```

The columns are also connected to the [IPython](#) completion mechanism so they can be tab-completed:

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

7.3 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:

7.3.1 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
```

```
<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
```

7.3.2 From dict of DataFrame objects

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



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

```
<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:

Parameter	Default	Description
<code>intersect</code>	<code>False</code>	drops elements whose indices do not align
<code>orient</code>	<code>items</code>	use <code>minor</code> to use DataFrames' columns as panel items

For example, compare to the construction above:

```
In [107]: Panel.from_dict(data, orient='minor')
```

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

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

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

```
In [109]: df
```

```
      a      b
0  foo  0.464794
1  bar -0.309337
2  baz -0.649593
[3 rows x 2 columns]
```

```
In [110]: data = {'item1': df, 'item2': df}
```

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

```
In [112]: panel['a']
```

```
      item1 item2
0    foo    foo
1    bar    bar
2    baz    baz
[3 rows x 2 columns]
```

```
In [113]: panel['b']
```

```
      item1      item2
0  0.464794  0.464794
```

```
1 -0.309337 -0.309337
2 -0.649593 -0.649593
[3 rows x 2 columns]
```

```
In [114]: panel['b'].dtypes
```

```
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.

7.3.3 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.

```
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()
```

```
<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
```

7.3.4 Item selection / addition / deletion

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

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

```
          A          B          C          D
2000-01-01 -0.008434  1.952541 -1.056652  0.533946
2000-01-02 -1.226970  0.040403 -0.507516 -0.230096
2000-01-03  0.394500 -1.934370 -1.652499  1.488753
2000-01-04 -0.896484  0.576897  1.146000  1.487349
2000-01-05  0.604603  2.121453  0.597701  0.563700
[5 rows x 4 columns]
```

```
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.

7.3.5 Transposing

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

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

<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
```

7.3.6 Indexing / Selection

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

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

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

           A           B           C           D
2000-01-01 -0.008434  1.952541 -1.056652  0.533946
2000-01-02 -1.226970  0.040403 -0.507516 -0.230096
2000-01-03  0.394500 -1.934370 -1.652499  1.488753
2000-01-04 -0.896484  0.576897  1.146000  1.487349
2000-01-05  0.604603  2.121453  0.597701  0.563700
[5 rows x 4 columns]
```

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

```
           Item1           Item2           Item3
A  0.394500 -2.461467 -0.160270
B -1.934370 -1.553902  1.244847
C -1.652499  2.015523 -0.819886
D  1.488753 -1.833722 -0.811875
[4 rows x 3 columns]
```

```
In [123]: wp.minor_axis
Index([u'A', u'B', u'C', u'D'], dtype='object')
```

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

```
           Item1           Item2           Item3
2000-01-01 -1.056652  1.375020 -0.768463
2000-01-02 -0.507516  0.377953 -1.342801
2000-01-03 -1.652499  2.015523 -0.819886
2000-01-04  1.146000  0.049307 23.242339
2000-01-05  0.597701  0.146111  4.090725
[5 rows x 3 columns]
```

7.3.7 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()
```

```
          A          B          C          D
2000-01-01 -0.008434  1.952541 -1.056652  0.533946
2000-01-02 -1.226970  0.040403 -0.507516 -0.230096
2000-01-03  0.394500 -1.934370 -1.652499  1.488753
2000-01-04 -0.896484  0.576897  1.146000  1.487349
2000-01-05  0.604603  2.121453  0.597701  0.563700
[5 rows x 4 columns]
```

```
In [126]: wp.reindex(items=['Item1'],minor=['B']).squeeze()
```

```
2000-01-01    1.952541
2000-01-02    0.040403
2000-01-03   -1.934370
2000-01-04    0.576897
2000-01-05    2.121453
Freq: D, Name: B, dtype: float64
```

7.3.8 Conversion to DataFrame

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

```
In [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()
```

```
          one          two          three
major  minor
2000-01-01 a    0.683758 -0.937731 -0.280853
          b   -0.643834 -1.537770  0.025653
          c    0.421287  0.555759 -1.386071
          d    1.032814 -2.277282  0.863937
2000-01-02 a   -1.290493 -0.390201  0.252462
          b    0.787872  1.207122  1.500571
          c    1.515707  0.178690  1.053202
          d   -0.276487 -1.004168 -2.338595
2000-01-03 a   -0.223762 -1.377627 -0.374279
          b    1.397431  0.499281 -2.359958
          c    1.503874 -1.405256 -1.157886
          d   -0.478905  0.162565 -0.551865
2000-01-04 a   -0.135950 -0.067785  1.592673
          b   -0.730327 -1.260006  1.559318
          c   -0.033277 -1.132896  1.562443
          d    0.281151 -2.006481  0.763264
2000-01-05 a   -1.298915  0.301016  0.162027
          b   -2.819487  0.059117 -0.902704
          c   -0.851985  1.138469  1.106010
          d   -1.106952 -2.400634 -0.199234
```

```
[20 rows x 3 columns]
```

7.4 Panel4D (Experimental)

Panel4D is a 4-Dimensional named container very much like a Panel, but having 4 named dimensions. It is intended as a test bed for more N-Dimensional named containers.

- **labels:** axis 0, each item corresponds to a Panel contained inside
- **items:** axis 1, each item corresponds to a DataFrame contained inside
- **major_axis:** axis 2, it is the **index** (rows) of each of the DataFrames
- **minor_axis:** axis 3, it is the **columns** of each of the DataFrames

Panel4D is a sub-class of Panel, so most methods that work on Panels are applicable to Panel4D. The following methods are disabled:

- `join` , `to_frame` , `to_excel` , `to_sparse` , `groupby`

Construction of Panel4D works in a very similar manner to a Panel

7.4.1 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
```

```
<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
```

7.4.2 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)
```

```
<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.

7.4.3 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']
```

```
<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']
```

```
<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']
```

```
      Label1      Label2
Item1  0.458265  1.288393
Item2 -1.531095 -0.388231
[2 rows x 2 columns]
```

4D -> Series

```
In [136]: p4d.ix[:, 0, 0, 'A']
```

```
Label1    0.458265
Label2    1.288393
Name: A, dtype: float64
```

7.4.4 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)
```

```
<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
```

7.5 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

<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, :]

<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)

<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
(1, 2, 2, 5, 4)

In [145]: p5d.ndim
```


ESSENTIAL BASIC FUNCTIONALITY

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'])
...:
...:
```

8.1 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()
```

```
0    -0.199038
1     1.095864
2    -0.200875
3     0.162291
4    -0.430489
dtype: float64
```

```
In [7]: long_series.tail(3)
```

```
997    -1.198693
998     1.238029
999    -1.344716
dtype: float64
```

8.2 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!

```
In [8]: df[:2]
```

```
          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]
```

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

```
In [10]: df
```

```
          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:

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

```
In [12]: df.values
```

```
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]])
```

```
In [13]: wp.values
```

```
array([[ -1.1181,  0.4313,  0.5547, -1.3336],
       [-0.3322, -0.4859,  1.7259,  1.7993],
       [-0.9689, -0.7795, -2.0007, -1.8666],
       [-1.1013,  1.9575,  0.0589,  0.7581],
```

```
[ 0.0766, -0.5485, -0.1605, -0.3778]],
[[ 0.2499, -0.3413, -0.2726, -0.2774],
 [-1.1029,  0.1003, -1.6028,  0.9201],
 [-0.6439,  0.0603, -0.4349, -0.4943],
 [ 0.738 ,  0.4516,  0.3341, -0.7871],
 [ 0.6514, -0.7419,  1.1939, -2.3958]]])
```

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.

8.3 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):

Operation	0.11.0 (ms)	Prior Vern (ms)	Ratio to Prior
df1 > df2	13.32	125.35	0.1063
df1 * df2	21.71	36.63	0.5928
df1 + df2	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section [Recommended Dependencies](#) for more installation info.

8.4 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.

8.4.1 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
```

```
      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')
```

```
      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)
```

```
      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')
```

```
      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)
```

```
      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]
```

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]: major_mean = wp.mean(axis='major')
```

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

```
      Item1      Item2
A -0.688773 -0.021497
B   0.114982 -0.094183
C   0.035674 -0.156470
D -0.204142 -0.606887
[4 rows x 2 columns]
```

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

```
<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...

8.4.2 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
```

```
      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 [27]: df2
```

```
      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
[4 rows x 3 columns]
```

```
In [28]: df + df2
```

```
      one      three      two
a -1.402736      NaN -0.174206
b   0.218666 -0.708719   1.275347
```

```
c -0.463233 -0.296773 -0.005333
d          NaN -0.334814  0.208088
[4 rows x 3 columns]
```

```
In [29]: df.add(df2, fill_value=0)
```

```
      one      three      two
a -1.402736  1.000000 -0.174206
b  0.218666 -0.708719  1.275347
c -0.463233 -0.296773 -0.005333
d          NaN -0.334814  0.208088
[4 rows x 3 columns]
```

8.4.3 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)
```

```
      one  three  two
a False False False
b False False False
c False False False
d False False False
[4 rows x 3 columns]
```

```
In [31]: df2.ne(df)
```

```
      one  three  two
a False  True  False
b False False  False
c False False  False
d  True  False  False
[4 rows x 3 columns]
```

These operations produce a pandas object the same type as the left-hand-side input that is of dtype `bool`. These boolean objects can be used in indexing operations, see [here](#)

8.4.4 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()
```

```
one      False
three    False
two      False
dtype: bool
```

```
In [33]: (df>0).any()
```

```
one      True
three    False
two      True
dtype: bool
```

You can reduce to a final boolean value.

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

You can test if a pandas object is empty, via the `empty` property.

```
In [35]: df.empty
False
```

```
In [36]: DataFrame(columns=list('ABC')).empty
True
```

To evaluate single-element pandas objects in a boolean context, use the method `.bool()`:

```
In [37]: Series([True]).bool()
True
```

```
In [38]: Series([False]).bool()
False
```

```
In [39]: DataFrame([[True]]).bool()
True
```

```
In [40]: DataFrame([[False]]).bool()
False
```

Warning: You might be tempted to do the following:

```
>>> if df:
...     ...
```

Or

```
>>> 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.

8.4.5 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 [41]: df1 = DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
.....:                  'B' : [np.nan, 2., 3., np.nan, 6.]})
.....:
```

```
In [42]: df2 = DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
.....:                  'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
```

```
.....:

In [43]: df1

   A  B
0  1 NaN
1 NaN 2
2  3  3
3  5 NaN
4 NaN 6
[5 rows x 2 columns]

In [44]: df2

   A  B
0  5 NaN
1  2 NaN
2  4  3
3 NaN 4
4  3  6
5  7  8
[6 rows x 2 columns]

In [45]: df1.combine_first(df2)

   A  B
0  1 NaN
1  2  2
2  3  3
3  5  4
4  3  6
5  7  8
[6 rows x 2 columns]
```

8.4.6 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:

```
In [46]: combiner = lambda x, y: np.where(isnull(x), y, x)

In [47]: df1.combine(df2, combiner)

   A  B
0  1 NaN
1  2  2
2  3  3
3  5  4
4  3  6
5  7  8
[6 rows x 2 columns]
```


8.5 Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*, and *Panel*. Most of these are aggregations (hence producing a lower-dimensional result) like **sum**, **mean**, and **quantile**, but some of them, like **cumsum** and **cumprod**, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like *ndarray*.{*sum*, *std*, ...}, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: “index” (axis=0, default), “columns” (axis=1)
- **Panel**: “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

In [48]: df

```

      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 [49]: df.mean(0)

```

one      -0.274551
three    -0.223384
two       0.162987
dtype: float64
```

In [50]: df.mean(1)

```

a    -0.394235
b     0.130882
c    -0.127557
d    -0.031682
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

In [51]: df.sum(0, skipna=False)

```

one      NaN
three     NaN
two      0.651948
dtype: float64
```

In [52]: df.sum(axis=1, skipna=True)

```

a    -0.788471
b     0.392647
c    -0.382670
d    -0.063363
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [53]: ts_stand = (df - df.mean()) / df.std()
```

```
In [54]: ts_stand.std()
```

```
one      1
three    1
two      1
dtype: float64
```

```
In [55]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)
```

```
In [56]: xs_stand.std(1)
```

```
a      1
b      1
c      1
d      1
dtype: float64
```

Note that methods like **cumsum** and **cumprod** preserve the location of NA values:

```
In [57]: df.cumsum()
```

```
      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](#).

Function	Description
count	Number of non-null observations
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation
median	Arithmetic median of values
min	Minimum
max	Maximum
mode	Mode
abs	Absolute Value
prod	Product of values
std	Unbiased standard deviation
var	Unbiased variance
skew	Unbiased skewness (3rd moment)
kurt	Unbiased kurtosis (4th moment)
quantile	Sample quantile (value at %)
cumsum	Cumulative sum
cumprod	Cumulative product
cummax	Cumulative maximum
cummin	Cumulative minimum

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```
In [58]: np.mean(df['one'])
-0.27455055654271204
```

```
In [59]: np.mean(df['one'].values)
nan
```

Series also has a method `nunique` which will return the number of unique non-null values:

```
In [60]: series = Series(randn(500))
```

```
In [61]: series[20:500] = np.nan
```

```
In [62]: series[10:20] = 5
```

```
In [63]: series.nunique()
11
```

8.5.1 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 [64]: series = Series(randn(1000))
```

```
In [65]: series[::2] = np.nan
```

```
In [66]: series.describe()
```

```
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 [67]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
```

```
In [68]: frame.ix[::2] = np.nan
```

```
In [69]: frame.describe()
```

```
count    a    b    c    d    e
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:

```
In [70]: s = Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])
```

```
In [71]: s.describe()
```

```
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`.

8.5.2 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 [72]: s1 = Series(randn(5))
```

```
In [73]: s1
```

```
0    -0.574018
1     0.668292
2     0.303418
3    -1.190271
4     0.138399
dtype: float64
```

```
In [74]: s1.idxmin(), s1.idxmax()
(3, 1)
```

```
In [75]: df1 = DataFrame(randn(5,3), columns=['A','B','C'])
```

```
In [76]: df1
```

```
      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 [77]: df1.idxmin(axis=0)
```

```
A      3
B      1
C      1
dtype: int64
```

```
In [78]: df1.idxmax(axis=1)
```

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

```
4      B
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin` and `idxmax` return the first matching index:

```
In [79]: df3 = DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))
```

```
In [80]: df3
```

```
      A
e     2
d     1
c     1
b     3
a  NaN
[5 rows x 1 columns]
```

```
In [81]: df3['A'].idxmin()
'd'
```

Note: `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

8.5.3 Value counts (histogramming) / Mode

The `value_counts` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [82]: data = np.random.randint(0, 7, size=50)
```

```
In [83]: data
```

```
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, 2, 3, 2, 3, 0, 1,
       2, 4, 5, 5])
```

```
In [84]: s = Series(data)
```

```
In [85]: s.value_counts()
```

```
3      11
2       9
4       8
1       8
5       5
0       5
6       4
dtype: int64
```

```
In [86]: value_counts(data)
```

```
3      11
2       9
4       8
1       8
5       5
```

```
0      5
6      4
dtype: int64
```

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```
In [87]: s5 = Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])
```

```
In [88]: s5.mode()
```

```
0      3
1      7
dtype: int64
```

```
In [89]: df5 = DataFrame({"A": np.random.randint(0, 7, size=50),
.....:                  "B": np.random.randint(-10, 15, size=50)})
.....:
```

```
In [90]: df5.mode()
```

```
   A   B
0  5  -4
1  6 NaN
[2 rows x 2 columns]
```

8.5.4 Discretization and quantiling

Continuous values can be discretized using the `cut` (bins based on values) and `qcut` (bins based on sample quantiles) functions:

```
In [91]: arr = np.random.randn(20)
```

```
In [92]: factor = cut(arr, 4)
```

```
In [93]: factor
```

```
(-0.886, -0.0912]
(-0.886, -0.0912]
(-0.886, -0.0912]
(1.493, 2.285]
(0.701, 1.493]
...
(-0.0912, 0.701]
(-0.886, -0.0912]
(0.701, 1.493]
(0.701, 1.493]
(-0.0912, 0.701]
(1.493, 2.285]
Levels (4): Index(['(-0.886, -0.0912]', '(-0.0912, 0.701]',
                  '(0.701, 1.493]', '(1.493, 2.285]'], dtype=object)
Length: 20
```

```
In [94]: factor = cut(arr, [-5, -1, 0, 1, 5])
```

```
In [95]: factor
```

```
(-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 [96]: arr = np.random.randn(30)

In [97]: factor = qcut(arr, [0, .25, .5, .75, 1])

In [98]: factor

[-1.861, -0.487]
(0.0554, 0.658]
(0.658, 2.259]
[-1.861, -0.487]
(0.658, 2.259]
...
(0.0554, 0.658]
(0.0554, 0.658]
(0.658, 2.259]
[-1.861, -0.487]
(0.0554, 0.658]
(-0.487, 0.0554]
Levels (4): Index(['[-1.861, -0.487]', '(-0.487, 0.0554]',
                  '(0.0554, 0.658]', '(0.658, 2.259]'], dtype=object)
Length: 30

In [99]: value_counts(factor)

(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 [100]: arr = np.random.randn(20)

In [101]: factor = cut(arr, [-np.inf, 0, np.inf])

In [102]: factor

(0, inf]
(0, inf]
(-inf, 0]
(0, inf]
```

```
(-inf, 0]
...
(-inf, 0]
(0, inf]
(0, inf]
(-inf, 0]
(0, inf]
(-inf, 0]
Levels (2): Index(['(-inf, 0]', '(0, inf]'], dtype=object)
Length: 20
```

8.6 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:

```
In [103]: df.apply(np.mean)
```

```
one      NaN
three    NaN
two      0.162987
dtype: float64
```

```
In [104]: df.apply(np.mean, axis=1)
```

```
a    -0.394235
b     0.130882
c    -0.127557
d    -0.031682
dtype: float64
```

```
In [105]: df.apply(lambda x: x.max() - x.min())
```

```
one      NaN
three    NaN
two      0.724777
dtype: float64
```

```
In [106]: df.apply(np.cumsum)
```

```
      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 [107]: df.apply(np.exp)
```

```
      one      three      two
a  0.495907      NaN  0.916583
b  1.115534  0.701623  1.892074
c  0.793250  0.862098  0.997337
d      NaN  0.845855  1.109649
[4 rows x 3 columns]
```


Depending on the return type of the function passed to `apply`, the result will either be of lower dimension or the same dimension.

`apply` combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [108]: tsdf = DataFrame(randn(1000, 3), columns=['A', 'B', 'C'],
.....:                    index=date_range('1/1/2000', periods=1000))
.....:
```

```
In [109]: tsdf.apply(lambda x: x.idxmax())
```

```
A    2002-08-19 00:00:00
B    2000-11-30 00:00:00
C    2002-01-10 00:00:00
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the `apply` method. For instance, consider the following function you would like to apply:

```
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [110]: tsdf
```

```
      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
[10 rows x 3 columns]
```

```
In [111]: tsdf.apply(Series.interpolate)
```

```
      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
2000-01-07  0.872963  0.595784  1.249585
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
[10 rows x 3 columns]
```

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.

8.6.1 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:

```
In [112]: df4
```

```
      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 [113]: f = lambda x: len(str(x))
```

```
In [114]: df4['one'].map(f)
```

```
a    15
b    14
c    15
d     3
Name: one, dtype: int64
```

```
In [115]: df4.applymap(f)
```

```
      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](#):

```
In [116]: s = Series(['six', 'seven', 'six', 'seven', 'six'],
.....:               index=['a', 'b', 'c', 'd', 'e'])
.....:
```

```
In [117]: t = Series({'six' : 6., 'seven' : 7.})
```

```
In [118]: s
```

```
a      six
b     seven
c      six
d     seven
e      six
```

```
dtype: object
```

```
In [119]: s.map(t)
```

```
a      6
b      7
c      6
d      7
e      6
dtype: float64
```

8.7 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 [120]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [121]: s
```

```
a      0.166882
b     -1.759496
c      0.901336
d     -0.317478
e     -0.681335
dtype: float64
```

```
In [122]: s.reindex(['e', 'b', 'f', 'd'])
```

```
e     -0.681335
b     -1.759496
f              NaN
d     -0.317478
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 [123]: df
```

```
      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 [124]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
```

```
      three      two      one
c -0.148387 -0.002666 -0.231617
f      NaN      NaN      NaN
b -0.354359  0.637674  0.109333
[3 rows x 3 columns]
```

For convenience, you may utilize the `reindex_axis` method, which takes the labels and a keyword `axis` parameter.

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a `Series` and a `DataFrame`, the following can be done:

```
In [125]: rs = s.reindex(df.index)
```

```
In [126]: rs
```

```
a    0.166882
b   -1.759496
c    0.901336
d   -0.317478
dtype: float64
```

```
In [127]: rs.index is df.index
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.

8.7.1 Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like` method is available to make this simpler:

```
In [128]: df2
```

```
      one      two
a -0.701368 -0.087103
b  0.109333  0.637674
c -0.231617 -0.002666
[3 rows x 2 columns]
```

```
In [129]: df3
```

```
      one      two
a -0.426817 -0.269738
b  0.383883  0.455039
c  0.042934 -0.185301
```

```
[3 rows x 2 columns]
```

```
In [130]: df.reindex_like(df2)
```

```
      one      two
a -0.701368 -0.087103
b  0.109333  0.637674
c -0.231617 -0.002666
[3 rows x 2 columns]
```

8.7.2 Reindexing with `reindex_axis`

8.7.3 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 [131]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [132]: s1 = s[:4]
```

```
In [133]: s2 = s[1:]
```

```
In [134]: s1.align(s2)
```

```
(a    -0.597361
b    -1.514940
c    -1.640398
d    -1.130643
e         NaN
dtype: float64,
a         NaN
b    -1.514940
c    -1.640398
d    -1.130643
e    -0.245890
dtype: float64)
```

```
In [135]: s1.align(s2, join='inner')
```

```
(b    -1.514940
c    -1.640398
d    -1.130643
dtype: float64,
b    -1.514940
c    -1.640398
d    -1.130643
dtype: float64)
```

```
In [136]: s1.align(s2, join='left')
```

```
(a    -0.597361
b    -1.514940
c    -1.640398
d    -1.130643
dtype: float64,
a         NaN
b    -1.514940
c    -1.640398
d    -1.130643
dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [137]: df.align(df2, join='inner')
```

```
(      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 [138]: df.align(df2, join='inner', axis=0)
```

```
(      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 [139]: df.align(df2.ix[0], axis=1)
```

```
(      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
three         NaN
two      -0.087103
Name: a, dtype: float64)
```

8.7.4 Filling while reindexing

`reindex` takes an optional parameter `method` which is a filling method chosen from the following table:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward

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 [140]: rng = date_range('1/3/2000', periods=8)
```

```
In [141]: ts = Series(randn(8), index=rng)
```

```
In [142]: ts2 = ts[[0, 3, 6]]
```

```
In [143]: ts
```

```
2000-01-03    -1.200639
2000-01-04    -1.872993
2000-01-05     0.825210
2000-01-06    -0.392715
2000-01-07    -1.994150
2000-01-08     0.174260
2000-01-09    -0.581163
2000-01-10     0.087916
Freq: D, dtype: float64
```

```
In [144]: ts2
```

```
2000-01-03    -1.200639
2000-01-06    -0.392715
2000-01-09    -0.581163
dtype: float64
```

```
In [145]: ts2.reindex(ts.index)
```

```
2000-01-03    -1.200639
2000-01-04         NaN
2000-01-05         NaN
2000-01-06    -0.392715
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    -0.581163
2000-01-10         NaN
Freq: D, dtype: float64
```

```
In [146]: ts2.reindex(ts.index, method='ffill')
```

```
2000-01-03    -1.200639
2000-01-04    -1.200639
2000-01-05    -1.200639
2000-01-06    -0.392715
2000-01-07    -0.392715
```

```
2000-01-08    -0.392715
2000-01-09    -0.581163
2000-01-10    -0.581163
Freq: D, dtype: float64
```

```
In [147]: ts2.reindex(ts.index, method='bfill')
```

```
2000-01-03    -1.200639
2000-01-04    -0.392715
2000-01-05    -0.392715
2000-01-06    -0.392715
2000-01-07    -0.581163
2000-01-08    -0.581163
2000-01-09    -0.581163
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 [148]: ts2.reindex(ts.index).fillna(method='ffill')
```

```
2000-01-03    -1.200639
2000-01-04    -1.200639
2000-01-05    -1.200639
2000-01-06    -0.392715
2000-01-07    -0.392715
2000-01-08    -0.392715
2000-01-09    -0.581163
2000-01-10    -0.581163
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.

8.7.5 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 [149]: df
```

```
      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 [150]: df.drop(['a', 'd'], axis=0)
```

```
      one      three      two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
[2 rows x 3 columns]
```

```
In [151]: df.drop(['one'], axis=1)
```



```

      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 [152]: df.reindex(df.index - ['a', 'd'])
```

```

      one      three      two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
[2 rows x 3 columns]

```

8.7.6 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 [153]: s
```

```

a    -0.597361
b    -1.514940
c    -1.640398
d    -1.130643
e    -0.245890
dtype: float64

```

```
In [154]: s.rename(str.upper)
```

```

A    -0.597361
B    -1.514940
C    -1.640398
D    -1.130643
E    -0.245890
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 [155]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
.....:               index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
.....:
```

```

      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.

8.8 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 [156]: for col in df:
.....:     print(col)
.....:
one
three
two
```

8.8.1 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 [157]: 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 -0.494305
2000-01-04  0.737973  0.451632  0.334124 -0.787062
2000-01-05  0.651396 -0.741919  1.193881 -2.395763
[5 rows x 4 columns]
```

8.8.2 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 [158]: for row_index, row in df2.iterrows():
.....:     print('%s\n%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 [159]: df2 = DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [160]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6
[3 rows x 2 columns]

In [161]: print(df2.T)
   0  1  2
x  1  2  3
y  4  5  6
[2 rows x 3 columns]

In [162]: df2_t = DataFrame(dict((idx, values) for idx, values in df2.iterrows()))

In [163]: print(df2_t)
   0  1  2
x  1  2  3
y  4  5  6
[2 rows x 3 columns]
```

Note: `iterrows` does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [164]: df_iter = DataFrame([[1, 1.0]], columns=['x', 'y'])

In [165]: row = next(df_iter.iterrows())[1]

In [166]: print(row['x'].dtype)
float64

In [167]: print(df_iter['x'].dtype)
int64
```

8.8.3 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,

```
In [168]: for r in df2.itertuples():
.....:     print(r)
.....:
(0, 1, 4)
(1, 2, 5)
(2, 3, 6)
```

8.9 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:

8.9.1 Splitting and Replacing Strings

```
In [169]: s = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
```

```
In [170]: s.str.lower()
```

```
0      a
1      b
2      c
3    aaba
4    baca
5     NaN
6    caba
7    dog
8    cat
dtype: object
```

```
In [171]: s.str.upper()
```

```
0      A
1      B
2      C
3    AABA
4    BACA
5     NaN
6    CABA
7    DOG
8    CAT
dtype: object
```

```
In [172]: s.str.len()
```

```
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 [173]: s2 = Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])
```

```
In [174]: s2.str.split('_')
```

```
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 [175]: s2.str.split('_').str.get(1)
```

```
0      b
1      d
2     NaN
3      g
dtype: object
```

```
In [176]: s2.str.split('_').str[1]
```

```
0      b
1      d
2     NaN
3      g
dtype: object
```

Methods like `replace` and `findall` take regular expressions, too:

```
In [177]: s3 = Series(['A', 'B', 'C', 'Aaba', 'Baca',
.....:                '', np.nan, 'CABA', 'dog', 'cat'])
.....:
```

```
In [178]: s3
```

```
0      A
1      B
2      C
3     Aaba
4     Baca
5
6     NaN
7     CABA
8     dog
9     cat
dtype: object
```

```
In [179]: s3.str.replace('^a|dog', 'XX-XX ', case=False)
```

```
0          A
1          B
2          C
3  XX-XX ba
4  XX-XX ca
5
6         NaN
7  XX-XX BA
8    XX-XX
9  XX-XX t
dtype: object
```

8.9.2 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.

```
In [180]: Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)')
```

```
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.

```
In [181]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
```

```
   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`'s. Thus, a Series of messy strings can be "converted" into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects.

Named groups like

```
In [182]: Series(['a1', 'b2', 'c3']).str.extract('(P<letter>[ab])(P<digit>\d)')
```

```
   letter digit
0      a      1
1      b      2
2    NaN    NaN
[3 rows x 2 columns]
```

and optional groups like

```
In [183]: Series(['a1', 'b2', '3']).str.extract('(P<letter>[ab])?(P<digit>\d)')
```

```
   letter digit
0      a      1
1      b      2
```

```
2      NaN      3
[3 rows x 2 columns]
```

can also be used.

8.9.3 Testing for Strings that Match or Contain a Pattern

In previous versions, *extracting* match groups was accomplished by `match`, which returned a not-so-convenient Series of tuples. Starting in version 0.14, the default behavior of `match` will change. It will return a boolean indexer, analagous to the method `contains`.

The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match` while `contains` relies on `re.search`.

In version 0.13, `match` performs its old, deprecated behavior by default, but the new behavior is available through the keyword argument `as_indexer=True`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```
In [184]: s4 = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
```

```
In [185]: s4.str.contains('A', na=False)
```

```
0      True
1     False
2     False
3      True
4     False
5     False
6      True
7     False
8     False
dtype: bool
```

Method	Description
cat	Concatenate strings
split	Split strings on delimiter
get	Index into each element (retrieve i-th element)
join	Join strings in each element of the Series with passed separator
contains	Return boolean array if each string contains pattern/regex
replace	Replace occurrences of pattern/regex with some other string
repeat	Duplicate values (<code>s.str.repeat(3)</code> equivalent to <code>x * 3</code>)
pad	Add whitespace to left, right, or both sides of strings
center	Equivalent to <code>pad(side='both')</code>
slice	Slice each string in the Series
slice_replace	Replace slice in each string with passed value
count	Count occurrences of pattern
startswith	Equivalent to <code>str.startswith(pat)</code> for each element
endswith	Equivalent to <code>str.endswith(pat)</code> for each element
findall	Compute list of all occurrences of pattern/regex for each string
match	Call <code>re.match</code> on each element, returning matched groups as list
extract	Call <code>re.match</code> on each element, as <code>match</code> does, but return matched groups as strings for convenience.
len	Compute string lengths
strip	Equivalent to <code>str.strip</code>
rstrip	Equivalent to <code>str.rstrip</code>
lstrip	Equivalent to <code>str.lstrip</code>
lower	Equivalent to <code>str.lower</code>
upper	Equivalent to <code>str.upper</code>

8.10 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 [186]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
.....:                               columns=['three', 'two', 'one'])
.....:
```

```
In [187]: unsorted_df.sort_index()
```

```
      three      two      one
a      NaN -0.087103 -0.701368
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 [188]: unsorted_df.sort_index(ascending=False)
```

```
      three      two      one
d -0.167407  0.104044      NaN
c -0.148387 -0.002666 -0.231617
b -0.354359  0.637674  0.109333
a      NaN -0.087103 -0.701368
[4 rows x 3 columns]
```

```
In [189]: unsorted_df.sort_index(axis=1)
```



```

      one      three      two
a -0.701368      NaN -0.087103
d      NaN -0.167407  0.104044
c -0.231617 -0.148387 -0.002666
b  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 [190]: df1 = DataFrame({'one': [2, 1, 1, 1], 'two': [1, 3, 2, 4], 'three': [5, 4, 3, 2]})
```

```
In [191]: df1.sort_index(by='two')
```

```

      one  three  two
0      2      5    1
2      1      3    2
1      1      4    3
3      1      2    4
[4 rows x 3 columns]

```

The `by` argument can take a list of column names, e.g.:

```
In [192]: df1[['one', 'two', 'three']].sort_index(by=['one', 'two'])
```

```

      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 [193]: s[2] = np.nan
```

```
In [194]: s.order()
```

```

0      A
3    Aaba
1      B
4    Baca
6    CABA
8    cat
7    dog
2    NaN
5    NaN
dtype: object

```

```
In [195]: s.order(na_last=False)
```

```

2    NaN
5    NaN
0      A
3    Aaba
1      B
4    Baca
6    CABA

```

```
8      cat
7      dog
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`.

8.11 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.

8.12 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.

```
In [196]: 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 [197]: dft
```

```
      A    B    C      D    E    F    G
0  0.254252  1  foo 2001-01-02 00:00:00  1  False  1
1  0.615858  1  foo 2001-01-02 00:00:00  1  False  1
2  0.410404  1  foo 2001-01-02 00:00:00  1  False  1
[3 rows x 7 columns]
```

```
In [198]: dft.dtypes
```

```
A      float64
B      int64
C      object
D  datetime64[ns]
```

```
E          float32
F          bool
G          int8
dtype: object
```

On a Series use the dtype method.

```
In [199]: dft['A'].dtype
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).

```
# these ints are coerced to floats
In [200]: Series([1, 2, 3, 4, 5, 6.])

0    1
1    2
2    3
3    4
4    5
5    6
dtype: float64
```

```
# string data forces an ``object`` dtype
In [201]: Series([1, 2, 3, 6., 'foo'])

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 [202]: dft.get_dtype_counts()

bool          1
datetime64[ns] 1
float32        1
float64        1
int64          1
int8           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 [203]: df1 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')
```

```
In [204]: df1
```

```
      A
0  0.416971
1  0.635240
2  1.500829
```

```
3  0.990323
4  1.090392
5 -1.349864
6 -0.235045
7 -0.719088
[8 rows x 1 columns]
```

```
In [205]: df1.dtypes
```

```
A      float32
dtype: object
```

```
In [206]: df2 = DataFrame(dict( A = Series(randn(8),dtype='float16'),
.....:                          B = Series(randn(8)),
.....:                          C = Series(np.array(randn(8),dtype='uint8')) ))
.....:
```

```
In [207]: df2
```

```
      A      B      C
0 -0.603516 -0.592695    0
1  1.252930  0.325837    1
2 -0.797852  0.510273    0
3 -0.179565  0.337502  255
4  0.701660 -0.670413    0
5 -0.204956  0.291389    0
6 -0.971191  0.437209    1
7 -1.087891 -0.351341    0
[8 rows x 3 columns]
```

```
In [208]: df2.dtypes
```

```
A      float16
B      float64
C         uint8
dtype: object
```

8.12.1 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 [209]: DataFrame([1, 2], columns=['a']).dtypes
```

```
a      int64
dtype: object
```

```
In [210]: DataFrame({'a': [1, 2]}).dtypes
```

```
a      int64
dtype: object
```

```
In [211]: DataFrame({'a': 1 }, index=list(range(2))).dtypes
```

```
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 [212]: frame = DataFrame(np.array([1, 2]))
```

8.12.2 upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (say `int` to `float`)

```
In [213]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```

```
In [214]: df3
```

```
      A      B      C
0 -0.186544 -0.592695  0
1  1.888170  0.325837  1
2  0.702977  0.510273  0
3  0.810758  0.337502 255
4  1.792053 -0.670413  0
5 -1.554820  0.291389  0
6 -1.206237  0.437209  1
7 -1.806979 -0.351341  0
[8 rows x 3 columns]
```

```
In [215]: df3.dtypes
```

```
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 accomodate **ALL** of the types in the resulting homogenous dtype numpy array. This can force some *upcasting*.

```
In [216]: df3.values.dtype
dtype('float64')
```

8.12.3 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 [217]: df3
```

```
      A      B      C
0 -0.186544 -0.592695  0
1  1.888170  0.325837  1
2  0.702977  0.510273  0
3  0.810758  0.337502 255
4  1.792053 -0.670413  0
5 -1.554820  0.291389  0
```

```
6 -1.206237  0.437209    1
7 -1.806979 -0.351341    0
[8 rows x 3 columns]
```

```
In [218]: df3.dtypes
```

```
A    float32
B    float64
C    float64
dtype: object
```

```
# conversion of dtypes
```

```
In [219]: df3.astype('float32').dtypes
```

```
A    float32
B    float32
C    float32
dtype: object
```

8.12.4 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 [220]: df3['D'] = '1.'
```

```
In [221]: df3['E'] = '1'
```

```
In [222]: df3.convert_objects(convert_numeric=True).dtypes
```

```
A    float32
B    float64
C    float64
D    float64
E     int64
dtype: object
```

```
# same, but specific dtype conversion
```

```
In [223]: df3['D'] = df3['D'].astype('float16')
```

```
In [224]: df3['E'] = df3['E'].astype('int32')
```

```
In [225]: df3.dtypes
```

```
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 datetimelike 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 [226]: s = Series([datetime(2001,1,1,0,0),
.....:               'foo', 1.0, 1, Timestamp('20010104'),
.....:               '20010105'], dtype='O')
.....:
```

```
In [227]: s
```

```
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 [228]: s.convert_objects(convert_dates='coerce')
```

```
0    2001-01-01 00:00:00
1                NaT
2                NaT
3                NaT
4    2001-01-04 00:00:00
5    2001-01-05 00:00:00
dtype: 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.

8.12.5 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 [229]: dfi = df3.astype('int32')
```

```
In [230]: dfi['E'] = 1
```

```
In [231]: dfi
```

```
   A  B   C  D  E
0  0  0   0  1  1
1  1  0   1  1  1
2  0  0   0  1  1
3  0  0 255  1  1
4  1  0   0  1  1
5 -1  0   0  1  1
6 -1  0   1  1  1
7 -1  0   0  1  1
[8 rows x 5 columns]
```

```
In [232]: dfi.dtypes
```

```
A    int32
B    int32
C    int32
D    int32
E    int64
```

```
dtype: object
```

```
In [233]: casted = dfi[dfi>0]
```

```
In [234]: casted
```

```
      A   B   C   D   E
0  NaN NaN NaN   1   1
1    1 NaN   1   1   1
2  NaN NaN NaN   1   1
3  NaN NaN 255   1   1
4    1 NaN NaN   1   1
5  NaN NaN NaN   1   1
6  NaN NaN   1   1   1
7  NaN NaN NaN   1   1
[8 rows x 5 columns]
```

```
In [235]: casted.dtypes
```

```
A      float64
B      float64
C      float64
D       int32
E       int64
dtype: object
```

While float dtypes are unchanged.

```
In [236]: dfa = df3.copy()
```

```
In [237]: dfa['A'] = dfa['A'].astype('float32')
```

```
In [238]: dfa.dtypes
```

```
A      float32
B      float64
C      float64
D      float16
E       int32
dtype: object
```

```
In [239]: casted = dfa[df2>0]
```

```
In [240]: casted
```

```
      A      B   C   D   E
0    NaN    NaN NaN NaN NaN
1 1.888170 0.325837   1 NaN NaN
2    NaN 0.510273 NaN NaN NaN
3    NaN 0.337502 255 NaN NaN
4 1.792053    NaN NaN NaN NaN
5    NaN 0.291389 NaN NaN NaN
6    NaN 0.437209   1 NaN NaN
7    NaN    NaN NaN NaN NaN
[8 rows x 5 columns]
```

```
In [241]: casted.dtypes
```



```
A    float32
B    float64
C    float64
D    float16
E    float64
dtype: object
```

8.13 Working with package options

New in version 0.10.1. Pandas has an options system that let's you customize some aspects of it's 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:

```
In [242]: import pandas as pd
```

```
In [243]: pd.options.display.max_rows
15
```

```
In [244]: pd.options.display.max_rows = 999
```

```
In [245]: pd.options.display.max_rows
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 :

```
In [246]: get_option("display.max_rows")
999
```

```
In [247]: set_option("display.max_rows", 101)
```

```
In [248]: get_option("display.max_rows")
101
```

```
In [249]: set_option("max_r", 102)
```

```
In [250]: get_option("display.max_rows")
102
```

The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```
In [251]: 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 [252]: 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.

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

Deprecated.
(Deprecated)

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 than 'max_seq_items' will be printed. If items are omitted, they will be denoted by the addition of "..." to the resulting string.

If set to None, the number of items to be printed is unlimited.

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.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:

```
In [253]: 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:

```
In [254]: get_option("display.max_rows")
60

In [255]: set_option("display.max_rows", 999)

In [256]: get_option("display.max_rows")
999

In [257]: reset_option("display.max_rows")

In [258]: get_option("display.max_rows")
60
```

It's also possible to reset multiple options at once (using a regex):

```
In [259]: reset_option("^display")
```

8.14 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 [260]: set_eng_float_format(accuracy=3, use_eng_prefix=True)
```

```
In [261]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [262]: s/1.e3
```

```
a      1.116m
b    -829.274u
c     402.284u
d     159.191u
e    -447.033u
dtype: float64
```

```
In [263]: s/1.e6
```

```
a      1.116u
b    -829.274n
c     402.284n
d     159.191n
e    -447.033n
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 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

9.1 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 hierarchical indexes. As using integer slices with `.ix` have different behavior depending on whether the slice is interpreted as position based or label based, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at [Advanced Indexing](#), [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', :, :]`)

Object Type	Indexers
Series	<code>s.loc[indexer]</code>
DataFrame	<code>df.loc[row_indexer, column_indexer]</code>
Panel	<code>p.loc[item_indexer, major_indexer, minor_indexer]</code>

9.2 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.

9.3 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,

Object Type	Selection	Return Value Type
Series	<code>series[label]</code>	scalar value
DataFrame	<code>frame[colname]</code>	Series corresponding to colname
Panel	<code>panel[itemname]</code>	DataFrame corresponding to the itemname

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
```

```

           A          B          C          D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
[8 rows x 4 columns]
```

```
In [4]: panel = Panel({'one' : df, 'two' : df - df.mean()})
```

```
In [5]: panel
```

```

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D
```

Note: None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using []:

```
In [6]: s = df['A']
```

```
In [7]: s[dates[5]]
-0.67368970808837059
```

```
In [8]: panel['two']
```

```

           A          B          C          D
2000-01-01  0.409571  0.113086 -0.610826 -0.936507
2000-01-02  1.152571  0.222735  1.017442 -0.845111
2000-01-03 -0.921390 -1.708620  0.403304  1.270929
2000-01-04  0.662014 -0.310822 -0.141342  0.470985
2000-01-05 -0.484513  0.962970  1.174465 -0.888276
2000-01-06 -0.733231  0.509598 -0.580194  0.724113
2000-01-07  0.345164  0.972995 -0.816769 -0.840143
2000-01-08 -0.430188 -0.761943 -0.446079  1.044010
[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:

```
In [9]: df
```

```

           A          B          C          D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
```

```
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  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
```

```
          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.

9.4 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
2
```

```
In [15]: dfa.A
```

```
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
```

```
          A          B          C          D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
```

```
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
[8 rows x 4 columns]
```

Setting is allowed as well

```
In [17]: sa.a = 5
```

```
In [18]: sa
```

```
a      5
b      2
c      3
dtype: int64
```

```
In [19]: dfa.A = list(range(len(dfa.index)))
```

```
In [20]: dfa
```

```
          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 -1.478427  0.524988
2000-01-07  6  0.404705 -1.715002 -1.039268
2000-01-08  7 -0.370647 -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.

9.5 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]
```

```
2000-01-01    -0.282863
2000-01-02    -0.173215
2000-01-03    -2.104569
2000-01-04    -0.706771
2000-01-05     0.567020
Freq: D, Name: A, dtype: float64
```

```
In [22]: s[::2]
```

```
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]`

```
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`

```
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]`

```
          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]`

```
          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]
```

9.6 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
```

```
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:']
```

```
c    1.643563
d   -1.469388
e    0.357021
f   -0.674600
dtype: float64
```

```
In [32]: s1.loc['b']
-0.10904997528022223
```

Note that setting works as well:

```
In [33]: s1.loc['c:'] = 0
```

```
In [34]: s1
```

```
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
```

```
      A      B      C      D
a -1.776904 -0.968914 -1.294524  0.413738
b  0.276662 -0.472035 -0.013960 -0.362543
c -0.006154 -0.923061  0.895717  0.805244
d -1.206412  2.565646  1.431256  1.340309
e -1.170299 -0.226169  0.410835  0.813850
f  0.132003 -0.827317 -0.076467 -1.187678
[6 rows x 4 columns]
```

```
In [37]: df1.loc[['a','b','d'],:]
```

```
      A      B      C      D
a -1.776904 -0.968914 -1.294524  0.413738
b  0.276662 -0.472035 -0.013960 -0.362543
d -1.206412  2.565646  1.431256  1.340309
[3 rows x 4 columns]
```

Accessing via label slices

```
In [38]: df1.loc['d':,'A':'C']
```

```
      A      B      C
d -1.206412  2.565646  1.431256
e -1.170299 -0.226169  0.410835
f  0.132003 -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']
```

```
A    -1.776904
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
```

```
A    False
B    False
C    False
D     True
Name: a, dtype: bool
```

```
In [41]: df1.loc[:,df1.loc['a']>0]
```

```
      D
a  0.413738
b -0.362543
c  0.805244
d  1.340309
e  0.813850
f -1.187678
[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']
-1.7769037169718671
```

9.7 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
```

```
0    1.130127
2   -1.436737
4   -1.413681
6    1.607920
8    1.024180
dtype: float64
```

```
In [45]: s1.iloc[:3]
```

```
0    1.130127
2   -1.436737
4   -1.413681
dtype: float64
```

```
In [46]: s1.iloc[3]
1.6079204745847746
```

Note that setting works as well:

```
In [47]: s1.iloc[:3] = 0
```

```
In [48]: s1
```

```
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
```

```
      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]
```

```
      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]
```

```
      4         6
2 -0.078638  0.545952
4  0.769804 -1.281247
6 -0.097883  0.695775
8 -1.110336 -0.619976
[4 rows x 2 columns]
```

Select via integer list

```
In [53]: df1.iloc[[1,3,5],[1,3]]
```

```
      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,:]
```

```
      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]
```

```
      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]
-0.41000056806065832
```

For getting a cross section using an integer position (equiv to `df.xs(1)`)

```
In [57]: df1.iloc[1]
```

```
0    -2.006747
2    -0.410001
4    -0.078638
6     0.545952
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]
['e', 'f']
```

```
In [60]: x[8:10]
[]
```

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

```
>>> df.iloc[:,3:6]
IndexError: out-of-bounds on slice (end)
```

9.8 Setting With Enlargement

New in version 0.13. The `.loc/.ix/[]` operations can perform enlargement when setting a non-existent key for that axis.

In the `Series` case this is effectively an appending operation

```
In [61]: se = Series([1,2,3])
```

```
In [62]: se
```

```
0    1
1    2
2    3
dtype: int64
```

```
In [63]: se[5] = 5.
```

```
In [64]: se
```

```
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
```

```
   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
```

```
   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
```

```
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
[4 rows x 3 columns]
```

9.9 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.

Similarly to `loc`, `at` provides **label** based scalar lookups, while, `iat` provides **integer** based lookups analogously to `iloc`

```
In [71]: s.iat[5]
0.1136484096888855
```

```
In [72]: df.at[dates[5], 'A']
0.1136484096888855
```

```
In [73]: df.iat[3, 0]
-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
```

	A	B	C	D	E	0
2000-01-01	-0.282863	0.469112	-1.509059	-1.135632	NaN	NaN
2000-01-02	-0.173215	1.212112	0.119209	-1.044236	NaN	NaN
2000-01-03	-2.104569	-0.861849	-0.494929	1.071804	NaN	NaN
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
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]

9.10 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: `|` for `or`, `&` for `and`, and `~` for `not`. These **must** be grouped by using parentheses.

Using a boolean vector to index a `Series` works exactly as in a `numpy ndarray`:

```
In [78]: s[s > 0]
```

```
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)]
```

```
2000-01-01    -0.282863
2000-01-02    -0.173215
Freq: D, Name: A, dtype: float64
```

```
In [80]: s[(s < -1) | (s > 1)]
```

```
2000-01-03    -2.104569
2000-01-08    -1.157892
Name: A, dtype: float64
```

```
In [81]: s[~(s < 0)]
```

```
2000-01-05     0.567020
2000-01-06     0.113648
2000-01-07     0.577046
Freq: D, Name: A, dtype: float64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame's index (for example, something derived from one of the columns of the DataFrame):

```
In [82]: df[df['A'] > 0]
```

```
          A          B          C          D  E  0
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]
```

List comprehensions and map method of Series can also be used to produce more complex criteria:

```
In [83]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
.....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
.....:                  'c' : randn(7)})
.....:
```

```
# only want 'two' or 'three'
```

```
In [84]: criterion = df2['a'].map(lambda x: x.startswith('t'))
```

```
In [85]: df2[criterion]
```

```
   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']]]
```

```
   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')]
```

```

      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']
```

```

      b      c
3  x -2.179861
[1 rows x 2 columns]

```

9.10.1 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
```

```

4      0
3      1
2      2
1      3
0      4
dtype: int64

```

```
In [91]: s.isin([2, 4])
```

```

4      False
3      False
2       True
1      False
0       True
dtype: bool

```

```
In [92]: s[s.isin([2, 4])]
```

```

2      2
0      4
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)
```

```

      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)
```

```
      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]
```

```
      ids ids2  vals
0     a    a     1
[1 rows x 3 columns]
```

9.11 The `where()` Method and Masking

Selecting values from a `Series` with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows

```
In [101]: s[s > 0]
```

```
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)
```

```
4    NaN
3     1
2     2
1     3
```

```
0      4
dtype: float64
```

Selecting values from a DataFrame with a boolean criterion now also preserves input data shape. `where` is used under the hood as the implementation. Equivalent is `df[df < 0]`

```
In [103]: df[df < 0]
```

```

      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, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```
In [104]: df.where(df < 0, -df)
```

```

      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
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]
```

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
```

```

4      0
3      1
2      2
1      3
0      4
dtype: int64
```

```
In [108]: df2 = df.copy()
```

```
In [109]: df2[df2 < 0] = 0
```

```
In [110]: df2
```

```

      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    0.132885    0.000000
2000-01-07    2.410179    1.450520    0.206053    0.000000
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 [112]: df_orig
```

```
          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
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]
```

alignment

Furthermore, `where` aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analagous to partial setting via `.ix` (but on the contents rather than the axis labels)

```
In [113]: df2 = df.copy()

In [114]: df2[ df2[1:4] > 0 ] = 3

In [115]: df2
```

```
          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    3.000000
2000-01-04 -0.487602 -0.082240 -2.182937    3.000000
2000-01-05    0.084844    0.432390    1.519970 -0.493662
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]
```

New in version 0.13. `where` can also accept `axis` and `level` parameters to align the input when performing the `where`.

```
In [116]: df2 = df.copy()

In [117]: df2.where(df2>0,df2['A'],axis='index')
```

```
          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]

```

This is equivalent (but faster than) the following.

```
In [118]: df2 = df.copy()
```

```
In [119]: df.apply(lambda x, y: x.where(x>0,y), y=df['A'])
```

```

          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]

```

mask

mask is the inverse boolean operation of where.

```
In [120]: s.mask(s >= 0)
```

```

4    NaN
3    NaN
2    NaN
1    NaN
0    NaN
dtype: float64

```

```
In [121]: df.mask(df >= 0)
```

```

          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]

```

9.12 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 [122]: n = 10
```

```
In [123]: df = DataFrame(rand(n, 3), columns=list('abc'))
```

```
In [124]: df
```

```
      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
[10 rows x 3 columns]
```

```
# pure python
```

```
In [125]: df[(df.a < df.b) & (df.b < df.c)]
```

```
      a      b      c
2  0.276464  0.801872  0.958139
[1 rows x 3 columns]
```

```
# query
```

```
In [126]: df.query('(a < b) & (b < c)')
```

```
      a      b      c
2  0.276464  0.801872  0.958139
[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 [127]: df = DataFrame(randint(n / 2, size=(n, 2)), columns=list('bc'))
```

```
In [128]: df.index.name = 'a'
```

```
In [129]: df
```

```
      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 [130]: df.query('a < b and b < c')
```

```
      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 [131]: df = DataFrame(randint(n, size=(n, 2)), columns=list('bc'))
```

```
In [132]: df
```

```
   b  c
0  3  1
1  2  5
2  2  5
3  6  7
4  4  3
5  5  6
6  4  6
7  2  4
8  2  7
9  9  7
[10 rows x 2 columns]
```

```
In [133]: df.query('index < b < c')
```

```
   b  c
1  2  5
3  6  7
[2 rows x 2 columns]
```

9.12.1 MultiIndex query() Syntax

You can also use the levels of a DataFrame with a MultiIndex as if they were columns in the frame:

```
In [134]: import pandas.util.testing as tm
```

```
In [135]: n = 10
```

```
In [136]: colors = tm.choice(['red', 'green'], size=n)
```

```
In [137]: foods = tm.choice(['eggs', 'ham'], size=n)
```

```
In [138]: colors
```

```
array(['green', 'red', 'green', 'red', 'green', 'red', 'green', 'red',
       'green', 'red'],
      dtype='<S5')
```

```
In [139]: foods
```

```
array(['ham', 'eggs', 'ham', 'eggs', 'ham', 'eggs', 'ham', 'ham', 'ham',
       'ham'],
      dtype='<S4')
```

```
In [140]: index = MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
```

```
In [141]: df = DataFrame(randn(n, 2), index=index)
```

```
In [142]: df
```

```
           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
red  ham    0.704228  0.523508
green ham  -0.926254  2.007843
red  ham    0.226963 -1.152659
[10 rows x 2 columns]
```

```
In [143]: df.query('color == "red"')
```

```
           0          1
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 [144]: df.index.names = [None, None]
```

```
In [145]: df
```

```
           0          1
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 [146]: df.query('ilevel_0 == "red"')
```

```
           0          1
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.

9.12.2 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 [147]: df = DataFrame(rand(n, 3), columns=list('abc'))
```

```
In [148]: df
```

```
      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 [149]: df2 = DataFrame(rand(n + 2, 3), columns=df.columns)
```

```
In [150]: df2
```

```
      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 [151]: expr = '0.0 <= a <= c <= 0.5'
```

```
In [152]: map(lambda frame: frame.query(expr), [df, df2])
```

```
[      a      b      c
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]]
```

9.12.3 query() Python versus pandas Syntax Comparison

Full numpy-like syntax

```
In [153]: df = DataFrame(randint(n, size=(n, 3)), columns=list('abc'))
```

```
In [154]: df
```

```
   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 [155]: df.query('(a < b) & (b < c)')
```

```
   a  b  c
5  1  3  6
9  2  3  4
[2 rows x 3 columns]
```

```
In [156]: df[(df.a < df.b) & (df.b < df.c)]
```

```
   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 [157]: df.query('a < b & b < c')
```

```
   a  b  c
5  1  3  6
9  2  3  4
[2 rows x 3 columns]
```

Use English instead of symbols

```
In [158]: df.query('a < b and b < c')
```

```
   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 [159]: df.query('a < b < c')
```

```
   a  b  c
5  1  3  6
9  2  3  4
[2 rows x 3 columns]
```

9.12.4 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 [160]: df = DataFrame({'a': list('aabbccddee'), 'b': list('aaaabbbbcccc'),
.....:                  'c': randint(5, size=12), 'd': randint(9, size=12)})
.....:
```

```
In [161]: df
```

```
   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 [162]: df.query('a in b')
```

```
   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 [163]: df[df.a.isin(df.b)]
```

```
   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 [164]: df.query('a not in b')
```

```
   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]

# pure Python
In [165]: df[~df.a.isin(df.b)]

   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 [166]: df.query('a in b and c < d')

   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 [167]: df[df.b.isin(df.a) & (df.c < df.d)]

   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 operations that can be evaluated using `numexpr` will be.

9.12.5 Special use of the `==` operator with `list` objects

Comparing a `list` of values to a column using `==/!=` works similarly to `in/not in`

```
In [168]: df.query('b == ["a", "b", "c"]')

   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]

```

pure Python

```
In [169]: df[df.b.isin(["a", "b", "c"])]
```

```

      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 [170]: df.query('c == [1, 2]')
```

```

      a   b   c   d
0   a   a   2   2
2   b   a   1   8
3   b   a   1   8
7   d   b   2   0
8   e   c   1   0
10  f   c   2   6
[6 rows x 4 columns]

```

```
In [171]: df.query('c != [1, 2]')
```

```

      a   b   c   d
1   a   a   3   5
4   c   b   4   7
5   c   b   0   5
6   d   b   0   7
9   e   c   4   6
11  f   c   3   1
[6 rows x 4 columns]

```

using in/not in

```
In [172]: df.query('[1, 2] in c')
```

```

      a   b   c   d

```

```
0    a    a    2    2
2    b    a    1    8
3    b    a    1    8
7    d    b    2    0
8    e    c    1    0
10   f    c    2    6
[6 rows x 4 columns]
```

```
In [173]: df.query('[1, 2] not in c')
```

```
      a    b    c    d
1    a    a    3    5
4    c    b    4    7
5    c    b    0    5
6    d    b    0    7
9    e    c    4    6
11   f    c    3    1
[6 rows x 4 columns]
```

```
# pure Python
```

```
In [174]: df[df.c.isin([1, 2])]
```

```
      a    b    c    d
0    a    a    2    2
2    b    a    1    8
3    b    a    1    8
7    d    b    2    0
8    e    c    1    0
10   f    c    2    6
[6 rows x 4 columns]
```

9.12.6 Boolean Operators

You can negate boolean expressions with the word `not` or the `~` operator.

```
In [175]: df = DataFrame(rand(n, 3), columns=list('abc'))
```

```
In [176]: df['bools'] = rand(len(df)) > 0.5
```

```
In [177]: df.query('~bools')
```

```
      a      b      c  bools
0  0.035334  0.943947  0.405569  False
1  0.447902  0.782636  0.574193  False
4  0.791346  0.491674  0.395827  False
5  0.035597  0.171689  0.189045  False
7  0.898831  0.435002  0.078368  False
8  0.224708  0.697626  0.499990  False
9  0.504279  0.746247  0.877177  False
[7 rows x 4 columns]
```

```
In [178]: df.query('not bools')
```

```
      a      b      c  bools
0  0.035334  0.943947  0.405569  False
1  0.447902  0.782636  0.574193  False
4  0.791346  0.491674  0.395827  False
```

```

5  0.035597  0.171689  0.189045  False
7  0.898831  0.435002  0.078368  False
8  0.224708  0.697626  0.499990  False
9  0.504279  0.746247  0.877177  False
[7 rows x 4 columns]

```

```
In [179]: df.query('not bools') == df[~df.bools]
```

```

      a      b      c bools
0  True  True  True   True
1  True  True  True   True
4  True  True  True   True
5  True  True  True   True
7  True  True  True   True
8  True  True  True   True
9  True  True  True   True
[7 rows x 4 columns]

```

Of course, expressions can be arbitrarily complex too

```
# short query syntax
```

```
In [180]: shorter = df.query('a < b < c and (not bools) or bools > 2')
```

```
# equivalent in pure Python
```

```
In [181]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]
```

```
In [182]: shorter
```

```

      a      b      c bools
5  0.035597  0.171689  0.189045  False
9  0.504279  0.746247  0.877177  False
[2 rows x 4 columns]

```

```
In [183]: longer
```

```

      a      b      c bools
5  0.035597  0.171689  0.189045  False
9  0.504279  0.746247  0.877177  False
[2 rows x 4 columns]

```

```
In [184]: shorter == longer
```

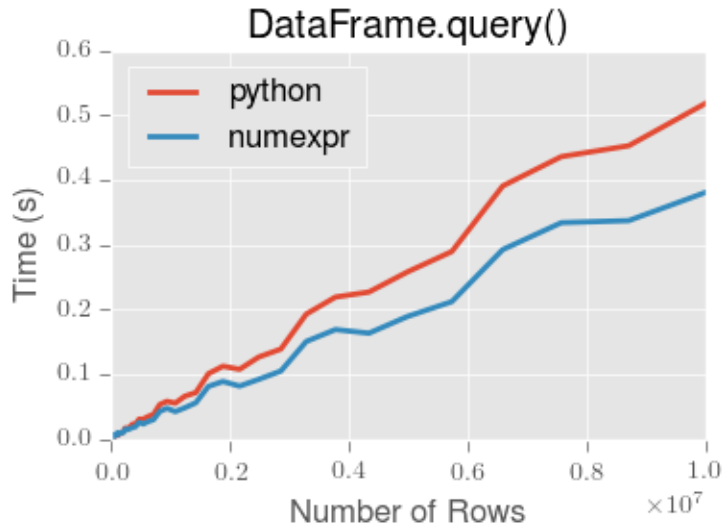
```

      a      b      c bools
5  True  True  True   True
9  True  True  True   True
[2 rows x 4 columns]

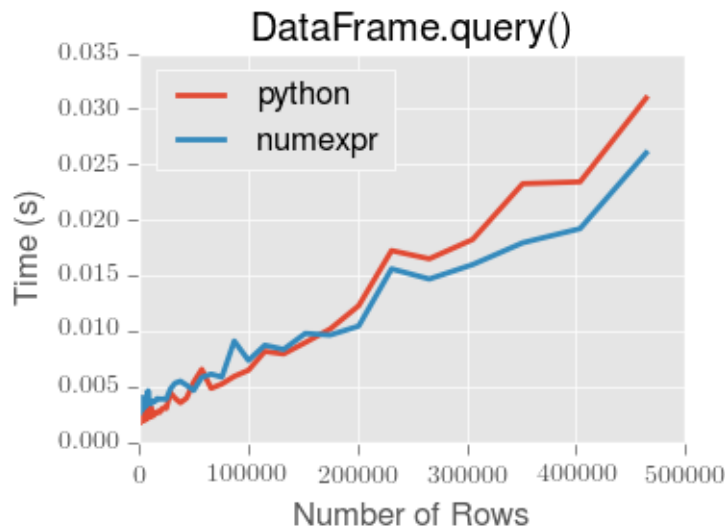
```

9.12.7 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()`.

9.13 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 [185]: index = Index(randint(0, 1000, 10))
```

```
In [186]: index
Int64Index([399, 134, 575, 635, 358, 102, 468, 657, 848, 343], dtype='int64')
```

```
In [187]: positions = [0, 9, 3]
```

```
In [188]: index[positions]
Int64Index([399, 343, 635], dtype='int64')
```

```
In [189]: index.take(positions)
Int64Index([399, 343, 635], dtype='int64')
```

```
In [190]: ser = Series(randn(10))
```

```
In [191]: ser.ix[positions]
```

```
0    -0.921988
9     0.391944
3    -0.220720
dtype: float64
```

```
In [192]: ser.take(positions)
```

```
0    -0.921988
9     0.391944
3    -0.220720
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [193]: frm = DataFrame(randn(5, 3))
```

```
In [194]: frm.take([1, 4, 3])
```

```
      0      1      2
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]
```

```
In [195]: frm.take([0, 2], axis=1)
```

```
      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.

```
In [196]: arr = randn(10)
```

```
In [197]: arr.take([False, False, True, True])
array([ 0.191 ,  0.191 ,  0.2296,  0.2296])
```

```
In [198]: arr[[0, 1]]
array([ 0.191 ,  0.2296])
```

```
In [199]: ser = Series(randn(10))

In [200]: ser.take([False, False, True, True])

0    1.557902
0    1.557902
1    1.089202
1    1.089202
dtype: float64

In [201]: ser.ix[[0, 1]]

0    1.557902
1    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.

9.14 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 [202]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
.....:                   'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
.....:                   'c' : np.random.randn(7)})
.....:

In [203]: df2.duplicated(['a', 'b'])

0    False
1    False
2    False
3    False
4     True
5     True
6    False
dtype: bool

In [204]: df2.drop_duplicates(['a', 'b'])

   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 [205]: df2.drop_duplicates(['a','b'], take_last=True)
```

```

      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]
```

9.15 Dictionary-like get () method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.

```
In [206]: s = Series([1,2,3], index=['a','b','c'])
```

```
In [207]: s.get('a')                                # equivalent to s['a']
1
```

```
In [208]: s.get('x', default=-1)
-1
```

9.16 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 [209]: subindex = dates[[3,4,5]]
```

```
In [210]: df.reindex(index=subindex, columns=['C', 'B'])
```

```

              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 [211]: df.ix[subindex, ['C', 'B']]
```

```
           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 [212]: df2 = df.copy()
```

```
In [213]: df2.ix[subindex, ['C', 'B']] = 0
```

```
In [214]: df2
```

```
           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  0.245116  0.000000  0.000000 -0.546831
2000-01-05  1.459886  0.000000  0.000000 -0.038520
2000-01-06  1.926220  0.000000  0.000000  0.710816
2000-01-07 -0.042475 -1.265025 -0.677097  1.701349
2000-01-08  0.518029 -0.592656  1.482845  0.227322
[8 rows x 4 columns]
```

Indexing with an array of integers can also be done:

```
In [215]: df.ix[[4,3,1]]
```

```
           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]
```

```
In [216]: df.ix[dates[[4,3,1]]]
```

```
           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 [217]: df.ix[1:7, :2]
```

```
           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 [218]: date1, date2 = dates[[2, 4]]
```

```
In [219]: print(date1, date2)
(timestamp('2000-01-03 00:00:00', tz=None), timestamp('2000-01-05 00:00:00', tz=None))
```

```
In [220]: df.ix[date1:date2]
```

```

           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
2000-01-05  1.459886 -1.180301  0.378125 -0.038520
[3 rows x 4 columns]
```

```
In [221]: df['A'].ix[date1:date2]
```

```

2000-01-03    1.315562
2000-01-04    0.245116
2000-01-05    1.459886
Freq: D, Name: A, dtype: float64
```

Getting and setting rows in a DataFrame, especially by their location, is much easier:

```
In [222]: df2 = df[:5].copy()
```

```
In [223]: df2.ix[3]
```

```

A    0.245116
B    0.484166
C    0.036249
D   -0.546831
Name: 2000-01-04 00:00:00, dtype: float64
```

```
In [224]: df2.ix[3] = np.arange(len(df2.columns))
```

```
In [225]: df2
```

```

           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
[5 rows x 4 columns]
```

Column or row selection can be combined as you would expect with arrays of labels or even boolean vectors:

```
In [226]: df.ix[df['A'] > 0, 'B']
```

```

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 [227]: df.ix[date1:date2, 'B']
```

```
2000-01-03    0.089291
2000-01-04    0.484166
2000-01-05   -1.180301
Freq: D, Name: B, dtype: float64
```

```
In [228]: df.ix[date1, 'B']
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).

9.17 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 [229]: df.select(lambda x: x == 'A', axis=1)
```

```
      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]
```

9.18 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 [230]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])
```

```
In [231]: dflookup.lookup(list(range(0,10,2)), ['B','C','A','B','D'])
array([ 0.4254,  0.4579,  0.0333,  0.3469,  0.3085])
```

9.19 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 [232]: indexf = Index([1.5, 2, 3, 4.5, 5])
```

```
In [233]: indexf
Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='object')
```

```
In [234]: sf = Series(range(5), index=indexf)
```

```
In [235]: sf
```

```
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 [236]: sf[3]
2
```

```
In [237]: sf[3.0]
2
```

```
In [238]: sf.ix[3]
2
```

```
In [239]: sf.ix[3.0]
2
```

```
In [240]: sf.loc[3]
2
```

```
In [241]: sf.loc[3.0]
2
```

The only positional indexing is via `iloc`

```
In [242]: sf.iloc[3]
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 [243]: sf[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [244]: sf.ix[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [245]: sf.loc[2:4]
```

```
2    1
3    2
dtype: int64
```

```
In [246]: sf.iloc[2:4]
```

```
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed

```
In [247]: sf[2.1:4.6]
```

```
3.0    2
4.5    3
dtype: int64
```

```
In [248]: sf.loc[2.1:4.6]
```

```
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 [249]: 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 [250]: dfir
```

```
      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
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 [251]: dfir[0:1000.4]
```

```

      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 [252]: dfir.loc[0:1001, 'A']
```

```

0.0      -0.384281
250.0    -1.804211
500.0      0.640348
750.0    -0.582619
1000.0     0.039911
1000.4     1.115044
Name: A, dtype: float64
```

```
In [253]: dfir.loc[1000.4]
```

```

A      1.115044
B      0.969404
Name: 1000.4, dtype: float64
```

You could then easily pick out the first 1 second (1000 ms) of data then.

```
In [254]: dfir[0:1000]
```

```

      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 [255]: dfir.iloc[0:5]
```

```

      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]
```

9.20 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 [256]: dfb = DataFrame({'a' : ['one', 'one', 'two',
.....:                        'three', 'two', 'one', 'six'],
.....:                  'c' : np.arange(7)})
.....:
```

```
# passed via reference (will stay)
```

```
In [257]: 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
Traceback (most recent call last):
...
SettingWithCopyWarning:
  A value is trying to be set on a copy of a slice from a DataFrame.
  Try using .loc[row_index,col_indexer] = value instead
```

A chained assignment can also crop up in setting in a mixed dtype frame.

Note: These setting rules apply to all of `.loc/.iloc/.ix`

This is the correct access method

```
In [258]: dfc = DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})
```

```
In [259]: dfc.loc[0, 'A'] = 11
```

```
In [260]: dfc
```

```
   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 [261]: dfc = dfc.copy()
```

```
In [262]: dfc['A'][0] = 111
```

```
In [263]: dfc
```

```
   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

```
>>> 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.

9.21 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 positional indexing.

```
In [264]: df = pd.DataFrame(np.random.randn(4,4),
.....:                      columns=list('ABCD'), index=[1.0, 2.0, 3.0, 4.0])
.....:
```

```
In [265]: df
```

```
      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 [266]: df.ix[1]
```

```
A      0.379122
B     -1.909492
C     -1.431211
D      1.329653
Name: 1.0, dtype: float64
```

To select the row you do expect, instead use a float label or use `iloc`.

```
In [267]: df.ix[1.0]
```

```
A      0.379122
B     -1.909492
C     -1.431211
D      1.329653
Name: 1.0, dtype: float64
```

```
In [268]: df.iloc[0]
```

```
A      0.379122
B     -1.909492
C     -1.431211
```

```
D      1.329653
Name: 1.0, dtype: float64
```

Instead of using a float index, it is often better to convert to an integer index:

```
In [269]: df_new = df.reset_index()
```

```
In [270]: df_new[df_new['index'] == 1.0]
```

```
   index      A      B      C      D
0      1  0.379122 -1.909492 -1.431211  1.329653
[1 rows x 5 columns]
```

```
# now you can also do "float selection"
```

```
In [271]: df_new[(df_new['index'] >= 1.0) & (df_new['index'] < 2)]
```

```
   index      A      B      C      D
0      1  0.379122 -1.909492 -1.431211  1.329653
[1 rows x 5 columns]
```

9.22 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`:

```
In [272]: index = Index(['e', 'd', 'a', 'b'])
```

```
In [273]: index
Index([u'e', u'd', u'a', u'b'], dtype='object')
```

```
In [274]: 'd' in index
True
```

You can also pass a name to be stored in the index:

```
In [275]: index = Index(['e', 'd', 'a', 'b'], name='something')
```

```
In [276]: index.name
'something'
```

Starting with pandas 0.5, the name, if set, will be shown in the console display:

```
In [277]: index = Index(list(range(5)), name='rows')
```

```
In [278]: columns = Index(['A', 'B', 'C'], name='cols')
```

```
In [279]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)
```

```
In [280]: df
```

```
cols      A      B      C
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 [281]: df['A']
```

```

rows
0    0.242701
1   -1.524904
2    1.059562
3    0.150077
4   -1.972104
Name: A, dtype: float64

```

9.22.1 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 [282]: a = Index(['c', 'b', 'a'])
```

```
In [283]: b = Index(['c', 'e', 'd'])
```

```
In [284]: a.union(b)
Index([u'a', u'b', u'c', u'd', u'e'], dtype='object')
```

```
In [285]: a | b
Index([u'a', u'b', u'c', u'd', u'e'], dtype='object')
```

```
In [286]: a & b
Index([u'c'], dtype='object')
```

```
In [287]: a - b
Index([u'a', u'b'], dtype='object')
```

9.22.2 The `isin` method of Index objects

One additional operation is the `isin` method that works analogously to the `Series.isin` method found [here](#).

9.23 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.

9.23.1 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) or an array of tuples (using MultiIndex.from_tuples).

```
In [288]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
.....:             ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
.....:
```

```
In [289]: tuples = list(zip(*arrays))
```

```
In [290]: tuples
```

```
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]
```

```
In [291]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

```
In [292]: s = Series(randn(8), index=index)
```

```
In [293]: s
```

```
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
```

As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

```
In [294]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'])
.....: ,
.....:               np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])
.....: ]
.....:
```

```
In [295]: s = Series(randn(8), index=arrays)
```

```
In [296]: 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 [297]: df = DataFrame(randn(8, 4), index=arrays)
```

```
In [298]: 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
[8 rows x 4 columns]
```

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 [299]: df.index.names
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 [300]: df = DataFrame(randn(3, 8), index=['A', 'B', 'C'], columns=index)
```

```
In [301]: df
```

```
first      bar      baz      foo      qux \
second    one    two    one    two    one    two    one
A      -0.007381 -1.219794  0.145578 -0.249321 -1.046479  1.314373  0.716789
B      -0.365315  0.370955  1.428502 -0.292967 -1.250595  0.333150  0.616471
C      -0.024817 -0.795125 -0.408384 -1.849202  0.781722  0.133331 -0.298493
first
second      two
A      0.385795
B     -0.915417
C     -1.367644
[3 rows x 8 columns]
```

```
In [302]: DataFrame(randn(6, 6), index=index[:6], columns=index[:6])
```

```
first      bar      baz      foo
second    one    two    one    two    one    two
first second
bar one    0.392245 -0.738972  0.357817  1.291147 -0.787450  1.023850
```

```
      two      0.475844  0.159213  1.002647  0.137063  0.287958 -0.651968
baz    one    -0.422738 -0.304204  1.234844  0.692625 -2.093541  0.688230
      two      1.060943  1.152768  1.264767  0.140697  0.057916  0.405542
foo    one      0.084720  1.833111  2.103399  0.073064 -0.687485 -0.015795
      two     -0.242492  0.697262  1.151237  0.627468  0.397786 -0.811265
[6 rows x 6 columns]
```

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:

```
In [303]: Series(randn(8), index=tuples)
```

```
(bar, one)    -0.198387
(bar, two)     1.403283
(baz, one)     0.024097
(baz, two)    -0.773295
(foo, one)     0.463600
(foo, two)     1.969721
(qux, one)     0.948590
(qux, two)    -0.490665
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

Note that how the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_printoptions`:

```
In [304]: pd.set_option('display.multi_sparse', False)
```

```
In [305]: df
```

```
first      bar      bar      baz      baz      foo      foo      qux  \
second     one      two      one      two      one      two      one
A    -0.007381 -1.219794  0.145578 -0.249321 -1.046479  1.314373  0.716789
B    -0.365315  0.370955  1.428502 -0.292967 -1.250595  0.333150  0.616471
C    -0.024817 -0.795125 -0.408384 -1.849202  0.781722  0.133331 -0.298493
first      qux
second     two
A          0.385795
B         -0.915417
C         -1.367644
[3 rows x 8 columns]
```

```
In [306]: pd.set_option('display.multi_sparse', True)
```

9.23.2 Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

```
In [307]: index.get_level_values(0)
Index([u'bar', u'bar', u'baz', u'baz', u'foo', u'foo', u'qux', u'qux'], dtype='object')
```

```
In [308]: index.get_level_values('second')
Index([u'one', u'two', u'one', u'two', u'one', u'two', u'one', u'two'], dtype='object')
```

9.23.3 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 [309]: df['bar']
```

```
second      one      two
A      -0.007381 -1.219794
B      -0.365315  0.370955
C      -0.024817 -0.795125
[3 rows x 2 columns]
```

```
In [310]: df['bar', 'one']
```

```
A      -0.007381
B      -0.365315
C      -0.024817
Name: (bar, one), dtype: float64
```

```
In [311]: df['bar']['one']
```

```
A      -0.007381
B      -0.365315
C      -0.024817
Name: one, dtype: float64
```

```
In [312]: s['qux']
```

```
one      0.051545
two      1.452842
dtype: float64
```

9.23.4 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 [313]: s + s[:-2]
```

```
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 [314]: s + s[:,2]
```

```
bar  one      2.078365
      two             NaN
baz  one      2.897837
      two             NaN
```

```
foo one    0.348063
   two      NaN
qux one    0.103090
   two      NaN
dtype: float64
```

`reindex` can be called with another `MultiIndex` or even a list or array of tuples:

```
In [315]: s.reindex(index[:3])
```

```
first second
bar one    1.039182
   two    0.956255
baz one    1.448918
dtype: float64
```

```
In [316]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
```

```
foo two   -0.793292
bar one    1.039182
qux one    0.051545
baz one    1.448918
dtype: float64
```

9.23.5 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 [317]: df = df.T
```

```
In [318]: df
```

```
          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 [319]: df.ix['bar']
```

```
          A          B          C
second
one   -0.007381 -0.365315 -0.024817
two   -1.219794  0.370955 -0.795125
[2 rows x 3 columns]
```

```
In [320]: df.ix['bar', 'two']
```

```
A   -1.219794
B    0.370955
```

```
C    -0.795125
Name: (bar, two), dtype: float64
```

“Partial” slicing also works quite nicely:

```
In [321]: df.ix['baz':'foo']
```

```
          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]
```

```
In [322]: df.ix[('baz', 'two'):( 'qux', 'one')]
```

```
          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 [323]: df.ix[('baz', 'two'):'foo']
```

```
          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
[3 rows x 3 columns]
```

Passing a list of labels or tuples works similar to reindexing:

```
In [324]: df.ix[[('bar', 'two'), ('qux', 'one')]]
```

```
          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.

9.23.6 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 [325]: df.xs('one', level='second')
```

```
      A      B      C
first
bar -0.007381 -0.365315 -0.024817
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 [326]: df = df.T
```

```
In [327]: df.xs('one', level='second', axis=1)
```

```
first      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 [328]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
```

```
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)

9.23.7 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 [329]: midx = MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
.....:                      labels=[[1,1,0,0],[1,0,1,0]])
.....:
```

```
In [330]: df = DataFrame(randn(4,2), index=midx)
```

```
In [331]: print(df)
      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 [332]: df2 = df.mean(level=0)
```



```

In [333]: print(df2)
           0      1
zero  0.543033  0.029244
one   0.258129  0.522253
[2 rows x 2 columns]

In [334]: 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 [335]: df_aligned, df2_aligned = df.align(df2, level=0)

In [336]: print(df_aligned)
           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 [337]: 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
[4 rows x 2 columns]

```

9.23.8 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 [338]: import random; random.shuffle(tuples)

In [339]: s = Series(randn(8), index=MultiIndex.from_tuples(tuples))

In [340]: s
foo two    0.127064
qux two    0.396144
     one    1.043289
foo one   -0.229627
bar two    0.158186
     one   -0.281965
baz one    1.255148
     two    3.063464

```

```
dtype: float64
```

```
In [341]: s.sortlevel(0)
```

```
bar one -0.281965
      two  0.158186
baz one  1.255148
      two  3.063464
foo one -0.229627
      two  0.127064
qux one  1.043289
      two  0.396144
dtype: float64
```

```
In [342]: s.sortlevel(1)
```

```
bar one -0.281965
baz one  1.255148
foo one -0.229627
qux one  1.043289
bar two  0.158186
baz two  3.063464
foo two  0.127064
qux two  0.396144
dtype: float64
```

Note, you may also pass a level name to `sortlevel` if the `MultiIndex` levels are named.

```
In [343]: s.index.set_names(['L1', 'L2'], inplace=True)
```

```
In [344]: s.sortlevel(level='L1')
```

```
L1  L2
bar one -0.281965
      two  0.158186
baz one  1.255148
      two  3.063464
foo one -0.229627
      two  0.127064
qux one  1.043289
      two  0.396144
dtype: float64
```

```
In [345]: s.sortlevel(level='L2')
```

```
L1  L2
bar one -0.281965
baz one  1.255148
foo one -0.229627
qux one  1.043289
bar two  0.158186
baz two  3.063464
foo two  0.127064
qux two  0.396144
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 [346]: s['qux']
```

```
L2
two    0.396144
one    1.043289
dtype: float64
```

```
In [347]: s.sortlevel(1)['qux']
```

```
L2
one    1.043289
two    0.396144
dtype: float64
```

On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

```
In [348]: df.T.sortlevel(1, axis=1)
```

```
      zero      one      zero      one
      x      x      y      y
0  1.643615  0.203166 -0.557549  0.313092
1 -0.067716  1.632996  0.126204 -0.588491
[2 rows x 4 columns]
```

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 [349]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]
```

```
In [350]: idx = MultiIndex.from_tuples(tuples)
```

```
In [351]: idx.lexsort_depth
2
```

```
In [352]: reordered = idx[[1, 0, 3, 2]]
```

```
In [353]: reordered.lexsort_depth
1
```

```
In [354]: s = Series(randn(4), index=reordered)
```

```
In [355]: s.ix['a':'a']
```

```
a b    0.304771
a   -0.766820
dtype: float64
```

However:

```
>>> s.ix[('a', 'b'):(('b', 'a'))]
Traceback (most recent call last)
...
KeyError: Key length (3) was greater than MultiIndex lexsort depth (2)
```

9.23.9 Swapping levels with `swaplevel()`

The `swaplevel` function can switch the order of two levels:

```
In [356]: df[:5]
```

```
      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 [357]: df[:5].swaplevel(0, 1, axis=0)
```

```
      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]
```

9.23.10 Reordering levels with `reorder_levels()`

The `reorder_levels` function generalizes the `swaplevel` function, allowing you to permute the hierarchical index levels in one step:

```
In [358]: df[:5].reorder_levels([1,0], axis=0)
```

```
      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]
```

9.23.11 Some gory internal details

Internally, the `MultiIndex` consists of a few things: the **levels**, the integer **labels**, and the level **names**:

```
In [359]: index
```

```
MultiIndex(levels=[[u'bar', u'baz', u'foo', u'qux'], [u'one', u'two']],
            labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
            names=[u'first', u'second'])
```

```
In [360]: index.levels
```

```
FrozenList([[u'bar', u'baz', u'foo', u'qux'], [u'one', u'two']])
```

```
In [361]: index.labels
```

```
FrozenList([[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]])
```

```
In [362]: index.names
```

```
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.

9.24 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.

```
In [363]: ind = Index([1, 2, 3])

In [364]: ind.rename("apple")
Int64Index([1, 2, 3], dtype='int64')

In [365]: ind
Int64Index([1, 2, 3], dtype='int64')

In [366]: ind.set_names(["apple"], inplace=True)

In [367]: ind.name = "bob"

In [368]: ind
Int64Index([1, 2, 3], dtype='int64')
```

9.25 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.

9.26 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 [369]: data

   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 [370]: indexed1 = data.set_index('c')

In [371]: indexed1

   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 [372]: indexed2 = data.set_index(['a', 'b'])
```

```
In [373]: indexed2
```

```
      c  d
a  b
bar one z  1
      two y  2
foo one x  3
      two w  4
[4 rows x 2 columns]
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [374]: frame = data.set_index('c', drop=False)
```

```
In [375]: frame = frame.set_index(['a', 'b'], append=True)
```

```
In [376]: frame
```

```
      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]
```

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 [377]: data.set_index('c', drop=False)
```

```
      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 [378]: data.set_index(['a', 'b'], inplace=True)
```

```
In [379]: data
```

```
      c  d
a  b
bar one z  1
      two y  2
foo one x  3
      two w  4
[4 rows x 2 columns]
```

9.27 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 [380]: data
```

```
      c  d
a  b
bar one z  1
      two y  2
foo one x  3
      two w  4
[4 rows x 2 columns]
```

```
In [381]: data.reset_index()
```

```
      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 [382]: frame
```

```
      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 [383]: frame.reset_index(level=1)
```

```
      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.

9.28 Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
data.index = index
```

9.29 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`

COMPUTATIONAL TOOLS

10.1 Statistical functions

10.1.1 Percent Change

Both `Series` and `DataFrame` has 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()
```

```
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)
```

```
      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]
```

10.1.2 Covariance

The `Series` object has a method `cov` to compute covariance between series (excluding NA/null values).

```
In [5]: s1 = Series(randn(1000))
```

```
In [6]: s2 = Series(randn(1000))
```

```
In [7]: s1.cov(s2)
0.0006801088174310957
```

Analogously, `DataFrame` has a method `cov` to compute pairwise covariances among the series in the `DataFrame`, also excluding NA/null values.

```
In [8]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
```

```
In [9]: frame.cov()

           a          b          c          d          e
a  1.000882 -0.003177 -0.002698 -0.006889  0.031912
b -0.003177  1.024721  0.000191  0.009212  0.000857
c -0.002698  0.000191  0.950735 -0.031743 -0.005087
d -0.006889  0.009212 -0.031743  1.002983 -0.047952
e  0.031912  0.000857 -0.005087 -0.047952  1.042487
[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.

```
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()

           a          b          c
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)
```

```
           a          b          c
a  1.210090      NaN  0.018002
b      NaN  1.240960  0.347188
c  0.018002  0.347188  1.301149
[3 rows x 3 columns]
```

10.1.3 Correlation

Several methods for computing correlations are provided. Several kinds of correlation methods are provided:

Method name	Description
pearson (default)	Standard correlation coefficient
kendall	Kendall Tau correlation coefficient
spearman	Spearman rank correlation coefficient

All of these are currently computed using pairwise complete observations.

```
In [15]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
```

```
In [16]: frame.ix[:,2] = np.nan
```

```
# Series with Series
```

```
In [17]: frame['a'].corr(frame['b'])
0.013479040400098763
```

```
In [18]: frame['a'].corr(frame['b'], method='spearman')
-0.0072898851595406388
```

```
# Pairwise correlation of DataFrame columns
```

```
In [19]: frame.corr()

          a          b          c          d          e
a  1.000000  0.013479 -0.049269 -0.042239 -0.028525
b  0.013479  1.000000 -0.020433 -0.011139  0.005654
c -0.049269 -0.020433  1.000000  0.018587 -0.054269
d -0.042239 -0.011139  0.018587  1.000000 -0.017060
e -0.028525  0.005654 -0.054269 -0.017060  1.000000
[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:

```
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()
```

```
          a          b          c
a  1.000000 -0.076520  0.160092
b -0.076520  1.000000  0.135967
c  0.160092  0.135967  1.000000
[3 rows x 3 columns]
```

```
In [24]: frame.corr(min_periods=12)
```

```
          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.

```
In [25]: index = ['a', 'b', 'c', 'd', 'e']
```

```
In [26]: columns = ['one', 'two', 'three', 'four']
```

```
In [27]: df1 = DataFrame(randn(5, 4), index=index, columns=columns)
```

```
In [28]: df2 = DataFrame(randn(4, 4), index=index[:4], columns=columns)
```

```
In [29]: df1.corrwith(df2)
```

```
one      -0.125501
two      -0.493244
three     0.344056
four      0.004183
dtype: float64
```

```
In [30]: df2.corrwith(df1, axis=1)
```

```
a      -0.675817
b       0.458296
c       0.190809
d      -0.186275
e           NaN
dtype: float64
```

10.1.4 Data ranking

The rank method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```
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()
```

```
a      5.0
b      2.5
c      1.0
d      2.5
e      4.0
dtype: float64
```

rank is also a DataFrame method and can rank either the rows (axis=0) or the columns (axis=1). NaN values are excluded from the ranking.

```
In [34]: df = DataFrame(np.random.randn(10, 6))
```

```
In [35]: df[4] = df[2][:5] # some ties
```

```
In [36]: df
```

```
      0         1         2         3         4         5
0 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.204848         NaN  0.463347
6  0.376892  0.959292  0.095572 -0.593740         NaN -0.069180
7 -1.002601  1.957794 -0.120708  0.094214         NaN -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254         NaN -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477         NaN  1.375064
[10 rows x 6 columns]
```

```
In [37]: df.rank(1)
```

```

      0  1    2  3    4  5
0  4  3  1.5  5  1.5  6
1  2  6  4.5  1  4.5  3
2  1  6  3.5  5  3.5  2
3  4  5  1.5  3  1.5  6
4  5  3  1.5  4  1.5  6
5  1  2  5.0  3  NaN  4
6  4  5  3.0  1  NaN  2
7  2  5  3.0  4  NaN  1
8  2  5  3.0  4  NaN  1
9  2  3  1.0  4  NaN  5
[10 rows x 6 columns]
```

`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

10.2 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`.

Function	Description
<code>rolling_count</code>	Number of non-null observations
<code>rolling_sum</code>	Sum of values
<code>rolling_mean</code>	Mean of values
<code>rolling_median</code>	Arithmetic median of values
<code>rolling_min</code>	Minimum
<code>rolling_max</code>	Maximum
<code>rolling_std</code>	Unbiased standard deviation
<code>rolling_var</code>	Unbiased variance
<code>rolling_skew</code>	Unbiased skewness (3rd moment)
<code>rolling_kurt</code>	Unbiased kurtosis (4th moment)
<code>rolling_quantile</code>	Sample quantile (value at %)
<code>rolling_apply</code>	Generic apply
<code>rolling_cov</code>	Unbiased covariance (binary)
<code>rolling_corr</code>	Correlation (binary)
<code>rolling_corr_pairwise</code>	Pairwise correlation of DataFrame columns
<code>rolling_window</code>	Moving window function

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

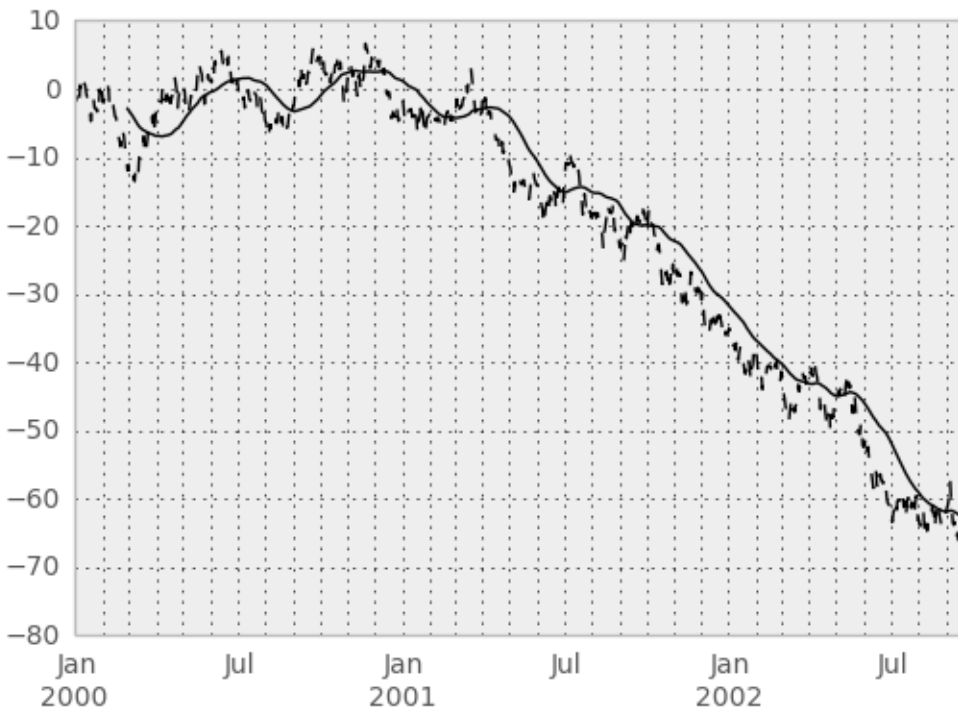
These functions can be applied to `ndarrays` or `Series` objects:

```
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--')
<matplotlib.axes.AxesSubplot at 0x85db190>
```

```
In [41]: rolling_mean(ts, 60).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x85db190>
```



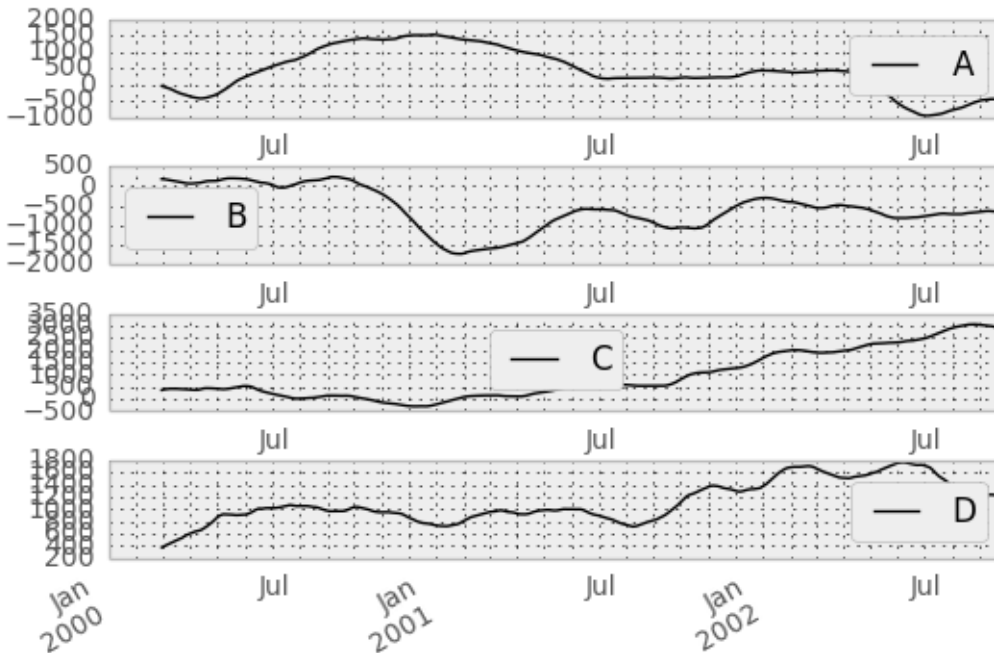
They can also be applied to `DataFrame` objects. This is really just syntactic sugar for applying the moving window operator to all of the `DataFrame`'s columns:

```
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)
```

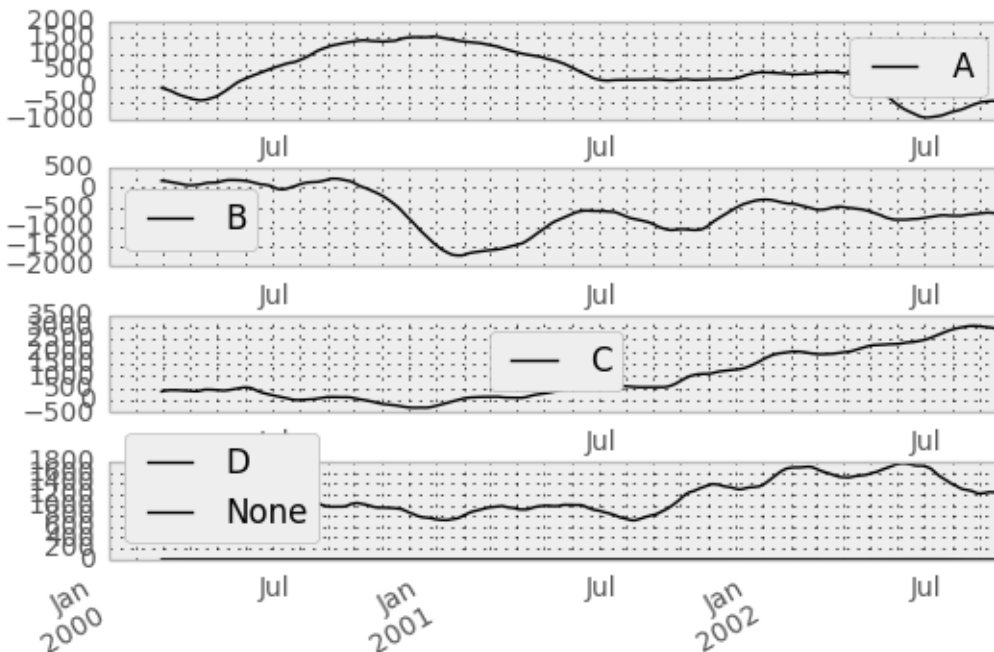
```
array([<matplotlib.axes.AxesSubplot object at 0x79706d0>,
      <matplotlib.axes.AxesSubplot object at 0x7b3db10>,
      <matplotlib.axes.AxesSubplot object at 0x8b52890>,
      <matplotlib.axes.AxesSubplot object at 0x69cb9d0>], dtype=object)
```



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')
<matplotlib.axes.AxesSubplot at 0x69cb9d0>
```



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')
```

```
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')
```

```
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)
```



```

2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05    -0.841164
2000-01-06    -0.779948
2000-01-07    -0.565487
2000-01-08    -0.502815
2000-01-09    -0.553755
2000-01-10    -0.472211
Freq: D, dtype: float64

```

For some windowing functions, additional parameters must be specified:

```
In [51]: rolling_window(ser, 5, 'gaussian', std=0.1)
```

```

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.

```
In [52]: rolling_window(ser, 5, 'boxcar')
```

```

2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05    -0.841164
2000-01-06    -0.779948
2000-01-07    -0.565487
2000-01-08    -0.502815
2000-01-09    -0.553755
2000-01-10    -0.472211
Freq: D, dtype: float64

```

```
In [53]: rolling_window(ser, 5, 'boxcar', center=True)
```

```

2000-01-01      NaN
2000-01-02      NaN
2000-01-03    -0.841164
2000-01-04    -0.779948
2000-01-05    -0.565487
2000-01-06    -0.502815
2000-01-07    -0.553755
2000-01-08    -0.472211
2000-01-09      NaN
2000-01-10      NaN
Freq: D, dtype: float64

```

```
In [54]: rolling_mean(ser, 5, center=True)
```

```
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
```

10.2.1 Binary rolling moments

`rolling_cov` and `rolling_corr` can compute moving window statistics about two Series or any combination of DataFrame/Series or DataFrame/DataFrame. Here is the behavior in each case:

- two Series: compute the statistic for the pairing
- DataFrame/Series: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame
- DataFrame/DataFrame: compute statistic for matching column names, returning a DataFrame

For example:

```
In [55]: df2 = df[:20]
```

```
In [56]: rolling_corr(df2, df2['B'], window=5)
```

```
          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]
```

10.2.2 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)
```

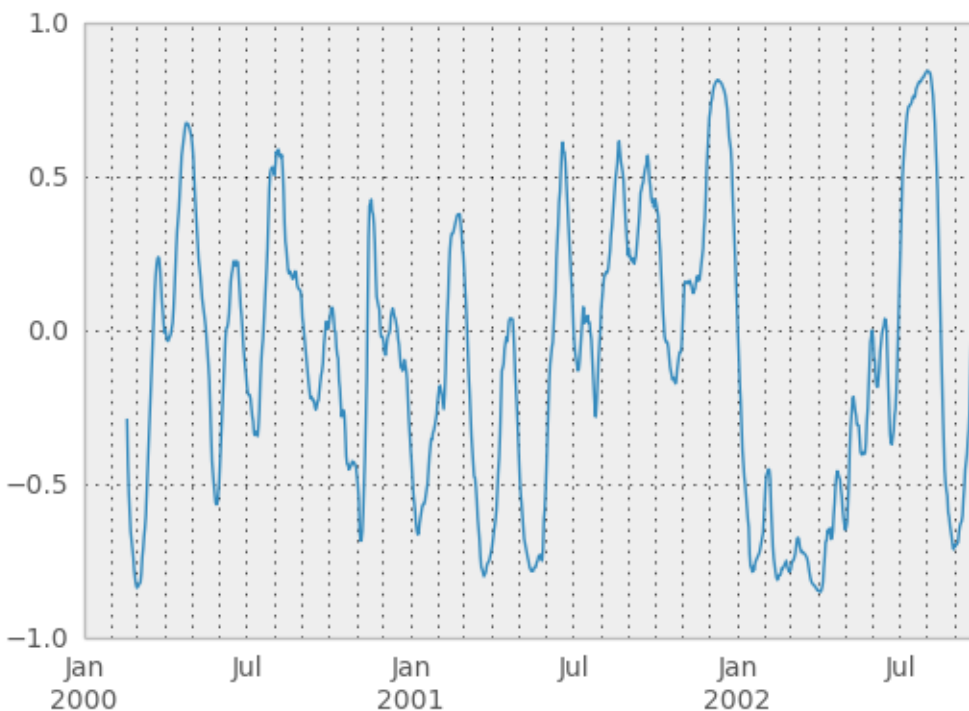
```
In [58]: correls[df.index[-50]]
```

	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 `ix` indexing:

```
In [59]: correls.ix[:, 'A', 'C'].plot()
<matplotlib.axes.AxesSubplot at 0x8f2b790>
```



10.3 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]
```

	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

```
[5 rows x 4 columns]
```

```
In [61]: expanding_mean(df)[:5]
```

```
          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
[5 rows x 4 columns]
```

Like the `rolling_` functions, the following methods are included in the pandas namespace or can be located in `pandas.stats.moments`.

Function	Description
<code>expanding_count</code>	Number of non-null observations
<code>expanding_sum</code>	Sum of values
<code>expanding_mean</code>	Mean of values
<code>expanding_median</code>	Arithmetic median of values
<code>expanding_min</code>	Minimum
<code>expanding_max</code>	Maximum
<code>expanding_std</code>	Unbiased standard deviation
<code>expanding_var</code>	Unbiased variance
<code>expanding_skew</code>	Unbiased skewness (3rd moment)
<code>expanding_kurt</code>	Unbiased kurtosis (4th moment)
<code>expanding_quantile</code>	Sample quantile (value at %)
<code>expanding_apply</code>	Generic apply
<code>expanding_cov</code>	Unbiased covariance (binary)
<code>expanding_corr</code>	Correlation (binary)
<code>expanding_corr_pairwise</code>	Pairwise correlation of DataFrame columns

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:

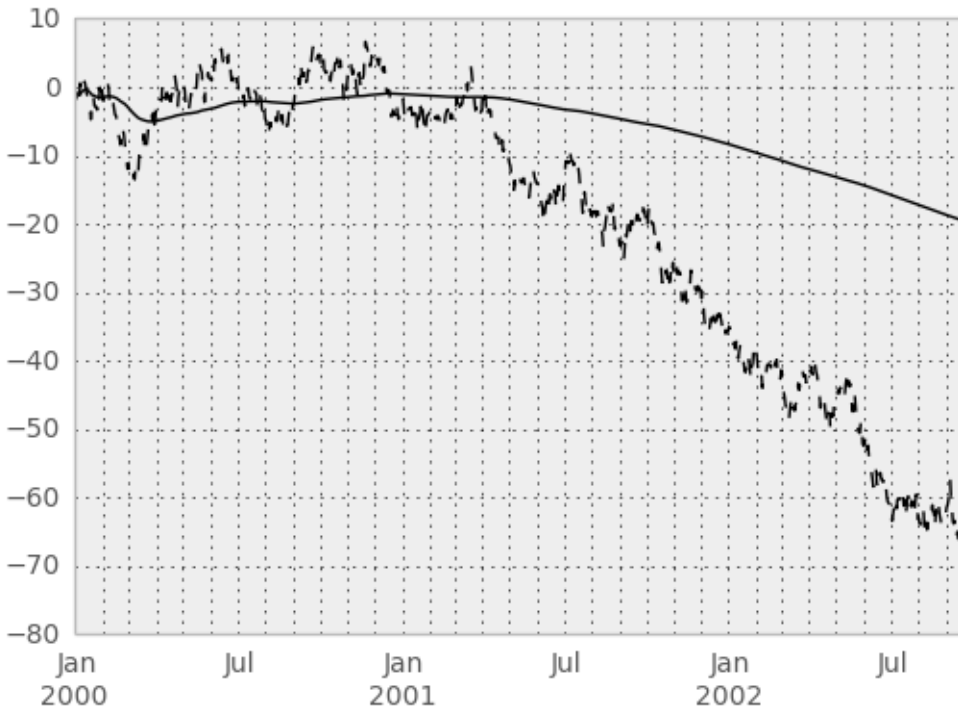
- `min_periods`: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No NaNs will be output once `min_periods` non-null data points have been seen.
- `freq`: optionally specify a *frequency string* or *DateOffset* to pre-conform the data to. Note that prior 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--')
<matplotlib.axes.AxesSubplot at 0x897ef10>
```

```
In [63]: expanding_mean(ts).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x897ef10>
```



10.4 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 α directly, it's easier to think about either the **span**, **center of mass** (**com**) or **halflife** of an EW moment:

$$\alpha = \begin{cases} \frac{2}{s+1}, s = \text{span} \\ \frac{1}{1+c}, c = \text{center of mass} \\ 1 - \exp^{\frac{\log 0.5}{h}}, h = \text{half life} \end{cases}$$

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 **com** = 9.5. **Halflife** is the period of time for the exponential weight to reduce to one half. Here is the list of functions available:

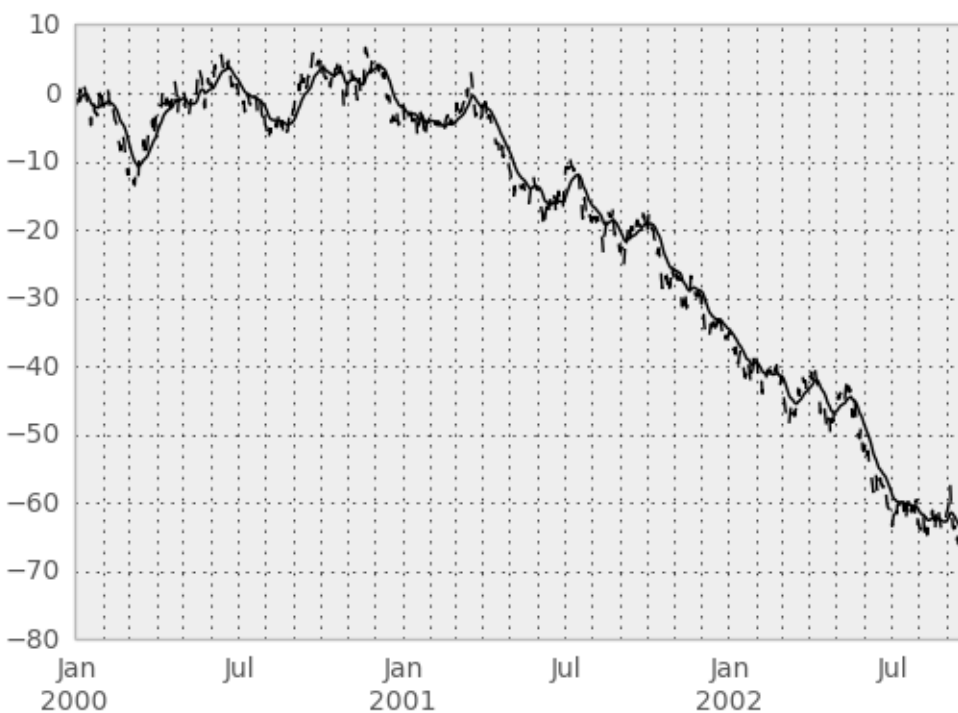
Function	Description
ewma	EW moving average
ewmvar	EW moving variance
ewmstd	EW moving standard deviation
ewmcorr	EW moving correlation
ewmcov	EW moving covariance

Here are an example for a univariate time series:

```
In [64]: plt.close('all')
```

```
In [65]: ts.plot(style='k--')
<matplotlib.axes.AxesSubplot at 0x7a21590>
```

```
In [66]: ewma(ts, span=20).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x7a21590>
```



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.

WORKING WITH MISSING DATA

In this section, we will discuss missing (also referred to as NA) values in pandas.

Note: The choice of using NaN internally to denote missing data was largely for simplicity and performance reasons. 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

11.1 Missing data basics

11.1.1 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
```

```
      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'])
```

```
In [6]: df2
```

```
      one      two      three four  five
a  0.059117  1.138469 -2.400634  bar   True
b      NaN      NaN      NaN  NaN   NaN
c -0.280853  0.025653 -1.386071  bar  False
d      NaN      NaN      NaN  NaN   NaN
e  0.863937  0.252462  1.500571  bar   True
f  1.053202 -2.338595 -0.374279  bar   True
g      NaN      NaN      NaN  NaN   NaN
h -2.359958 -1.157886 -0.551865  bar  False
[8 rows x 5 columns]
```

11.1.2 Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python None will arise and we wish to also consider that “missing” or “null”.

Until recently, for legacy reasons `inf` and `-inf` were also considered to be “null” in computations. This is no longer the case by default; use the `mode.use_inf_as_null` option to recover it. To make detecting missing values easier (and across different array dtypes), pandas provides the `isnull()` and `notnull()` functions, which are also methods on Series objects:

```
In [7]: df2['one']
```

```
a    0.059117
b         NaN
c   -0.280853
d         NaN
e    0.863937
f    1.053202
g         NaN
h   -2.359958
Name: one, dtype: float64
```

```
In [8]: isnull(df2['one'])
```

```
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool
```

```
In [9]: df2['four'].notnull()
```

```
a    True
b   False
c    True
d   False
e    True
f    True
g   False
```



```
h      True
Name: four, dtype: bool
```

Summary: NaN and None (in object arrays) are considered missing by the `isnull` and `notnull` functions. `inf` and `-inf` are no longer considered missing by default.

11.2 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
```

```
      one      two      three four  five  timestamp
a  0.059117  1.138469 -2.400634  bar   True  2012-01-01 00:00:00
c -0.280853  0.025653 -1.386071  bar  False  2012-01-01 00:00:00
e  0.863937  0.252462  1.500571  bar   True  2012-01-01 00:00:00
f  1.053202 -2.338595 -0.374279  bar   True  2012-01-01 00:00:00
h -2.359958 -1.157886 -0.551865  bar  False  2012-01-01 00:00:00
[5 rows x 6 columns]
```

```
In [13]: df2.ix[['a', 'c', 'h'], ['one', 'timestamp']] = np.nan
```

```
In [14]: df2
```

```
      one      two      three four  five  timestamp
a      NaN  1.138469 -2.400634  bar   True          NaT
c      NaN  0.025653 -1.386071  bar  False          NaT
e  0.863937  0.252462  1.500571  bar   True  2012-01-01 00:00:00
f  1.053202 -2.338595 -0.374279  bar   True  2012-01-01 00:00:00
h      NaN -1.157886 -0.551865  bar  False          NaT
[5 rows x 6 columns]
```

```
In [15]: df2.get_dtype_counts()
```

```
bool      1
datetime64[ns]  1
float64    3
object     1
dtype: int64
```

11.3 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [16]: a
```

```
      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

```
      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

```
      one  three      two
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]
```

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

In [19]: df

```
      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()
1.917139050150438

In [21]: df.mean(1)

```
a    -0.631082
c    -0.680209
e     0.872323
f    -0.553224
h    -0.854876
dtype: float64
```

In [22]: df.cumsum()

```
      one      two      three
a      NaN  1.138469 -2.400634
```

```
c      NaN    1.164122 -3.786705
e  0.863937    1.416584 -2.286134
f  1.917139   -0.922011 -2.660413
h      NaN   -2.079897 -3.212278
[5 rows x 3 columns]
```

11.3.1 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example.

11.4 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

11.4.1 Filling missing values: fillna

The `fillna` function can “fill in” NA values with non-null data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [23]: df2
```

```
      one      two      three four  five  timestamp
a      NaN  1.138469 -2.400634  bar   True         NaT
c      NaN  0.025653 -1.386071  bar  False         NaT
e  0.863937  0.252462  1.500571  bar   True  2012-01-01 00:00:00
f  1.053202 -2.338595 -0.374279  bar   True  2012-01-01 00:00:00
h      NaN -1.157886 -0.551865  bar  False         NaT
[5 rows x 6 columns]
```

```
In [24]: df2.fillna(0)
```

```
      one      two      three four  five  timestamp
a  0.000000  1.138469 -2.400634  bar   True  1970-01-01 00:00:00
c  0.000000  0.025653 -1.386071  bar  False  1970-01-01 00:00:00
e  0.863937  0.252462  1.500571  bar   True  2012-01-01 00:00:00
f  1.053202 -2.338595 -0.374279  bar   True  2012-01-01 00:00:00
h  0.000000 -1.157886 -0.551865  bar  False  1970-01-01 00:00:00
[5 rows x 6 columns]
```

```
In [25]: df2['four'].fillna('missing')
```

```
a    bar
c    bar
e    bar
f    bar
h    bar
Name: four, dtype: object
```

Fill gaps forward or backward

Using the same filling arguments as *reindexing*, we can propagate non-null values forward or backward:

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

```
      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 [27]: df.fillna(method='pad')
```

```
      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  1.053202 -1.157886 -0.551865
[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
```

```
      one      two      three
a  NaN  1.138469 -2.400634
c  NaN  0.025653 -1.386071
e  NaN      NaN      NaN
f  NaN      NaN      NaN
h  NaN -1.157886 -0.551865
[5 rows x 3 columns]
```

```
In [29]: df.fillna(method='pad', limit=1)
```

```
      one      two      three
a  NaN  1.138469 -2.400634
c  NaN  0.025653 -1.386071
e  NaN  0.025653 -1.386071
f  NaN      NaN      NaN
h  NaN -1.157886 -0.551865
[5 rows x 3 columns]
```

To remind you, these are the available filling methods:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

The `ffill()` function is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

11.4.2 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.

```
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
```

```
      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      NaN
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())
```

```
      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'])
```

```
      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')
```

```
      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]
```

11.4.3 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
```

```
      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)
```

```
Empty DataFrame
Columns: [one, two, three]
Index: []
[0 rows x 3 columns]
```

```
In [40]: df.dropna(axis=1)
```

```
      two      three
a  1.138469 -2.400634
c  0.025653 -1.386071
e  0.000000  0.000000
f  0.000000  0.000000
h -1.157886 -0.551865
[5 rows x 2 columns]
```

```
In [41]: df['one'].dropna()
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*.

11.4.4 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
```

```
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()
```

```
61
```

```
In [44]: ts.interpolate().count()
```

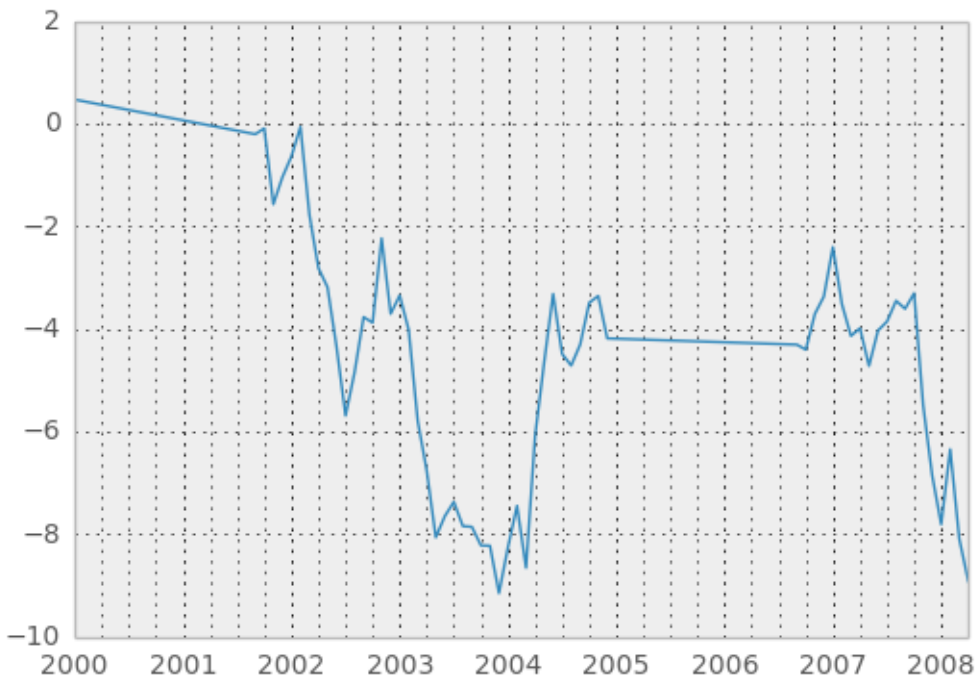
```
100
```

```
In [45]: plt.figure()
```

```
<matplotlib.figure.Figure at 0xf310610>
```

```
In [46]: ts.interpolate().plot()
```

```
<matplotlib.axes.AxesSubplot at 0xf310590>
```



Index aware interpolation is available via the `method` keyword:

```
In [47]: ts2
```

```
2000-01-31    0.469112
2000-02-29         NaN
2002-07-31   -5.689738
2005-01-31         NaN
2008-04-30   -8.916232
dtype: float64
```

```
In [48]: ts2.interpolate()
```

```
2000-01-31    0.469112
2000-02-29   -2.610313
2002-07-31   -5.689738
2005-01-31   -7.302985
2008-04-30   -8.916232
dtype: float64
```

```
In [49]: ts2.interpolate(method='time')
```

```
2000-01-31    0.469112
2000-02-29    0.273272
2002-07-31   -5.689738
2005-01-31   -7.095568
2008-04-30   -8.916232
dtype: float64
```

For a floating-point index, use `method='values'`:

```
In [50]: ser
```

```
0      0
1     NaN
10     10
dtype: float64
```

```
In [51]: ser.interpolate()
```

```
0      0
1      5
10     10
dtype: int64
```

```
In [52]: ser.interpolate(method='values')
```

```
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
```

```
      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()
```

```
      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]
```

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')
```

```
      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')
```

```
      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)
```

```
      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)
```

```
      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])
```

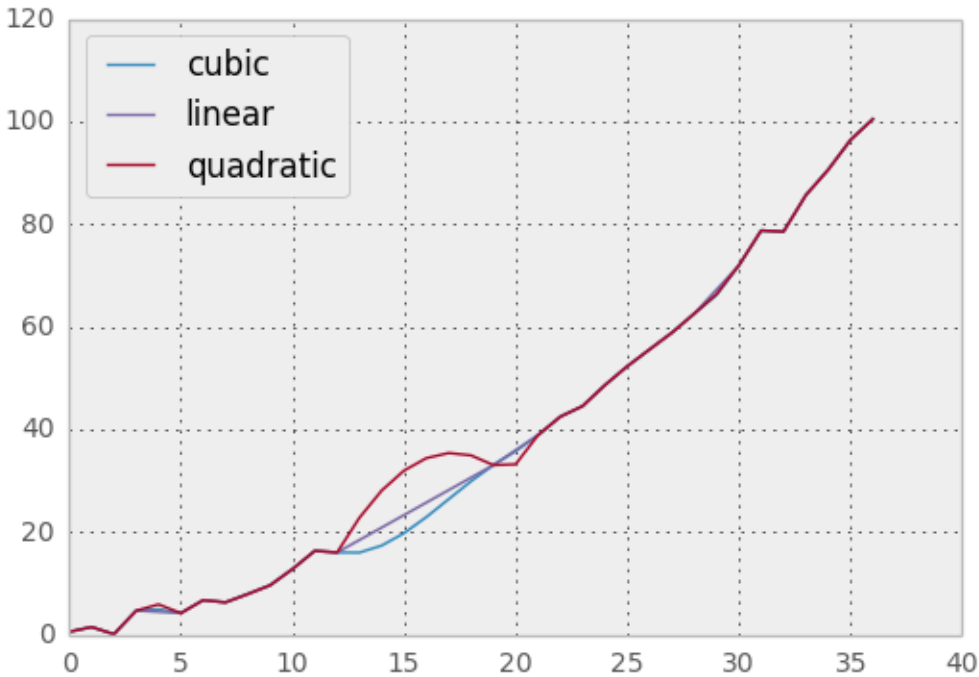
```
In [63]: ser[bad] = np.nan
```

```
In [64]: methods = ['linear', 'quadratic', 'cubic']
```

```
In [65]: df = DataFrame({m: ser.interpolate(method=m) for m in methods})
```

```
In [66]: plt.figure()
<matplotlib.figure.Figure at 0x8f00110>
```

```
In [67]: df.plot()
<matplotlib.axes.AxesSubplot at 0x685b710>
```



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]

49.00    0.471410
49.25    0.476841
49.50    0.481780
49.75    0.485998
50.00    0.489266
50.25    0.491814
50.50    0.493995
50.75    0.495763
51.00    0.497074
dtype: float64
```

Like other pandas fill methods, `interpolate` accepts a `limit` keyword argument. Use this to limit the number of consecutive interpolations, keeping NaN values for interpolations that are too far from the last valid observation:

```
In [72]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])

In [73]: ser.interpolate(limit=2)

0      1
1      3
2      5
```

```
3      7
4    NaN
5     11
dtype: float64
```

11.4.5 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:

```
In [74]: ser = Series([0., 1., 2., 3., 4.])
```

```
In [75]: ser.replace(0, 5)
```

```
0      5
1      1
2      2
3      3
4      4
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [76]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
```

```
0      4
1      3
2      2
3      1
4      0
dtype: float64
```

You can also specify a mapping dict:

```
In [77]: ser.replace({0: 10, 1: 100})
```

```
0      10
1     100
2      2
3      3
4      4
dtype: float64
```

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)
```

```
   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')
```

```
0    0
1    0
2    0
3    0
4    4
dtype: float64
```

11.4.6 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)
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  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)
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
[4 rows x 3 columns]
```

Replace a few different values (`list -> list`)

```
In [85]: df.replace(['a', '.'], ['b', nan])
```

```
   a    b    c
0  0    b    b
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
[4 rows x 3 columns]
```

list of regex -> list of regex

```
In [86]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
```

```
   a      b      c
0  0  {stuff  {stuff
1  1      b      b
2  2    dot    NaN
3  3    dot      d
[4 rows x 3 columns]
```

Only search in column 'b' (dict -> dict)

```
In [87]: df.replace({'b': '.'}, {'b': nan})
```

```
   a      b      c
0  0      a      a
1  1      b      b
2  2  NaN  NaN
3  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)

```
In [88]: df.replace({'b': r'\s*\.\s*'}, {'b': nan}, regex=True)
```

```
   a      b      c
0  0      a      a
1  1      b      b
2  2  NaN  NaN
3  3  NaN      d
[4 rows x 3 columns]
```

You can pass nested dictionaries of regular expressions that use `regex=True`

```
In [89]: df.replace({'b': {'b': r' '}}, regex=True)
```

```
   a      b      c
0  0      a      a
1  1      b      b
2  2      .  NaN
3  3      .      d
[4 rows x 3 columns]
```

or you can pass the nested dictionary like so

```
In [90]: df.replace(regex={'b': {r'\s*\.\s*': nan}})
```

```
   a      b      c
0  0      a      a
1  1      b      b
2  2  NaN  NaN
3  3  NaN      d
[4 rows x 3 columns]
```

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well

```
In [91]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
```

```
   a      b      c
0  0      a      a
```

```

1 1    b    b
2 2  .ty  NaN
3 3  .ty    d
[4 rows x 3 columns]

```

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex)

```
In [92]: df.replace([r'\s*\.\s*', r'a|b'], nan, regex=True)
```

```

   a    b    c
0  0 NaN NaN
1  1 NaN NaN
2  2 NaN NaN
3  3 NaN    d
[4 rows x 3 columns]

```

All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be

```
In [93]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=nan)
```

```

   a    b    c
0  0 NaN NaN
1  1 NaN NaN
2  2 NaN NaN
3  3 NaN    d
[4 rows x 3 columns]

```

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.

11.4.7 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)
```

```

      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'])
```

```
0      0      1
0      a -1.021415
1  0.4323957 -0.323580
2  0.4238247  0.799180
3  1.262614  0.751965
4      NaN      NaN
5      NaN      NaN
6 -0.4981742 -1.060799
7  0.5916665 -0.183257
8  1.019855 -1.482465
9      NaN      NaN
[10 rows x 2 columns]
```

```
In [99]: df[1].dtype
dtype('float64')
```

You can also operate on the DataFrame in place

```
In [100]: df.replace(1.5, nan, inplace=True)
```

11.5 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:

data type	Cast to
integer	float
boolean	object
float	no cast
object	no cast

For example:

```
In [101]: s = Series(randn(5), index=[0, 2, 4, 6, 7])
```

```
In [102]: s > 0
```

```
0      True
2      True
4      True
6      True
7      True
dtype: bool
```

```
In [103]: (s > 0).dtype
dtype('bool')
```



```
In [104]: crit = (s > 0).reindex(list(range(8)))
```

```
In [105]: crit
```

```
0    True
1    NaN
2    True
3    NaN
4    True
5    NaN
6    True
7    True
dtype: object
```

```
In [106]: crit.dtype
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)
<ipython-input-108-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]
/home/docbuild/CI/pandas/pandas/core/series.pyc in __getitem__(self, key)
    510         key = list(key)
    511
--> 512         if _is_bool_indexer(key):
    513             key = _check_bool_indexer(self.index, key)
    514
/home/docbuild/CI/pandas/pandas/core/common.pyc in _is_bool_indexer(key)
    1655         if len(key) and not lib.is_bool_array(key):
    1656             if isnull(key).any():
-> 1657                 raise ValueError('cannot index with vector containing '
    1658                                     'NA / NaN values')
    1659         return False
ValueError: cannot index with vector containing NA / NaN values
```

However, these can be filled in using **fillna** and it will work fine:

```
In [109]: reindexed[crit.fillna(False)]
```

```
0    0.126504
2    0.696198
4    0.697416
6    0.601516
7    0.003659
dtype: float64
```

```
In [110]: reindexed[crit.fillna(True)]
```

```
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:

```
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

12.1 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:

```
>>> 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:

```
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)})
...:
```

```
In [2]: df
```

```
   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
[8 rows x 4 columns]
```

We could naturally group by either the A or B columns or both:

```
In [3]: grouped = df.groupby('A')
```

```
In [4]: grouped = df.groupby(['A', 'B'])
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [5]: def get_letter_type(letter):
...:     if letter.lower() in 'aeiou':
...:         return 'vowel'
...:     else:
...:         return 'consonant'
...:
```

```
In [6]: grouped = df.groupby(get_letter_type, axis=1)
```

Starting with 0.8, pandas Index objects now supports duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```
In [7]: lst = [1, 2, 3, 1, 2, 3]

In [8]: s = Series([1, 2, 3, 10, 20, 30], lst)

In [9]: grouped = s.groupby(level=0)

In [10]: grouped.first()

1    1
2    2
3    3
dtype: int64

In [11]: grouped.last()

1    10
2    20
3    30
dtype: int64

In [12]: grouped.sum()

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.

12.1.1 GroupBy object attributes

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```
In [13]: df.groupby('A').groups
{'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}

In [14]: df.groupby(get_letter_type, axis=1).groups
{'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [15]: grouped = df.groupby(['A', 'B'])

In [16]: grouped.groups

{('bar', 'one'): [1],
```

```
('bar', 'three'): [3],
('bar', 'two'): [5],
('foo', 'one'): [0, 6],
('foo', 'three'): [7],
('foo', 'two'): [2, 4]}
```

```
In [17]: len(grouped)
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()
```

```
      Y
X
A      7
B      3
[2 rows x 1 columns]
```

```
In [20]: df2.groupby(['X'], sort=False).sum()
```

```
      Y
X
B      3
A      7
[2 rows x 1 columns]
```

GroupBy will tab complete column names (and other attributes)

```
In [21]: df
```

```
      gender  height  weight
2000-01-01  male  42.849980  157.500553
2000-01-02  male  49.607315  177.340407
2000-01-03  male  56.293531  171.524640
2000-01-04  female  48.421077  144.251986
2000-01-05  male  46.556882  152.526206
2000-01-06  female  68.448851  168.272968
2000-01-07  male  70.757698  136.431469
2000-01-08  female  58.909500  176.499753
2000-01-09  female  76.435631  174.094104
2000-01-10  male  45.306120  177.540920
[10 rows x 3 columns]
```

```
In [22]: gb = df.groupby('gender')
```

```
In [23]: gb.<TAB>
```

gb.agg	gb.boxplot	gb.cummin	gb.describe	gb.filter	gb.get_group	gb.height	gb
gb.aggregate	gb.count	gb.cumprod	gb.dtype	gb.first	gb.groups	gb.hist	gb
gb.apply	gb.cummax	gb.cumsum	gb.fillna	gb.gender	gb.head	gb.indices	gb

12.1.2 GroupBy with MultiIndex

With *hierarchically-indexed data*, it's quite natural to group by one of the levels of the hierarchy.

In [24]: s

```
first  second
bar    one   -0.575247
        two    0.254161
baz    one   -1.143704
        two    0.215897
foo    one    1.193555
        two   -0.077118
qux    one   -0.408530
        two   -0.862495
dtype: float64
```

In [25]: grouped = s.groupby(level=0)

In [26]: grouped.sum()

```
first
bar    -0.321085
baz    -0.927807
foo     1.116437
qux    -1.271025
dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

In [27]: s.groupby(level='second').sum()

```
second
one    -0.933926
two    -0.469555
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

In [28]: s.sum(level='second')

```
second
one    -0.933926
two    -0.469555
dtype: float64
```

Also as of v0.6, grouping with multiple levels is supported.

In [29]: s

```
first  second  third
bar    doo     one    1.346061
        two    1.511763
baz    bee     one    1.627081
        two   -0.990582
foo    bop     one   -0.441652
        two    1.211526
qux    bop     one    0.268520
        two    0.024580
dtype: float64
```

In [30]: s.groupby(level=['first', 'second']).sum()

```
first  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.

12.1.3 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'])
<pandas.core.groupby.SeriesGroupBy object at 0xcf9f6d0>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

12.2 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
1 bar  one -0.042379 -0.089329
3 bar three -0.009920 -0.945867
5 bar  two  0.495767  1.956030
[3 rows x 4 columns]
foo
   A      B      C      D
0 foo  one -0.919854 -1.131345
2 foo  two  1.247642  0.337863
4 foo  two  0.290213 -0.932132
6 foo  one  0.362949  0.017587
7 foo three  1.548106 -0.016692
[5 rows x 4 columns]
```

In the case of grouping by multiple keys, the group name will be a tuple:


```
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:`.

12.3 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:

```
In [38]: grouped = df.groupby('A')
```

```
In [39]: grouped.agg(np.sum)
```

```

      C      D
A
bar  0.443469  0.920834
foo  2.529056 -1.724719
[2 rows x 2 columns]
```

```
In [40]: grouped = df.groupby(['A', 'B'])
```

```
In [41]: grouped.agg(np.sum)
```

```

      C      D
A  B
bar one -0.042379 -0.089329
```

```
three -0.009920 -0.945867
two    0.495767  1.956030
foo one -0.556905 -1.113758
three  1.548106 -0.016692
two    1.537855 -0.594269
[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)
```

```
   A      B      C      D
0 bar  one -0.042379 -0.089329
1 bar  three -0.009920 -0.945867
2 bar  two  0.495767  1.956030
3 foo  one -0.556905 -1.113758
4 foo  three  1.548106 -0.016692
5 foo  two  1.537855 -0.594269
[6 rows x 4 columns]
```

```
In [44]: df.groupby('A', as_index=False).sum()
```

```
   A      C      D
0 bar  0.443469  0.920834
1 foo  2.529056 -1.724719
[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()
```

```
   A      B      C      D
0 bar  one -0.042379 -0.089329
1 bar  three -0.009920 -0.945867
2 bar  two  0.495767  1.956030
3 foo  one -0.556905 -1.113758
4 foo  three  1.548106 -0.016692
5 foo  two  1.537855 -0.594269
[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()
```

```
A      B
bar  one    1
     three  1
     two    1
foo  one    2
     three  1
     two    2
dtype: int64
```

12.3.1 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])
```

```
      sum      mean      std
A
bar  0.443469  0.147823  0.301765
foo  2.529056  0.505811  0.966450
[2 rows x 3 columns]
```

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})
.....:
```

```
      result2  result1
A
bar  0.306945  0.920834
foo -0.344944 -1.724719
[2 rows x 2 columns]
```

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])
```

```
      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
[2 rows x 6 columns]
```

Passing a dict of functions has different behavior by default, see the next section.

12.3.2 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)})
.....:
```

```
      C      D
A
bar  0.443469  1.490982
foo  2.529056  0.645875
[2 rows x 2 columns]
```

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'})
```

```
           C          D
A
bar  0.443469  1.490982
foo  2.529056  0.645875
[2 rows x 2 columns]
```

12.3.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, and `std`, have optimized Cython implementations:

```
In [53]: df.groupby('A').sum()
```

```
           C          D
A
bar  0.443469  0.920834
foo  2.529056 -1.724719
[2 rows x 2 columns]
```

```
In [54]: df.groupby(['A', 'B']).mean()
```

```
           C          D
A  B
bar one  -0.042379 -0.089329
    three -0.009920 -0.945867
    two   0.495767  1.956030
foo one  -0.278452 -0.556879
    three  1.548106 -0.016692
    two   0.768928 -0.297134
[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).

12.4 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:

```
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()
```

```
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()
```

```
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()
```

```
2000    0.442441
2001    0.526246
2002    0.459365
dtype: float64
```

```
In [65]: grouped.std()
```

```
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()
```

```
2000    1.168208e-15
2001    1.454544e-15
2002    1.726657e-15
dtype: float64
```

```
In [68]: grouped_trans.std()
```

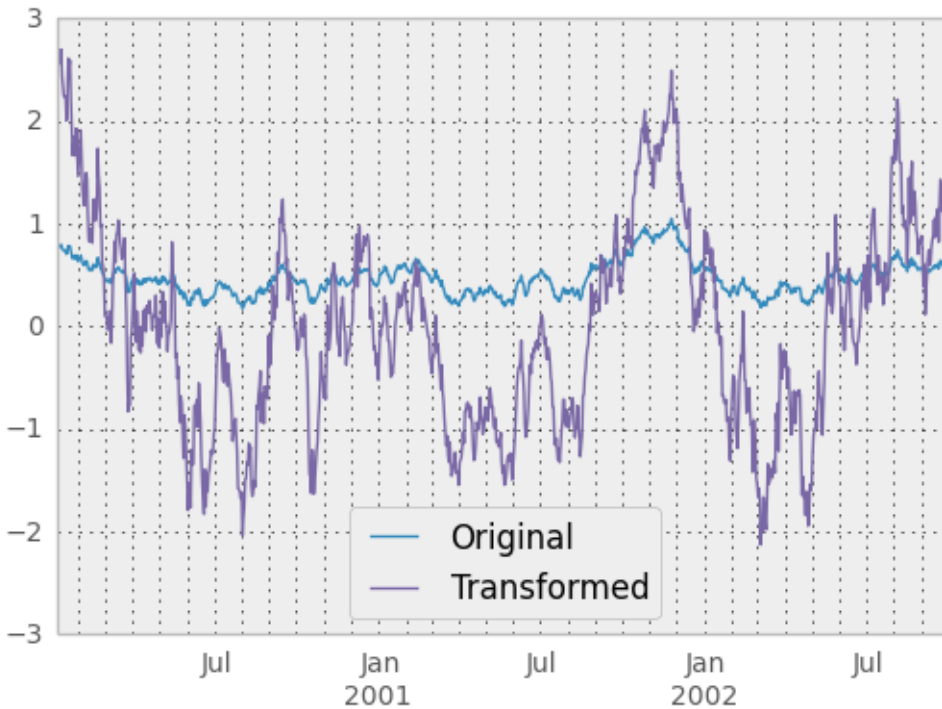
```
2000    1
2001    1
2002    1
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [69]: compare = DataFrame({'Original': ts, 'Transformed': transformed})
```

```
In [70]: compare.plot()
```

```
<matplotlib.axes.AxesSubplot at 0xb641d10>
```



Another common data transform is to replace missing data with the group mean.

```
In [71]: data_df
```

	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	1.376106	-0.645129
7	0.798630	-1.729858	0.392067
8	-0.347401	-0.429063	1.792958
9	-0.431059	1.605289	-3.302946
10	0.434332	-1.302198	0.756527
11	-0.349926	NaN	0.304228
12	NaN	-0.024779	NaN
13	1.026076	-0.151723	-1.136601
14	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()
```

	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 [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
```

```
      A      B      C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603
[4 rows x 3 columns]
```

```
In [80]: grouped_trans.mean() # transformation did not change group means
```

```
      A      B      C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603
[4 rows x 3 columns]
```

```
In [81]: grouped.count() # original has some missing data points
```

```
      A      B      C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217
[4 rows x 3 columns]
```

```
In [82]: grouped_trans.count() # counts after transformation
```

```
      A      B      C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258
[4 rows x 3 columns]
```

```
In [83]: grouped_trans.size() # Verify non-NA count equals group size
```

```
GR    228
JP    267
UK    247
US    258
dtype: int64
```

12.5 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)

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)

   A  B
2  2  b
3  3  b
4  4  b
5  5  b
[4 rows x 2 columns]
```

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [88]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)

   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)

   A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
[4 rows x 3 columns]
```


12.6 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())
```

```
      B      C      D
A
bar NaN  0.301765  1.490982
foo NaN  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.std()
```

```
      C      D
A
bar  0.301765  1.490982
foo  0.966450  0.645875
[2 rows x 2 columns]
```

What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```
In [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')
```

```
      A      B      C
2000-01-01  NaN  NaN  NaN
2000-01-02 -0.353501 -0.080957 -0.876864
2000-01-03 -0.353501 -0.080957 -0.876864
2000-01-04  0.050976  0.044273 -0.559849
2000-01-05  0.050976  0.044273 -0.559849
2000-01-06  0.030091  0.186460 -0.680149
2000-01-07  0.030091  0.186460 -0.680149
2000-01-08 -0.882655  0.661310  1.317217
2000-01-09 -0.882655  0.661310  1.317217
2000-01-10  0.008021  0.572938  0.309048
2000-01-11  0.008021  0.572938  0.309048
2000-01-12 -0.818637 -2.130013 -1.346086
2000-01-13 -0.818637 -2.130013 -1.346086
2000-01-14  0.315112 -1.667438 -0.363184
2000-01-15  0.315112 -1.667438 -0.363184
```

```
...      ...      ...  
[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

12.7 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
```

```
   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())
```

```
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)
```

```

      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

```

In [104]: def f(x):
.....:     return Series([ x, x**2 ], index = ['x', 'x^s'])
.....:

```

```

In [105]: s = Series(np.random.rand(5))

```

```

In [106]: s

```

```

0    0.321438
1    0.493496
2    0.139505
3    0.910103
4    0.194158
dtype: float64

```

```

In [107]: s.apply(f)

```

```

      x      x^s
0  0.321438  0.103323
1  0.493496  0.243538
2  0.139505  0.019462
3  0.910103  0.828287
4  0.194158  0.037697
[5 rows x 2 columns]

```

12.8 Other useful features

12.8.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:

```

In [108]: df

```

```

      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]
```

Supposed we wished to compute the standard deviation grouped by the A column. There is a slight problem, namely that we don't care about the data in column B. We refer to this as a “nuisance” column. If the passed aggregation function can't be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [109]: df.groupby('A').std()
```

```
           C          D
A
bar  0.301765  1.490982
foo  0.966450  0.645875
[2 rows x 2 columns]
```

12.8.2 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).

12.8.3 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:

```
In [110]: data = Series(np.random.randn(100))
```

```
In [111]: factor = qcut(data, [0, .25, .5, .75, 1.])
```

```
In [112]: data.groupby(factor).mean()
```

```
[-2.644, -0.523]    -1.362896
(-0.523, 0.0296]   -0.260266
(0.0296, 0.654]    0.361802
(0.654, 2.21]      1.073801
dtype: float64
```

12.8.4 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:

```
In [113]: df = pd.DataFrame(list('aaabba'), columns=['A'])
```

```
In [114]: df
```

```
   A
0  a
1  a
2  a
3  b
4  b
5  a
```

```
[6 rows x 1 columns]
```

```
In [115]: df.groupby('A').cumcount()
```

```
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
```

```
In [116]: df.groupby('A').cumcount(ascending=False) # kward only
```

```
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```


MERGE, JOIN, AND CONCATENATE

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

13.1 Concatenating objects

The `concat` function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say “if any” because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```
In [1]: df = DataFrame(np.random.randn(10, 4))
```

```
In [2]: df
```

```
      0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -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
```

```
      0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
```

```
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -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]
```

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'])
```

```
In [7]: concatenated
```

```
      0      1      2      3
first 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
second 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
third  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]
```


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']

           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.

13.1.1 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 randn

In [10]: df = DataFrame(np.random.randn(10, 4), columns=['a', 'b', 'c', 'd'],
.....:                  index=[randn(5) for _ in range(10)])
.....:

In [11]: df

           a          b          c          d
IQWEO -1.294524  0.413738  0.276662 -0.472035
W940c -0.013960 -0.362543 -0.006154 -0.923061
uubhy  0.895717  0.805244 -1.206412  2.565646
rkSzV  1.431256  1.340309 -1.170299 -0.226169
PJyaL  0.410835  0.813850  0.132003 -0.827317
5GJwg -0.076467 -1.187678  1.130127 -1.436737
bnvuO -1.413681  1.607920  1.024180  0.569605
t5rMS  0.875906 -2.211372  0.974466 -2.006747
8Xav9 -0.410001 -0.078638  0.545952 -1.219217
ItILU -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)
.....:

           a          b          c          d
5GJwg -0.076467 -1.187678  1.130127 -1.436737
8Xav9         NaN         NaN         NaN -1.219217
IQWEO -1.294524  0.413738         NaN         NaN
ItILU         NaN         NaN         NaN -0.727707
```

```
PJyaL 0.410835 0.813850 0.132003 -0.827317
W940c -0.013960 -0.362543      NaN      NaN
bnvuO -1.413681 1.607920 1.024180 0.569605
rkSzV 1.431256 1.340309 -1.170299 -0.226169
t5rMS      NaN      NaN 0.974466 -2.006747
uubhy 0.895717 0.805244 -1.206412      NaN
[10 rows x 4 columns]
```

Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```
In [13]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
.....:          df.ix[-7:, ['d']]], axis=1, join='inner')
.....:
```

```
      a      b      c      d
rkSzV 1.431256 1.340309 -1.170299 -0.226169
PJyaL 0.410835 0.813850 0.132003 -0.827317
5GJwg -0.076467 -1.187678 1.130127 -1.436737
bnvuO -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:

```
In [14]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
.....:          df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
.....:
```

```
      a      b      c      d
IQWEO -1.294524 0.413738      NaN      NaN
W940c -0.013960 -0.362543      NaN      NaN
uubhy 0.895717 0.805244 -1.206412      NaN
rkSzV 1.431256 1.340309 -1.170299 -0.226169
PJyaL 0.410835 0.813850 0.132003 -0.827317
5GJwg -0.076467 -1.187678 1.130127 -1.436737
bnvuO -1.413681 1.607920 1.024180 0.569605
t5rMS      NaN      NaN 0.974466 -2.006747
8Xav9      NaN      NaN      NaN -1.219217
ItILU      NaN      NaN      NaN -0.727707
[10 rows x 4 columns]
```

13.1.2 Concatenating using `append`

A useful shortcut to `concat` are the `append` instance methods on `Series` and `DataFrame`. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

```
In [15]: s = Series(randn(10), index=np.arange(10))
```

```
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)
```

```
0    -0.121306
1    -0.097883
2     0.695775
3     0.341734
```

```

4      0.959726
6     -0.619976
7      0.149748
8     -0.732339
9      0.687738
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

```

```

          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

```

```

          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)

```

```

          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])

```

```

          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  0.014871

```

```
2000-01-05  3.357427 -0.317441 -1.236269  0.896171
2000-01-06 -0.487602 -0.082240 -2.182937  0.380396
[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.

13.1.3 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
```

```
      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 rows x 4 columns]
```

```
In [32]: df2
```

```
      A      B      C      D
0  0.082423 -0.055758  0.536580 -0.489682
1  0.369374 -0.034571 -2.484478 -0.281461
2  0.030711  0.109121  1.126203 -0.977349
[3 rows x 4 columns]
```

To do this, use the `ignore_index` argument:

```
In [33]: concat([df1, df2], ignore_index=True)
```

```
      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]
```

This is also a valid argument to `DataFrame.append`:

```
In [34]: df1.append(df2, ignore_index=True)
```

```
      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]

```

13.1.4 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
```

```

      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
```

```

      one      two      three
      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]

```

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)
```

	one		three	two
	0	1	3	2
0	1.474071	-0.064034	0.781836	-1.282782
1	-1.071357	0.441153	0.583787	2.353925
2	0.221471	-0.744471	1.729689	0.758527
3	-0.964980	-0.845696	1.846883	-1.340896
4	-1.328865	1.682706	0.888782	-1.717693
5	0.228440	0.901805	0.520260	1.171216
6	-1.197071	-1.066969	-0.858447	-0.303421
7	0.306996	-0.028665	1.574159	0.384316
8	1.588931	0.476720	-0.242861	0.473424
9	-0.014805	-0.284319	-1.461665	0.650776

[10 rows x 4 columns]

```
In [42]: concat(pieces, keys=['three', 'two'])
```

	2	3
three 0	NaN	0.781836
1	NaN	0.583787
2	NaN	1.729689
3	NaN	1.846883
4	NaN	0.888782
5	NaN	0.520260
6	NaN	-0.858447
7	NaN	1.574159
8	NaN	-0.242861
9	NaN	-1.461665
two 0	-1.282782	NaN
1	2.353925	NaN
2	0.758527	NaN
3	-1.340896	NaN
4	-1.717693	NaN
	...	...

[20 rows x 2 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
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
```

group_key	one		two	three
	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]

```

```

In [46]: result.columns.levels
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.

13.1.5 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

```

```

      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 rows x 4 columns]

```

```

In [49]: s = df.xs(3)

```

```

In [50]: df.append(s, ignore_index=True)

```

```

      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
```

	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	0.967661	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]

13.2 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:

```
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.

13.2.1 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
```

```
   key  lval
0  foo     1
1  foo     2
[2 rows x 2 columns]
```

```
In [58]: right
```

```
   key  rval
0  foo     4
1  foo     5
[2 rows x 2 columns]
```

```
In [59]: merge(left, right, on='key')
```

```
   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')
```

```
   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
[5 rows x 4 columns]
```

```
In [63]: merge(left, right, how='inner')
```

```
   key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  bar  one     3     6
[3 rows x 4 columns]
```

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:

Merge method	SQL Join Name	Description
left	LEFT OUTER JOIN	Use keys from left frame only
right	RIGHT OUTER JOIN	Use keys from right frame only
outer	FULL OUTER JOIN	Use union of keys from both frames
inner	INNER JOIN	Use intersection of keys from both frames

13.2.2 Joining on index

`DataFrame.join` is a convenient method for combining the columns of two potentially differently-indexed `DataFrame`s 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
```

```
      A      B
1 -2.461467 -1.553902
2  1.771740 -0.670027
3 -3.201750  0.792716
4 -0.747169 -0.309038
5  0.936527  1.255746
6  0.062297 -0.110388
7  0.077849  0.629498
[7 rows x 2 columns]
```

```
In [68]: df2
```

```
      C      D
0  0.377953  0.493672
1  2.015523 -1.833722
2  0.049307 -0.521493
3  0.146111  1.903247
4  0.393876  1.861468
5 -2.655452  1.219492
[6 rows x 2 columns]
```

```
In [69]: df1.join(df2)
```

```
      A      B      C      D
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
[7 rows x 4 columns]
```

```
In [70]: df1.join(df2, how='outer')
```

```
      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 rows x 4 columns]
```

```
In [71]: df1.join(df2, how='inner')
```

```
      A      B      C      D
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
```

```
[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')
```

```
      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 rows x 4 columns]
```

13.2.3 Joining key columns on an index

`join` takes an optional `on` argument which may be a column or multiple column names, which specifies that the passed `DataFrame` is to be aligned on that column in the `DataFrame`. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
merge(left, right, left_on=key_or_keys, right_index=True,
      how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the `DataFrame`'s is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [73]: df['key'] = ['foo', 'bar'] * 4
```

```
In [74]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
.....:                        columns=['j1', 'j2'])
.....:
```

```
In [75]: df
```

```
      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
```

```
      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')
```

```

      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)
.....:
```

```

      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]
```

To join on multiple keys, the passed DataFrame must have a MultiIndex:

```
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
```

```
data key1 key2
```

```
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 -1.260006  baz     two
7 -1.132896  qux     two
8 -2.006481  qux   three
9  0.301016  snap    one
[10 rows x 3 columns]
```

In [86]: `to_join`

```
      j_one  j_two  j_three
first second
foo  one    0.464794 -0.309337 -0.649593
     two    0.683758 -0.643834  0.421287
     three  1.032814 -1.290493  0.787872
bar   one    1.515707 -0.276487 -0.223762
     two    1.397431  1.503874 -0.478905
baz   two   -0.135950 -0.730327 -0.033277
     three  0.281151 -1.298915 -2.819487
qux   one   -0.851985 -1.106952 -0.937731
     two   -1.537770  0.555759 -2.277282
     three -0.390201  1.207122  0.178690
[10 rows x 3 columns]
```

Now this can be joined by passing the two key column names:

In [87]: `data.join(to_join, on=['key1', 'key2'])`

```
      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')`

```
      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   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
```

```
[7 rows x 6 columns]
```

As you can see, this drops any rows where there was no match.

13.2.4 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'])
```

```
   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.

13.2.5 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
```

```
   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
```

```
   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')
```

```
   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]
```

13.2.6 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
```

```
      A      B
0 -0.308853 -0.681087
1 -2.461467 -1.553902
2  1.771740 -0.670027
3 -3.201750  0.792716
4 -0.747169 -0.309038
5  0.936527  1.255746
6  0.062297 -0.110388
7  0.077849  0.629498
[8 rows x 2 columns]
```

```
In [99]: df1.join([df2, df3])
```

```
      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]
```

13.2.7 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)
```

```
      0      1      2
0  NaN    3  5.0
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 inplace:

```
In [103]: df1.update(df2)
```

```
In [104]: df1
```

```
      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

14.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```
In [1]: df
```

	date	variable	value
0	2000-01-03 00:00:00	A	0.469112
1	2000-01-04 00:00:00	A	-0.282863
2	2000-01-05 00:00:00	A	-1.509059
3	2000-01-03 00:00:00	B	-1.135632
4	2000-01-04 00:00:00	B	1.212112
5	2000-01-05 00:00:00	B	-0.173215
6	2000-01-03 00:00:00	C	0.119209
7	2000-01-04 00:00:00	C	-1.044236
8	2000-01-05 00:00:00	C	-0.861849
9	2000-01-03 00:00:00	D	-2.104569
10	2000-01-04 00:00:00	D	-0.494929
11	2000-01-05 00:00:00	D	1.071804

[12 rows x 3 columns]

For the curious here is how the above DataFrame was created:

```
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[df['variable'] == 'A']
```

	date	variable	value
0	2000-01-03 00:00:00	A	0.469112
1	2000-01-04 00:00:00	A	-0.282863
2	2000-01-05 00:00:00	A	-1.509059

[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:

```
In [3]: df.pivot(index='date', columns='variable', values='value')
```

```
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:

```
In [4]: df['value2'] = df['value'] * 2
```

```
In [5]: pivoted = df.pivot('date', 'variable')
```

```
In [6]: pivoted
```

```
variable          value          B          C          D          value2          \
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

variable          C          D
date
2000-01-03  0.238417 -4.209138
2000-01-04 -2.088472 -0.989859
2000-01-05 -1.723698  2.143608
[3 rows x 8 columns]
```

You of course can then select subsets from the pivoted `DataFrame`:

```
In [7]: pivoted['value2']
```

```
variable          A          B          C          D
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.

14.2 Reshaping by stacking and unstacking

Closely related to the `pivot` function are the related `stack` and `unstack` functions currently available on `Series` and `DataFrame`. These functions are designed to work together with `MultiIndex` objects (see the section on *hierarchical indexing*). Here are essentially what these functions do:

- `stack`: “pivot” a level of the (possibly hierarchical) column labels, returning a `DataFrame` with an index with a new inner-most level of row labels.

- `unstack`: inverse operation from `stack`: “pivot” a level of the (possibly hierarchical) row index to the column axis, producing a reshaped DataFrame with a new inner-most level of column labels.

The clearest way to explain is by example. Let’s take a prior example data set from the hierarchical indexing section:

```
In [8]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
...:                        'foo', 'foo', 'qux', 'qux'],
...:                     ['one', 'two', 'one', 'two',
...:                     'one', 'two', 'one', 'two']]))
...:

In [9]: index = 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
```

```
          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:

```
In [13]: stacked = df2.stack()

In [14]: stacked
```

```
first second
bar  one    A    0.721555
     two    B   -0.706771
     two    A   -1.039575
     two    B    0.271860
baz   one    A   -0.424972
     two    B    0.567020
     two    A    0.276232
     two    B   -1.087401
dtype: float64
```

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of `stack` is `unstack`, which by default unstacks the **last level**:

```
In [15]: stacked.unstack()

          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)
```

```
second      one      two
first
bar   A   0.721555 -1.039575
      B  -0.706771  0.271860
baz   A  -0.424972  0.276232
      B   0.567020 -1.087401
[4 rows x 2 columns]
```

```
In [17]: stacked.unstack(0)
```

```
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')
```

```
second      one      two
first
bar   A   0.721555 -1.039575
      B  -0.706771  0.271860
baz   A  -0.424972  0.276232
      B   0.567020 -1.087401
[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
```

```
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
```

```
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')
```

```
animal          cat      dog
first second exp
bar   one    A   -0.370647  0.844885
      two    B   -1.344312 -1.157892
      two    A    1.075770 -1.469388
      two    B    1.643563 -0.109050
baz   one    A    0.357021 -0.968914
      two    B   -1.776904 -0.674600
foo   one    A   -0.013960 -0.923061
      two    B   -0.006154 -0.362543
      two    A    0.895717  2.565646
      two    B   -1.206412  0.805244
qux   two    A    0.410835 -0.827317
      two    B    0.132003  0.813850
[12 rows x 2 columns]
```

```
In [24]: df2.stack('animal')
```

```
exp          A      B
first second animal
bar   one    cat   -0.370647 -1.344312
      two    dog    0.844885 -1.157892
      two    cat    1.075770  1.643563
      two    dog   -1.469388 -0.109050
baz   one    cat    0.357021 -1.776904
      two    dog   -0.968914 -0.674600
foo   one    cat   -0.013960 -0.006154
      two    dog   -0.923061 -0.362543
      two    cat    0.895717 -1.206412
      two    dog    2.565646  0.805244
qux   two    cat    0.410835  0.132003
      two    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)
```

```
exp          A      B
animal      cat      dog
first      bar      baz      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)
```

```
exp      A      B      A \
animal   cat      dog
second  one    two    one    two
first
bar    -0.370647  1.075770 -1.157892 -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]
```

14.3 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 [27]: cheese = DataFrame({'first' : ['John', 'Mary'],
.....:                      'last'  : ['Doe', 'Bo'],
.....:                      'height' : [5.5, 6.0],
.....:                      'weight' : [130, 150]})
.....:
```

```
In [28]: cheese
```

```
   first  height last  weight
0  John     5.5  Doe    130
1  Mary     6.0   Bo    150
[2 rows x 4 columns]
```

```
In [29]: melt(cheese, id_vars=['first', 'last'])
```

```
   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')
```

```
   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 : .7},
.....:                      "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
.....:                      "X"      : dict(zip(range(3), np.random.randn(3)))
.....:                      })
.....:

```

```

In [32]: dft["id"] = dft.index

```

```

In [33]: df

```

```

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
gux   one    1.431256  1.340309 -1.170299 -0.226169
      two    0.410835  0.813850  0.132003 -0.827317
[8 rows x 4 columns]

```

```

In [34]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")

```

```

      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
[6 rows x 3 columns]

```

14.4 Combining with stats and GroupBy

It should be no shock that combining `pivot` / `stack` / `unstack` with `GroupBy` and the basic `Series` and `DataFrame` statistical functions can produce some very expressive and fast data manipulations.

```

In [35]: df

```

```

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 4 columns]
```

```
In [36]: df.stack().mean(1).unstack()
```

```
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]
```

```
# same result, another way
```

```
In [37]: df.groupby(level=1, axis=1).mean()
```

```
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]: df.stack().groupby(level=1).mean()
```

```
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)
```

```
exp      A      B
animal
cat      0.311433 -0.431481
dog     -0.184544  0.133632
[2 rows x 2 columns]
```

14.5 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
```

```
   A  B  C      D      E
0  one A  foo -1.436737  0.149748
1  one B  foo -1.413681 -0.732339
2  two C  foo  1.607920  0.687738
3 three A  bar  1.024180  0.176444
4  one B  bar  0.569605  0.403310
5  one C  bar  0.875906 -0.154951
6  two A  foo -2.211372  0.301624
7 three B  foo  0.974466 -2.179861
8  one C  foo -2.006747 -1.369849
9  one A  bar -0.410001 -0.954208
10 two B  bar -0.078638  1.462696
11 three C  bar  0.545952 -1.743161
12  one A  foo -1.219217 -0.826591
13  one B  foo -1.226825 -0.345352
14  two C  foo  0.769804  1.314232
... ..
[24 rows x 5 columns]
```

We can produce pivot tables from this data very easily:

```
In [42]: pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
```

```
 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 -0.594487      NaN
```

```
C      NaN  1.188862
[9 rows x 2 columns]
```

```
In [43]: pivot_table(df, values='D', rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
```

```
A      one      three      two
C      bar      foo      bar      foo      bar      foo
B
A  0.549725 -2.655954 -0.257067      NaN      NaN -2.309255
B -0.158102 -2.640506      NaN  1.670241 -1.188974      NaN
C  0.754600 -1.665013 -0.074024      NaN      NaN  2.377724
[3 rows x 6 columns]
```

```
In [44]: pivot_table(df, values=['D', 'E'], rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
```

```
      D      E \
A      one      three      two      one
C      bar      foo      bar      foo      bar      foo      bar
B
A  0.549725 -2.655954 -0.257067      NaN      NaN -2.309255 -2.190477
B -0.158102 -2.640506      NaN  1.670241 -1.188974      NaN  1.399070
C  0.754600 -1.665013 -0.074024      NaN      NaN  2.377724  2.241830

A      three      two
C      foo      bar      foo      bar      foo
B
A -0.676843  0.867024      NaN      NaN  0.316495
B -1.077692      NaN  1.177566  2.358867      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'])
```

```
      D      E
C      bar      foo      bar      foo
A  B
one  A  0.274863 -1.327977 -1.095238 -0.338421
      B -0.079051 -1.320253  0.699535 -0.538846
      C  0.377300 -0.832506  1.120915 -0.843645
three A -0.128534      NaN  0.433512      NaN
      B      NaN  0.835120      NaN  0.588783
      C -0.037012      NaN -1.115381      NaN
two   A      NaN -1.154627      NaN  0.158248
      B -0.594487      NaN  1.179433      NaN
      C      NaN  1.188862      NaN  1.000985
[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=''))
```

```
      D      E
C      bar      foo      bar      foo
A  B
```

```

one   A   0.274863 -1.327977 -1.095238 -0.338421
      B  -0.079051 -1.320253  0.699535 -0.538846
      C   0.377300 -0.832506  1.120915 -0.843645
three A -0.128534          0.433512
      B          0.835120          0.588783
      C -0.037012          -1.115381
two   A          -1.154627          0.158248
      B -0.594487          1.179433
      C          1.188862          1.000985

```

Note that `pivot_table` is also available as an instance method on `DataFrame`.

14.5.1 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:

```
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'])
```

```

b      one      two
c  dull  shiny  dull  shiny
a
bar     1     0     0     1
foo     2     1     1     0
[2 rows x 4 columns]

```

14.5.2 Adding margins (partial aggregates)

If you pass `margins=True` to `pivot_table`, special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
In [53]: df.pivot_table(rows=['A', 'B'], cols='C', margins=True, aggfunc=np.std)
```

C		D			E			
		bar	foo	All	bar	foo	All	
A	one	A	0.968543	0.153810	1.084870	0.199447	0.690376	0.602542
		B	0.917338	0.132127	0.894343	0.418926	0.273641	0.771139
		C	0.705136	1.660627	1.254131	1.804346	0.744165	1.598848
three	A	1.630183	NaN	1.630183	0.363548	NaN	0.363548	
		B	NaN	0.197065	0.197065	NaN	3.915454	3.915454
		C	0.824435	NaN	0.824435	0.887815	NaN	0.887815
two	A	NaN	1.494463	1.494463	NaN	0.202765	0.202765	
		B	0.729521	NaN	0.729521	0.400594	NaN	0.400594
		C	NaN	0.592638	0.592638	NaN	0.442998	0.442998
All			0.816058	1.294620	1.055572	1.190502	1.403041	1.249705

[10 rows x 6 columns]

14.6 Tiling

The `cut` function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

```
In [54]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])
```

```
In [55]: cut(ages, bins=3)
```

```
(9.95, 26.667]
(9.95, 26.667]
(9.95, 26.667]
(9.95, 26.667]
(9.95, 26.667]
(9.95, 26.667]
(26.667, 43.333]
(43.333, 60]
(43.333, 60]
Levels (3): Index(['(9.95, 26.667]', '(26.667, 43.333]', '(43.333, 60]'], dtype=object)
```

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])
```

```
(0, 18]
(0, 18]
(0, 18]
(0, 18]
(18, 35]
(18, 35]
(18, 35]
(35, 70]
(35, 70]
Levels (3): Index(['(0, 18]', '(18, 35]', '(35, 70]'], dtype=object)
```

14.7 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'), 'data1': range(6)})
```

```
In [58]: get_dummies(df['key'])
```

```
   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
```

```
   key_a  key_b  key_c
0      0     1     0
1      0     1     0
2      1     0     0
3      0     0     1
4      1     0     0
5      0     1     0
[6 rows x 3 columns]
```

```
In [61]: df[['data1']].join(dummies)
```

```
   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
```

```
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))
```

```
   (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      1      0      0
3      1      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      1      0      0
9      1      0      0      0
[10 rows x 4 columns]
```

14.8 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
```

```
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
array([ 0,  0, -1,  1,  2,  3])
```

```
In [70]: uniques
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)
(array([ 2,  2, -1,  3,  0,  1]), array([3.14, inf, 'A', 'B'], dtype=object))
```

```
In [72]: np.unique(x, return_inverse=True)[::-1]
(array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
```


TIME SERIES / DATE FUNCTIONALITY

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:

```
# 72 hours starting with midnight Jan 1st, 2011
In [1]: rng = date_range('1/1/2011', periods=72, freq='H')

In [2]: rng[:5]

<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:

```
In [3]: ts = Series(randn(len(rng)), index=rng)

In [4]: ts.head()

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:

```
# to 45 minute frequency and forward fill
In [5]: converted = ts.asfreq('45Min', method='pad')
```

```
In [6]: converted.head()

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')

2011-01-01    -0.319569
2011-01-02    -0.337703
2011-01-03     0.117258
Freq: D, dtype: float64
```

15.1 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)
pandas.tseries.index.DatetimeIndex

In [11]: ts

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)
pandas.tseries.period.PeriodIndex

In [15]: ts

2012-01    -1.219217
2012-02    -1.226825
2012-03     0.769804
Freq: M, dtype: float64
```

Starting with 0.8, pandas allows you to capture both representations and convert between them. Under the hood, pandas represents timestamps using instances of `Timestamp` and sequences of timestamps using instances of `DatetimeIndex`. For regular time spans, pandas uses `Period` objects for scalar values and `PeriodIndex` for sequences of spans. Better support for irregular intervals with arbitrary start and end points are forth-coming in future releases.

15.2 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`:

```
In [16]: to_datetime(Series(['Jul 31, 2009', '2010-01-10', None]))

0    2009-07-31 00:00:00
1    2010-01-10 00:00:00
2                NaT
dtype: datetime64[ns]
```

```
In [17]: to_datetime(['2005/11/23', '2010.12.31'])
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2005-11-23 00:00:00, 2010-12-31 00:00:00]
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:

```
In [18]: to_datetime(['04-01-2012 10:00'], dayfirst=True)
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-04 10:00:00]
Length: 1, Freq: None, Timezone: None
```

```
In [19]: to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-14 00:00:00, 2012-01-14 00:00:00]
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`.

15.2.1 Invalid Data

Pass `coerce=True` to convert invalid data to `NaT` (not a time):

```
In [20]: to_datetime(['2009-07-31', 'asd'])
array(['2009-07-31', 'asd'], dtype=object)
```

```
In [21]: to_datetime(['2009-07-31', 'asd'], coerce=True)
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2009-07-31 00:00:00, NaT]
Length: 2, Freq: None, Timezone: None
```

Take care, `to_datetime` may not act as you expect on mixed data:

```
In [22]: to_datetime([1, '1'])
array([1, '1'], dtype=object)
```

15.2.2 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

```
In [23]: to_datetime([1349720105, 1349806505, 1349892905,
.....:               1349979305, 1350065705], unit='s')
.....:

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-10-08 18:15:05, ..., 2012-10-12 18:15:05]
Length: 5, Freq: None, Timezone: None

In [24]: to_datetime([1349720105100, 1349720105200, 1349720105300,
.....:               1349720105400, 1349720105500 ], unit='ms')
.....:

<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.

```
In [25]: to_datetime([1])

<class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:00.000000001]
Length: 1, Freq: None, Timezone: None

In [26]: to_datetime([1, 3.14], unit='s')

<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.

15.3 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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01 00:00:00, ..., 2012-05-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

```
In [30]: index = Index(dates)
```

```
In [31]: index # Automatically converted to DatetimeIndex
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01 00:00:00, ..., 2012-05-03 00:00:00]
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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31 00:00:00, ..., 2083-04-30 00:00:00]
Length: 1000, Freq: M, Timezone: None
```

```
In [34]: index = bdate_range('2012-1-1', periods=250)
```

```
In [35]: index
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-02 00:00:00, ..., 2012-12-14 00:00:00]
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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2012-01-01 00:00:00]
Length: 366, Freq: D, Timezone: None
```

```
In [40]: rng = bdate_range(start, end)
```

```
In [41]: rng
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-12-30 00:00:00]
Length: 260, Freq: B, Timezone: None
```

`date_range` and `bdate_range` makes it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`:

```
In [42]: date_range(start, end, freq='BM')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None
```

```
In [43]: date_range(start, end, freq='W')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-02 00:00:00, ..., 2012-01-01 00:00:00]
Length: 53, Freq: W-SUN, Timezone: None
```

```
In [44]: bdate_range(end=end, periods=20)

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-12-05 00:00:00, ..., 2011-12-30 00:00:00]
Length: 20, Freq: B, Timezone: None
```

```
In [45]: bdate_range(start=start, periods=20)

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-28 00:00:00]
Length: 20, Freq: B, Timezone: None
```

The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

15.4 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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None
```

```
In [49]: ts[:5].index
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-05-31 00:00:00]
Length: 5, Freq: BM, Timezone: None
```

```
In [50]: ts[::2].index
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-11-30 00:00:00]
Length: 6, Freq: 2BM, Timezone: None
```

15.4.1 Partial String Indexing

You can pass in dates and strings that parse to dates as indexing parameters:

```
In [51]: ts['1/31/2011']
-1.2812473076599531
```

```
In [52]: ts[datetime(2011, 12, 25):]
```

```
2011-12-30    0.687738
Freq: BM, dtype: float64
```

```
In [53]: ts['10/31/2011':'12/31/2011']
```

```
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']
```

```
2011-01-31   -1.281247
2011-02-28   -0.727707
2011-03-31   -0.121306
2011-04-29   -0.097883
2011-05-31    0.695775
2011-06-30    0.341734
2011-07-29    0.959726
2011-08-31   -1.110336
2011-09-30   -0.619976
2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
```

```
Freq: BM, dtype: float64
```

```
In [55]: ts['2011-6']
```

```
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='BMS',
```

```
In [57]: dft
```

```

              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']
```

```

              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']
```

```

      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']
```

```

      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']
```

```

      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
...
[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']
```

```

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']
```

```
A    0.193284
Name: 2013-01-15 12:30:00, dtype: float64
```

15.4.2 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)]
```

```

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
...
[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)]

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]

```

15.4.3 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')

2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-31    0.687738
Freq: BM, dtype: float64

```

Even complicated fancy indexing that breaks the DatetimeIndex's frequency regularity will result in a DatetimeIndex (but frequency is lost):

```

In [67]: ts[[0, 2, 6]].index

<class 'pandas.tseries.index.DatetimeIndex'>

```

```
[2011-01-31 00:00:00, ..., 2011-07-29 00:00:00]  
Length: 3, Freq: None, Timezone: None
```

15.5 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.

Class name	Description
<code>DateOffset</code>	Generic offset class, defaults to 1 calendar day
<code>BDay</code>	business day (weekday)
<code>CDay</code>	custom business day (experimental)
<code>Week</code>	one week, optionally anchored on a day of the week
<code>WeekOfMonth</code>	the x-th day of the y-th week of each month
<code>LastWeekOfMonth</code>	the x-th day of the last week of each month
<code>MonthEnd</code>	calendar month end
<code>MonthBegin</code>	calendar month begin
<code>BMonthEnd</code>	business month end
<code>BMonthBegin</code>	business month begin
<code>QuarterEnd</code>	calendar quarter end
<code>QuarterBegin</code>	calendar quarter begin
<code>BQuarterEnd</code>	business quarter end
<code>BQuarterBegin</code>	business quarter begin
<code>FY5253Quarter</code>	retail (aka 52-53 week) quarter
<code>YearEnd</code>	calendar year end
<code>YearBegin</code>	calendar year begin
<code>BYearEnd</code>	business year end
<code>BYearBegin</code>	business year begin
<code>FY5253</code>	retail (aka 52-53 week) year
<code>Hour</code>	one hour
<code>Minute</code>	one minute
<code>Second</code>	one second
<code>Milli</code>	one millisecond
<code>Micro</code>	one microsecond

The basic `DateOffset` takes the same arguments as `dateutil.relativedelta`, which works like:

```
In [68]: d = datetime(2008, 8, 18)  
  
In [69]: d + relativedelta(months=4, days=5)  
datetime.datetime(2008, 12, 23, 0, 0)
```

We could have done the same thing with `DateOffset`:

```
In [70]: from pandas.tseries.offsets import *  
  
In [71]: d + DateOffset(months=4, days=5)  
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:

```
class BDay(DateOffset):
    """DateOffset increments between business days"""
    def apply(self, other):
        ...
```

```
In [72]: d - 5 * BDay()
Timestamp('2008-08-11 00:00:00', tz=None)
```

```
In [73]: d + BMonthEnd()
Timestamp('2008-08-29 00:00:00', tz=None)
```

The `rollforward` and `rollback` methods do exactly what you would expect:

```
In [74]: d
datetime.datetime(2008, 8, 18, 0, 0)
```

```
In [75]: offset = BMonthEnd()
```

```
In [76]: offset.rollforward(d)
Timestamp('2008-08-29 00:00:00', tz=None)
```

```
In [77]: offset.rollback(d)
datetime.datetime(2008, 7, 31, 0, 0)
```

It’s definitely worth exploring the `pandas.tseries.offsets` module and the various docstrings for the classes.

15.5.1 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()
datetime.datetime(2008, 8, 25, 0, 0)
```

```
In [79]: d + Week(weekday=4)
Timestamp('2008-08-22 00:00:00', tz=None)
```

```
In [80]: (d + Week(weekday=4)).weekday()
4
```

Another example is parameterizing `YearEnd` with the specific ending month:

```
In [81]: d + YearEnd()
Timestamp('2008-12-31 00:00:00', tz=None)
```

```
In [82]: d + YearEnd(month=6)
Timestamp('2009-06-30 00:00:00', tz=None)
```

15.5.2 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.

```
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 00:00:00
2013-05-02    2013-05-02 00:00:00
2013-05-05    2013-05-05 00:00:00
2013-05-06    2013-05-06 00:00:00
2013-05-07    2013-05-07 00:00:00
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.

15.5.3 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).

Alias	Description
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
BM	business month end frequency
MS	month start frequency
BMS	business month start frequency
Q	quarter end frequency
BQ	business quarter end frequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
H	hourly frequency
T	minutely frequency
S	secondly frequency
L	milliseconds
U	microseconds

15.5.4 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```
In [92]: date_range(start, periods=5, freq='B')
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None
```

```
In [93]: date_range(start, periods=5, freq=BDay())
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None
```

You can combine together day and intraday offsets:

```
In [94]: date_range(start, periods=10, freq='2h20min')
```

```
<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')
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
```

```
[2011-01-01 00:00:00, ..., 2011-01-10 00:00:00.000090]  
Length: 10, Freq: 86400000010U, Timezone: None
```

15.5.5 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

Alias	Description
W-SUN	weekly frequency (sundays). Same as 'W'
W-MON	weekly frequency (mondays)
W-TUE	weekly frequency (tuesdays)
W-WED	weekly frequency (wednesdays)
W-THU	weekly frequency (thursdays)
W-FRI	weekly frequency (fridays)
W-SAT	weekly frequency (saturdays)
(B)Q(S)-DEC	quarterly frequency, year ends in December. Same as 'Q'
(B)Q(S)-JAN	quarterly frequency, year ends in January
(B)Q(S)-FEB	quarterly frequency, year ends in February
(B)Q(S)-MAR	quarterly frequency, year ends in March
(B)Q(S)-APR	quarterly frequency, year ends in April
(B)Q(S)-MAY	quarterly frequency, year ends in May
(B)Q(S)-JUN	quarterly frequency, year ends in June
(B)Q(S)-JUL	quarterly frequency, year ends in July
(B)Q(S)-AUG	quarterly frequency, year ends in August
(B)Q(S)-SEP	quarterly frequency, year ends in September
(B)Q(S)-OCT	quarterly frequency, year ends in October
(B)Q(S)-NOV	quarterly frequency, year ends in November
(B)A(S)-DEC	annual frequency, anchored end of December. Same as 'A'
(B)A(S)-JAN	annual frequency, anchored end of January
(B)A(S)-FEB	annual frequency, anchored end of February
(B)A(S)-MAR	annual frequency, anchored end of March
(B)A(S)-APR	annual frequency, anchored end of April
(B)A(S)-MAY	annual frequency, anchored end of May
(B)A(S)-JUN	annual frequency, anchored end of June
(B)A(S)-JUL	annual frequency, anchored end of July
(B)A(S)-AUG	annual frequency, anchored end of August
(B)A(S)-SEP	annual frequency, anchored end of September
(B)A(S)-OCT	annual frequency, anchored end of October
(B)A(S)-NOV	annual frequency, anchored end of November

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

15.5.6 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.

Legacy Time Rule	Offset Alias
WEEKDAY	B
EOM	BM
W@MON	W-MON
W@TUE	W-TUE
W@WED	W-WED
W@THU	W-THU
W@FRI	W-FRI
W@SAT	W-SAT
W@SUN	W-SUN
Q@JAN	BQ-JAN
Q@FEB	BQ-FEB
Q@MAR	BQ-MAR
A@JAN	BA-JAN
A@FEB	BA-FEB
A@MAR	BA-MAR
A@APR	BA-APR
A@MAY	BA-MAY
A@JUN	BA-JUN
A@JUL	BA-JUL
A@AUG	BA-AUG
A@SEP	BA-SEP
A@OCT	BA-OCT
A@NOV	BA-NOV
A@DEC	BA-DEC
min	T
ms	L
us	U

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.

15.6 Time series-related instance methods

15.6.1 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)
```

```
2011-01-31      NaN
2011-02-28    -1.281247
2011-03-31    -0.727707
2011-04-29    -0.121306
2011-05-31    -0.097883
Freq: BM, dtype: float64
```

The `shift` method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object

or also a *offset alias*:

```
In [98]: ts.shift(5, freq=datetools.bday)
```

```
2011-02-07    -1.281247
2011-03-07    -0.727707
2011-04-07    -0.121306
2011-05-06    -0.097883
2011-06-07     0.695775
dtype: float64
```

```
In [99]: ts.shift(5, freq='BM')
```

```
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:

```
In [100]: ts.tshift(5, freq='D')
```

```
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.

15.6.2 Frequency conversion

The primary function for changing frequencies is the `asfreq` function. For a `DatetimeIndex`, this is basically just a thin, but convenient wrapper around `reindex` which generates a `date_range` and calls `reindex`.

```
In [101]: dr = date_range('1/1/2010', periods=3, freq=3 * datetools.bday)
```

```
In [102]: ts = Series(randn(3), index=dr)
```

```
In [103]: ts
```

```
2010-01-01    -0.659574
2010-01-06     1.494522
2010-01-11    -0.778425
Freq: 3B, dtype: float64
```

```
In [104]: ts.asfreq(BDay())
```

```
2010-01-01    -0.659574
2010-01-04         NaN
2010-01-05         NaN
2010-01-06     1.494522
2010-01-07         NaN
2010-01-08         NaN
```

```
2010-01-11    -0.778425
Freq: B, dtype: float64
```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion

```
In [105]: ts.asfreq(BDay(), method='pad')
```

```
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
```

15.6.3 Filling forward / backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the [missing data section](#).

15.6.4 Converting to Python datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.

15.7 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')
```

```
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') # default is mean
```

```
2012-01-01    251.03
Freq: 5T, dtype: float64
```

```
In [110]: ts.resample('5Min', how='ohlc')
```

```
      open  high  low  close
2012-01-01   308   460    9   205
[1 rows x 4 columns]
```

```
In [111]: ts.resample('5Min', how=np.max)
```

```
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')
```

```
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')
```

```
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')
```

```
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')
```

```
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)
```

```
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    NaN
2012-01-01 00:00:01    204
Freq: 250L, dtype: float64
```

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'
```

```
2012-01-01    251.03
Freq: 5T, dtype: float64
```

```
In [118]: ts.resample('5Min', label='left')
```

```
2012-01-01    251.03
Freq: 5T, dtype: float64
```

```
In [119]: ts.resample('5Min', label='left', loffset='1s')
```

```
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.

15.8 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`.

15.8.1 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')
Period('2012', 'A-DEC')
```

```
In [121]: Period('2012-1-1', freq='D')
Period('2012-01-01', 'D')
```

```
In [122]: Period('2012-1-1 19:00', freq='H')
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
Period('2013', 'A-DEC')
```

```
In [125]: p - 3
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')
10
```

15.8.2 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
```

```
<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')
```

```
<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)
```

```
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
```

15.8.3 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:

```
In [131]: p = Period('2011', freq='A-DEC')
```

```
In [132]: p
Period('2011', 'A-DEC')
```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```
In [133]: p.asfreq('M', how='start')
Period('2011-01', 'M')
```

```
In [134]: p.asfreq('M', how='end')
Period('2011-12', 'M')
```

The shorthands `'s'` and `'e'` are provided for convenience:

```
In [135]: p.asfreq('M', 's')
Period('2011-01', 'M')
```

```
In [136]: p.asfreq('M', 'e')
Period('2011-12', 'M')
```

Converting to a “super-period” (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```
In [137]: p = Period('2011-12', freq='M')
```

```
In [138]: p.asfreq('A-NOV')
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:

```
In [139]: p = Period('2012Q1', freq='Q-DEC')
```

```
In [140]: p.asfreq('D', 's')
Period('2012-01-01', 'D')
```

```
In [141]: p.asfreq('D', 'e')
Period('2012-03-31', 'D')
```

Q-MAR defines fiscal year end in March:

```
In [142]: p = Period('2011Q4', freq='Q-MAR')
```

```
In [143]: p.asfreq('D', 's')
Period('2011-01-01', 'D')
```

```
In [144]: p.asfreq('D', 'e')
Period('2011-03-31', 'D')
```

15.9 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
```

```
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
```

```
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()
```

```
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')
```

```
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
```



```
In [155]: ts.head()

1990-03-01 09:00    -0.089851
1990-06-01 09:00     0.711329
1990-09-01 09:00     0.531761
1990-12-01 09:00     0.265615
1991-03-01 09:00    -0.174462
Freq: H, dtype: float64
```

15.10 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
```

```
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')
```

```
2012-03-05 19:00:00-05:00    -2.189293
2012-03-06 19:00:00-05:00    -1.819506
2012-03-07 19:00:00-05:00     0.229798
2012-03-08 19:00:00-05:00     0.119425
2012-03-09 19:00:00-05:00     1.808966
2012-03-10 19:00:00-05:00     1.015841
2012-03-11 20:00:00-04:00    -1.651784
2012-03-12 20:00:00-04:00     0.347674
2012-03-13 20:00:00-04:00    -0.773688
2012-03-14 20:00:00-04:00     0.425863
2012-03-15 20:00:00-04:00     0.579486
2012-03-16 20:00:00-04:00    -0.745396
2012-03-17 20:00:00-04:00     0.141880
2012-03-18 20:00:00-04:00    -1.077754
2012-03-19 20:00:00-04:00    -1.301174
Freq: D, dtype: float64
```

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]
Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [167]: rng_berlin[5]
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

In [168]: rng_eastern[5] == rng_berlin[5]
True
```

Like `Series`, `DataFrame`, and `DatetimeIndex`, `Timestamps` can be converted to other time zones using `tz_convert`:

```
In [169]: rng_eastern[5]
Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [170]: rng_berlin[5]
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

In [171]: rng_eastern[5].tz_convert('Europe/Berlin')
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of `Timestamps` functions just like `DatetimeIndex` and `TimeSeries`:

```
In [172]: rng[5]
Timestamp('2012-03-11 00:00:00', tz=None)

In [173]: rng[5].tz_localize('Asia/Shanghai')
Timestamp('2012-03-11 00:00:00+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
```

```
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
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-06 00:00:00, ..., 2012-03-20 00:00:00]
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 01:00', '11/06/2011 02:00',
.....:                               '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/docbuild/CI/pandas/pandas/tseries/index.pyc in tz_localize(self, tz, infer_dst)
    1602
    1603         # Convert to UTC
-> 1604         new_dates = tslib.tz_localize_to_utc(self.asi8, tz, infer_dst=infer_dst)
    1605         new_dates = new_dates.view(_NS_DTYPE)
    1606
/home/docbuild/CI/pandas/pandas/tslib.so in pandas.tslib.tz_localize_to_utc (pandas/tslib.c:28987)()
AmbiguousTimeError: Cannot infer dst time from Timestamp('2011-11-06 01:00:00', tz=None), try using t
```

```
In [181]: rng_hourly_eastern = rng_hourly.tz_localize('US/Eastern', infer_dst=True)
```

```
In [182]: rng_hourly_eastern.values
```

```
array(['2011-11-06T06:00:00.000000000+0200',
      '2011-11-06T07:00:00.000000000+0200',
      '2011-11-06T08:00:00.000000000+0200',
      '2011-11-06T09:00:00.000000000+0200',
```

```
'2011-11-06T10:00:00.000000000+0200'], dtype='datetime64[ns]')
```

15.11 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
```

```
      A      B
0 2012-01-01 00:00:00      00:00:00
1 2012-01-02 00:00:00 1 days, 00:00:00
2 2012-01-03 00:00:00 2 days, 00:00:00
[3 rows x 2 columns]
```

```
In [188]: df['C'] = df['A'] + df['B']
```

```
In [189]: df
```

```
      A      B      C
0 2012-01-01 00:00:00      00:00:00 2012-01-01 00:00:00
1 2012-01-02 00:00:00 1 days, 00:00:00 2012-01-03 00:00:00
2 2012-01-03 00:00:00 2 days, 00:00:00 2012-01-05 00:00:00
[3 rows x 3 columns]
```

```
In [190]: df.dtypes
```

```
A      datetime64[ns]
B      timedelta64[ns]
C      datetime64[ns]
dtype: object
```

```
In [191]: s - s.max()
```

```
0    -2 days, 00:00:00
1    -1 days, 00:00:00
2           00:00:00
dtype: timedelta64[ns]
```

```
In [192]: s - datetime(2011, 1, 1, 3, 5)
```

```
0    364 days, 20:55:00
1    365 days, 20:55:00
2    366 days, 20:55:00
dtype: timedelta64[ns]
```

```
In [193]: s + timedelta(minutes=5)
```

```
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]
```

```
In [194]: s + Minute(5)
```

```
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]
```

```
In [195]: s + Minute(5) + Milli(5)
```

```
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
```

```
0           00:00:00
1    1 days, 00:00:00
2    2 days, 00:00:00
dtype: timedelta64[ns]
```

Series of `timedeltas` with `NaT` values are supported

```
In [198]: y = s - s.shift()
```

```
In [199]: y
```

```
0           NaT
1    1 days, 00:00:00
2    1 days, 00:00:00
dtype: timedelta64[ns]
```

Elements can be set to `NaT` using `np.nan` analogously to datetimes

```
In [200]: y[1] = np.nan
```

```
In [201]: y
```

```
0           NaT
1           NaT
2    1 days, 00:00:00
dtype: timedelta64[ns]
```

Operands can also appear in a reversed order (a singular object operated with a Series)

```
In [202]: s.max() - s
```

```
0    2 days, 00:00:00
1    1 days, 00:00:00
2           00:00:00
```

```
dtype: timedelta64[ns]
```

```
In [203]: datetime(2011,1,1,3,5) - s
```

```
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
```

```
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)
```

```
0          -00:05:05.000005
1           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
```

```
           A           B
0    -00:05:05 -1 days, 00:00:00
1     23:54:55 -1 days, 00:00:00
2  1 days, 23:54:55 -1 days, 00:00:00
[3 rows x 2 columns]
```

```
In [210]: df.min()
```

```
A          -00:05:05
B    -1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [211]: df.min(axis=1)
```

```
0    -1 days, 00:00:00
1    -1 days, 00:00:00
2    -1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [212]: df.idxmin()
```

```
A    0
B    0
dtype: int64
```

```
In [213]: df.idxmax()
```

```
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()
```

```
0      -00:05:05
dtype: timedelta64[ns]
```

```
In [215]: df.min(axis=1).min()
```

```
0      -1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [216]: df.min().idxmax()
```

```
'A'
```

```
In [217]: df.min(axis=1).idxmin()
```

```
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)
```

```
0           00:00:00
1           00:00:00
2      1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [219]: y.fillna(10)
```

```
0           00:00:10
1           00:00:10
2      1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [220]: y.fillna(timedelta(days=-1,seconds=5))
```

```
0           -23:59:55
1           -23:59:55
2      1 days, 00:00:00
dtype: timedelta64[ns]
```

15.12 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
```

```
0          -23:59:55
1          -23:59:55
2    1 days, 00:00:00
dtype: timedelta64[ns]
```

```
In [223]: y2.mean()
```

```
0    -07:59:56.666667
dtype: timedelta64[ns]
```

```
In [224]: y2.quantile(.1)
numpy.timedelta64(-863950000000000,'ns')
```

15.13 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')
numpy.timedelta64(108301000030000,'ns')
```

```
In [226]: to_timedelta('15.5us')
numpy.timedelta64(15500,'ns')
```

```
In [227]: to_timedelta(['1 days 06:05:01.00003','15.5us','nan'])
```

```
0    1 days, 06:05:01.000030
1           00:00:00.000016
2                               NaT
dtype: timedelta64[ns]
```

```
In [228]: to_timedelta(np.arange(5),unit='s')
```

```
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')
```

```
0           00:00:00
1    1 days, 00:00:00
2    2 days, 00:00:00
3    3 days, 00:00:00
```



```
4    4 days, 00:00:00
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 dtypes 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
```

```
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')
```

```
0    31.000000
1    31.000000
2    31.003507
3         NaN
dtype: float64
```

```
In [235]: td.astype('timedelta64[D]')
```

```
0    31
1    31
2    31
3    NaN
dtype: float64
```

```
# to seconds
```

```
In [236]: td / np.timedelta64(1, 's')
```

```
0    2678400
1    2678400
2    2678703
3         NaN
dtype: float64
```

```
In [237]: td.astype('timedelta64[s]')
```

```
0    2678400
1    2678400
2    2678703
3         NaN
dtype: float64
```

Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series yields another timedelta64[ns] dtypes Series.

```
In [238]: td * -1
```

```
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])
```

```
0    31 days, 00:00:00
1    62 days, 00:00:00
2    93 days, 00:15:09
3                NaT
dtype: timedelta64[ns]
```

15.13.1 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.timetimedelta64` type only has 1 argument, the number of **micro** seconds.

The following are equivalent statements in the two versions of numpy.

```
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')
```

PLOTTING WITH MATPLOTLIB

Note: We intend to build more plotting integration with `matplotlib` as time goes on.

We use the standard convention for referencing the matplotlib API:

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

16.1 Basic plotting: `plot`

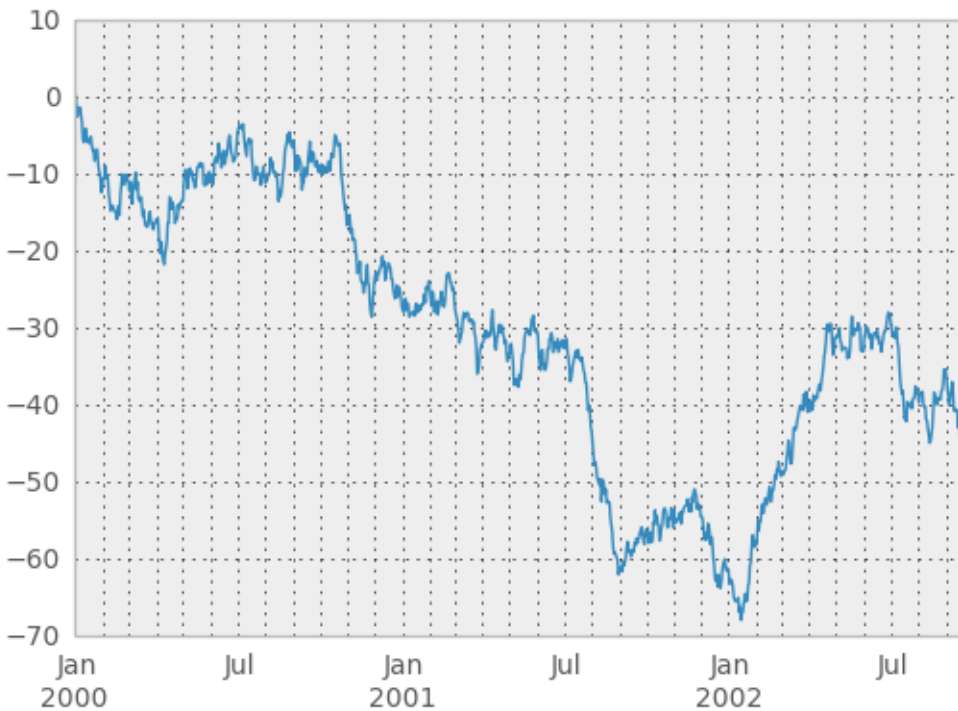
See the *cookbook* for some advanced strategies

The `plot` method on `Series` and `DataFrame` is just a simple wrapper around `plt.plot`:

```
In [2]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))
```

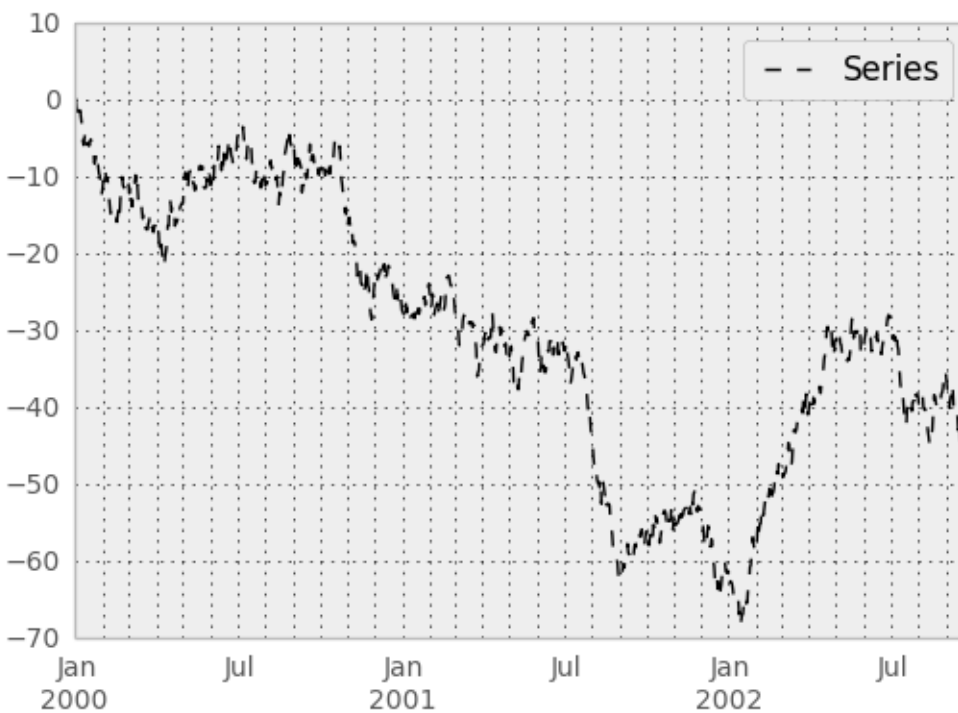
```
In [3]: ts = ts.cumsum()
```

```
In [4]: ts.plot()  
<matplotlib.axes.AxesSubplot at 0x14bade10>
```



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()  
<matplotlib.legend.Legend at 0x149c3c10>
```

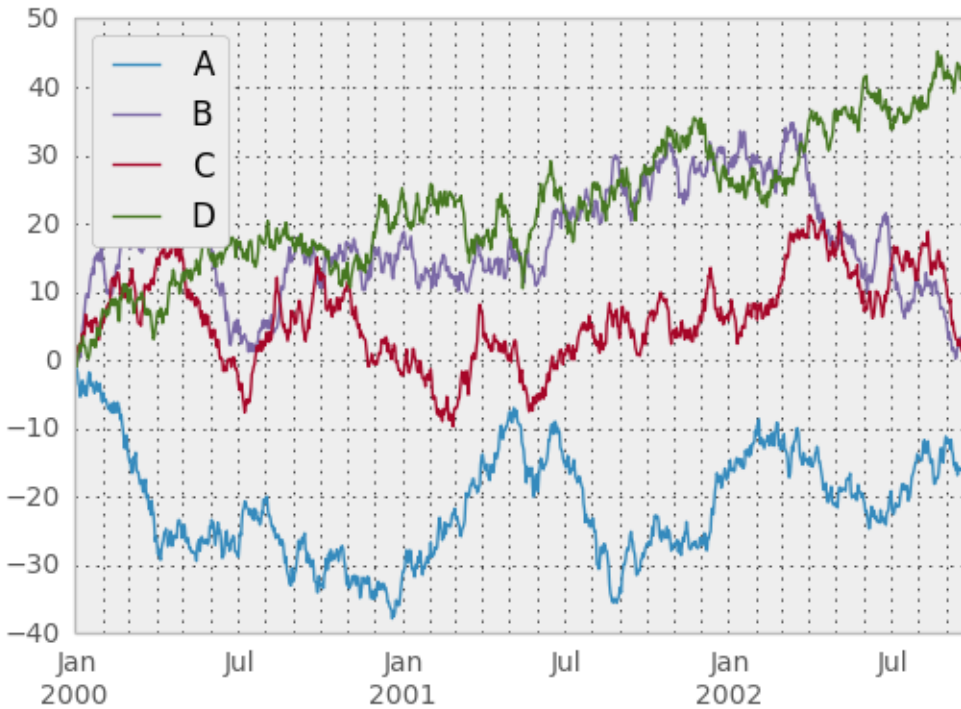


On `DataFrame`, `plot` is a convenience to plot all of the columns with labels:

```
In [6]: df = DataFrame(randn(1000, 4), index=ts.index, columns=list('ABCD'))
```

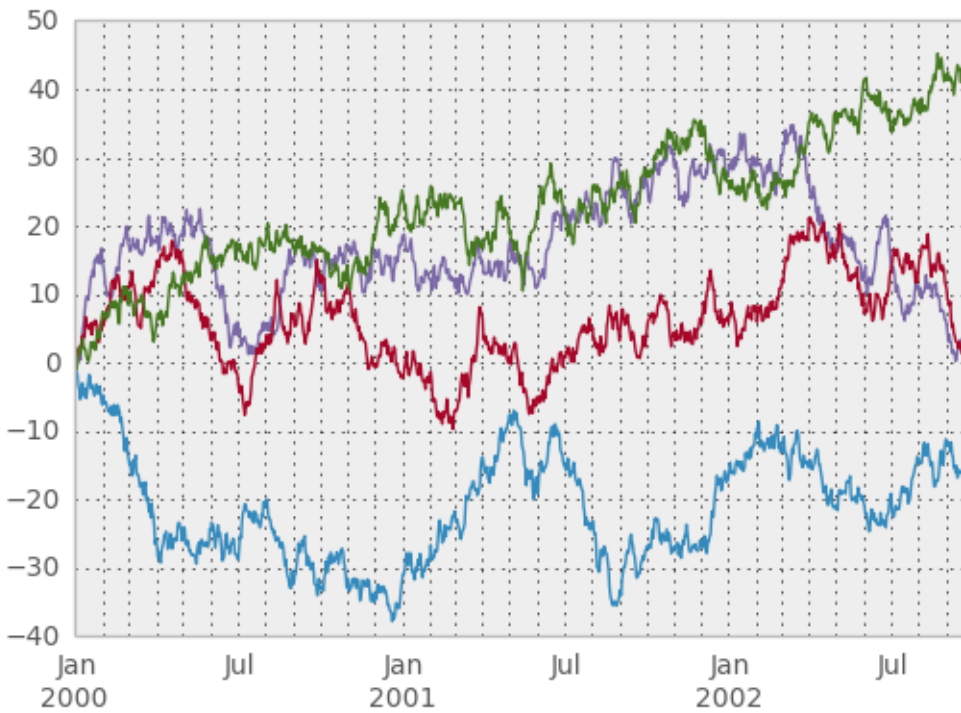
```
In [7]: df = df.cumsum()
```

```
In [8]: plt.figure(); df.plot(); plt.legend(loc='best')
<matplotlib.legend.Legend at 0x155843d0>
```



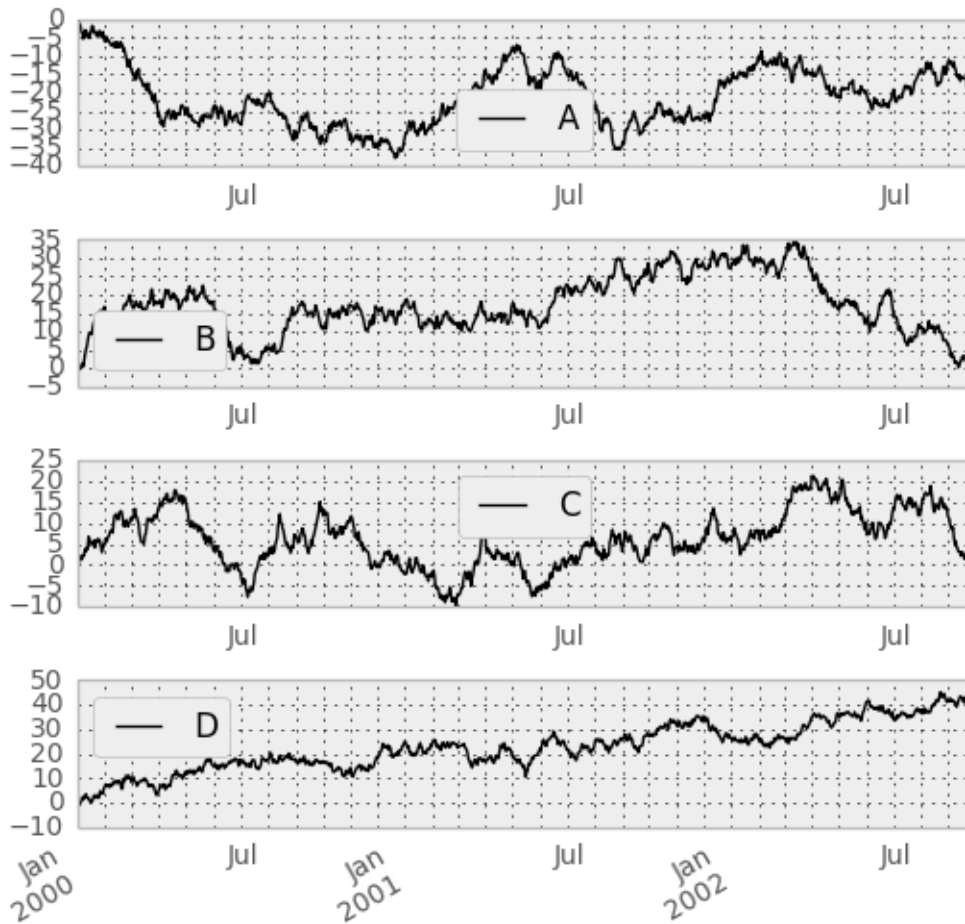
You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```
In [9]: df.plot(legend=False)
<matplotlib.axes.AxesSubplot at 0x134cbf50>
```



Some other options are available, like plotting each Series on a different axis:

```
In [10]: df.plot(subplots=True, figsize=(6, 6)); plt.legend(loc='best')
<matplotlib.legend.Legend at 0x74a6310>
```



You may pass `logy` to get a log-scale Y axis.

```
In [11]: plt.figure();
In [11]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [12]: ts = np.exp(ts.cumsum())

In [13]: ts.plot(logy=True)
<matplotlib.axes.AxesSubplot at 0x14bc3f90>
```



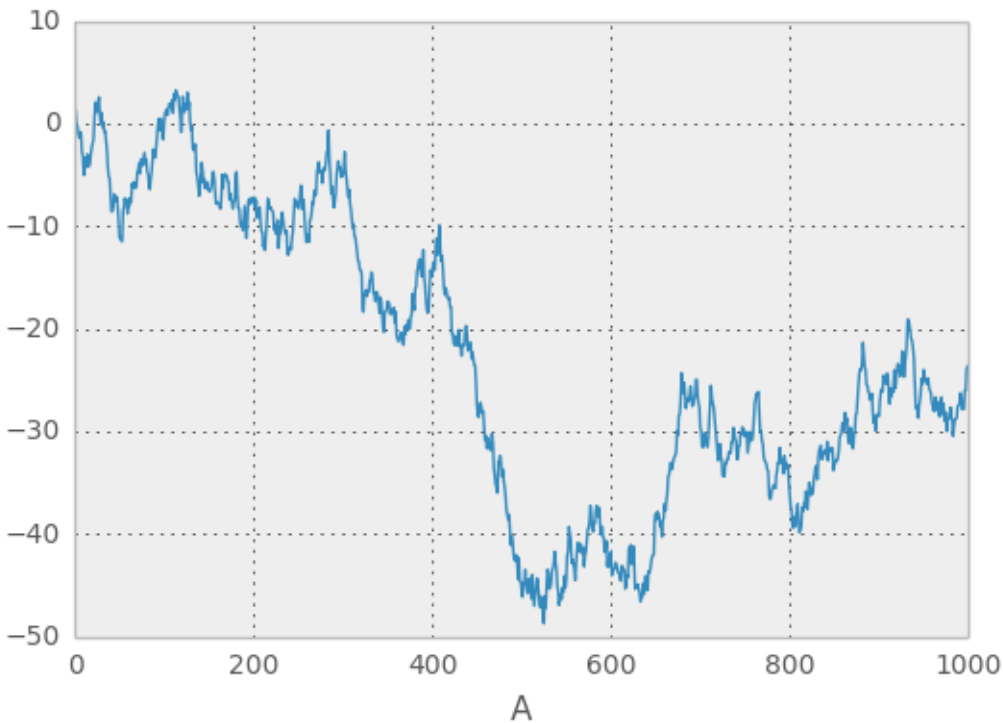
You can plot one column versus another using the *x* and *y* keywords in *DataFrame.plot*:

```
In [14]: plt.figure()
<matplotlib.figure.Figure at 0x1432ad50>

In [15]: df3 = DataFrame(randn(1000, 2), columns=['B', 'C']).cumsum()

In [16]: df3['A'] = Series(list(range(len(df))))

In [17]: df3.plot(x='A', y='B')
<matplotlib.axes.AxesSubplot at 0x14320090>
```

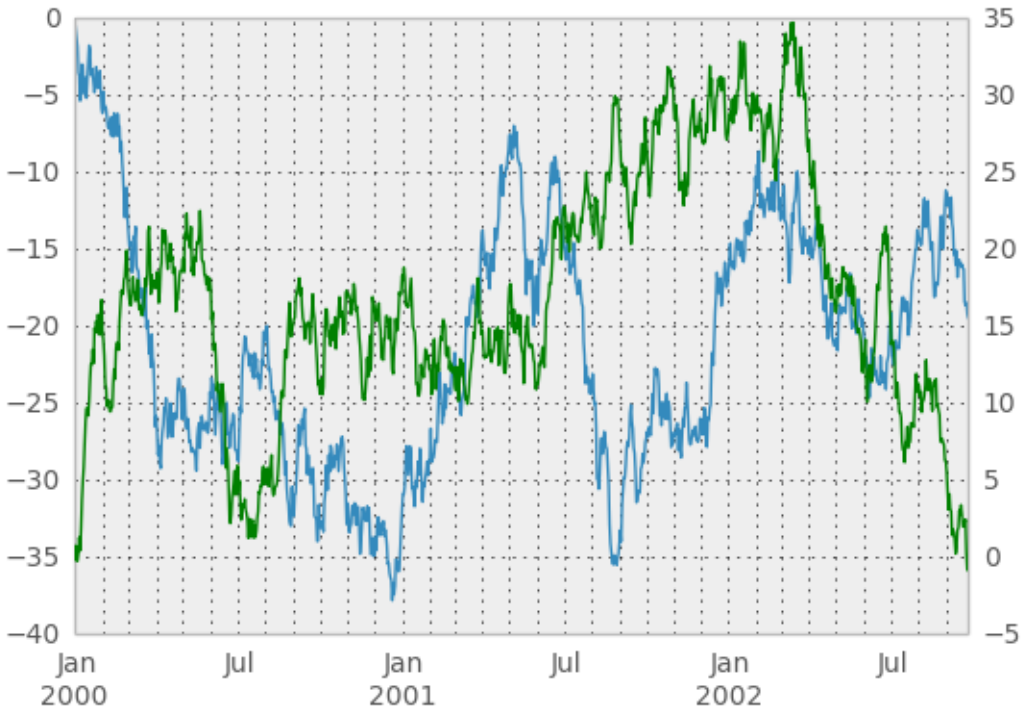
16.1.1 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```
In [18]: plt.figure()
<matplotlib.figure.Figure at 0xb76b610>

In [19]: df.A.plot()
<matplotlib.axes.AxesSubplot at 0xe14ee90>

In [20]: df.B.plot(secondary_y=True, style='g')
<matplotlib.axes.AxesSubplot at 0x1500f610>
```



16.1.2 Selective Plotting on Secondary Y-axis

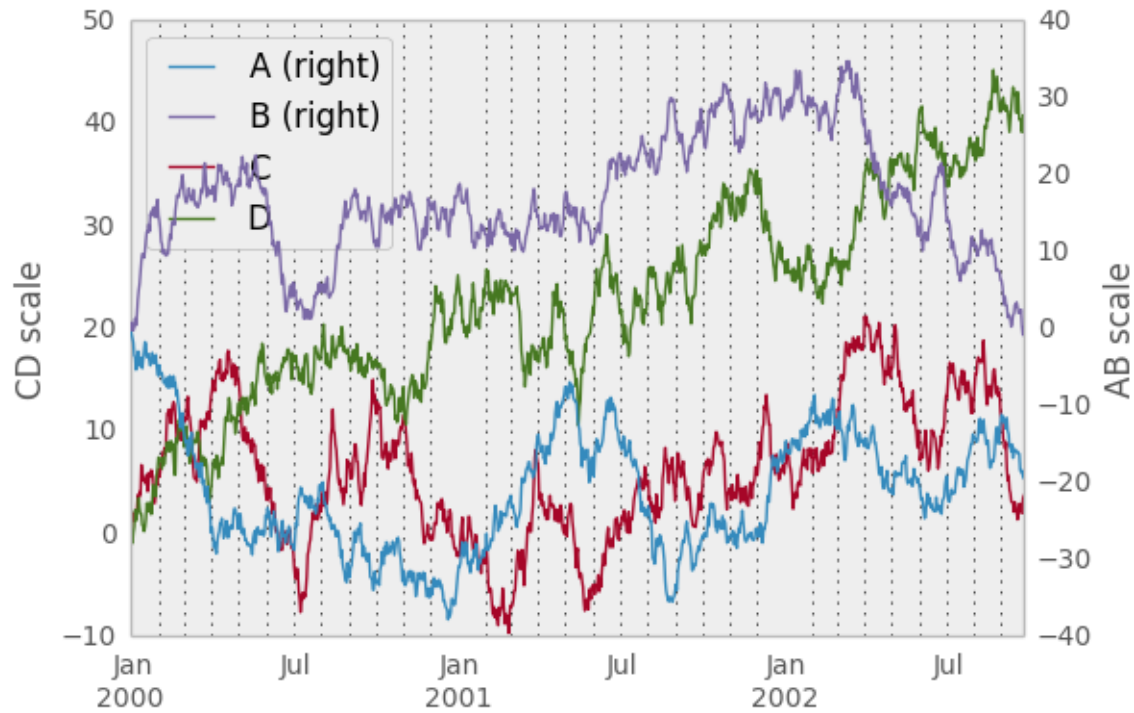
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```
In [21]: plt.figure()
<matplotlib.figure.Figure at 0x1500fa50>

In [22]: ax = df.plot(secondary_y=['A', 'B'])

In [23]: ax.set_ylabel('CD scale')
<matplotlib.text.Text at 0x15191f90>

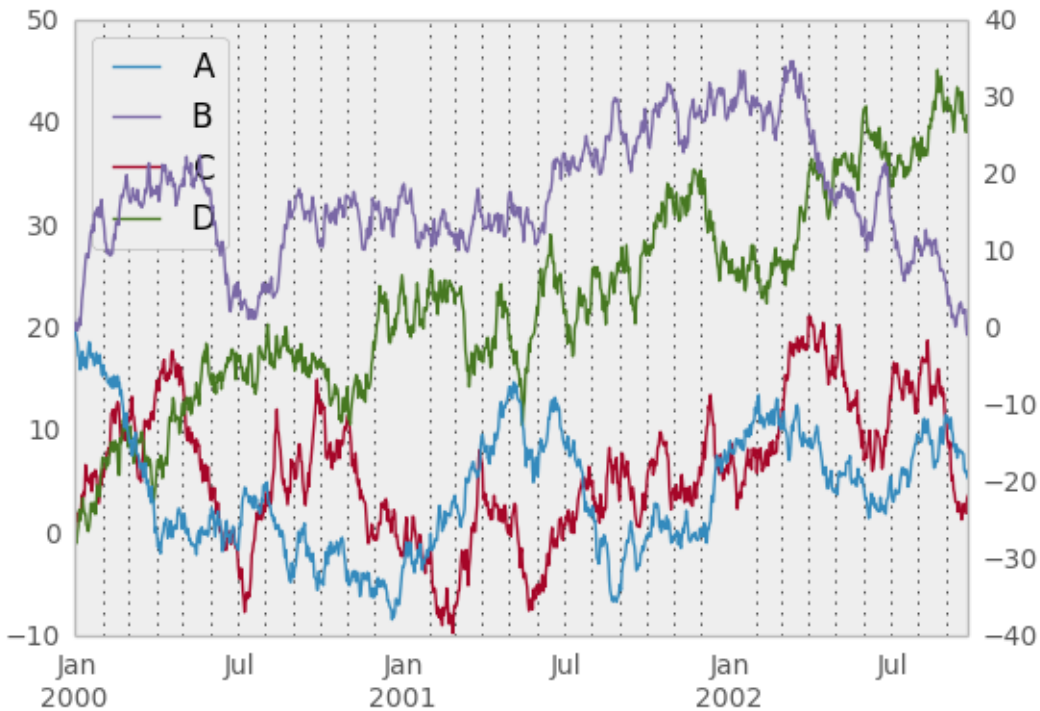
In [24]: ax.right_ax.set_ylabel('AB scale')
<matplotlib.text.Text at 0x15191590>
```



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 [25]: plt.figure()
<matplotlib.figure.Figure at 0x1500f690>

In [26]: df.plot(secondary_y=['A', 'B'], mark_right=False)
<matplotlib.axes.AxesSubplot at 0x153b77d0>
```



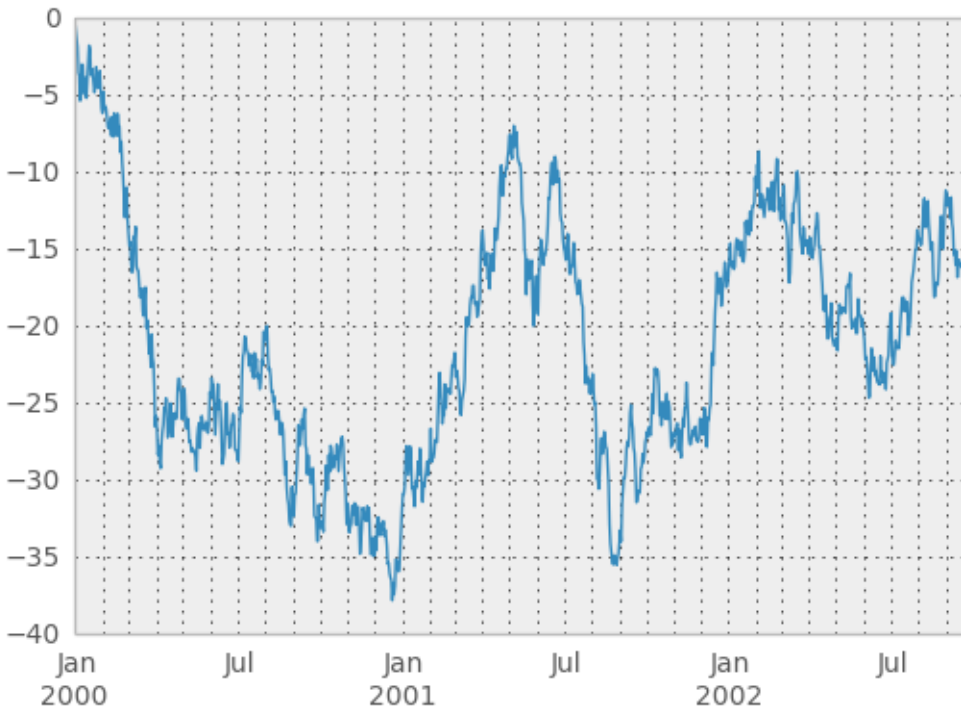
16.1.3 Suppressing tick resolution adjustment

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:

```
In [27]: plt.figure()
<matplotlib.figure.Figure at 0x1538e550>

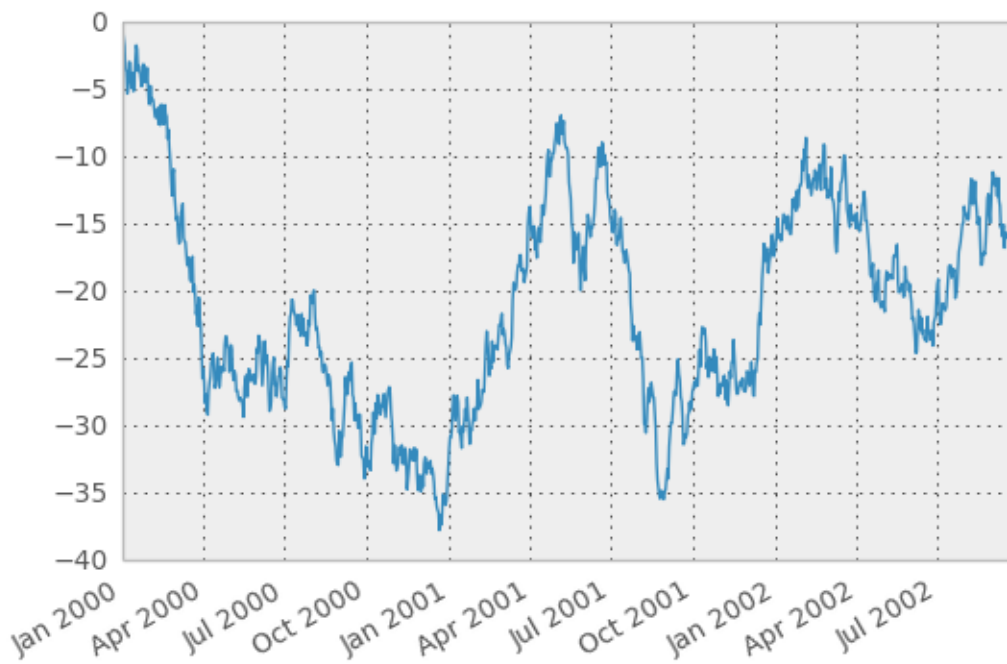
In [28]: df.A.plot()
<matplotlib.axes.AxesSubplot at 0x153bb2d0>
```



Using the `x_compat` parameter, you can suppress this behavior:

```
In [29]: plt.figure()
<matplotlib.figure.Figure at 0x1307c850>

In [30]: df.A.plot(x_compat=True)
<matplotlib.axes.AxesSubplot at 0x130a3fd0>
```



If you have more than one plot that needs to be suppressed, the use method in `pandas.plot_params` can be used

in a *with* statement:

```
In [31]: import pandas as pd

In [32]: plt.figure()
<matplotlib.figure.Figure at 0x1344d550>

In [33]: with pd.plot_params.use('x_compat', True):
.....:     df.A.plot(color='r')
.....:     df.B.plot(color='g')
.....:     df.C.plot(color='b')
.....:
```



16.1.4 Targeting different subplots

You can pass an `ax` argument to `Series.plot` to plot on a particular axis:

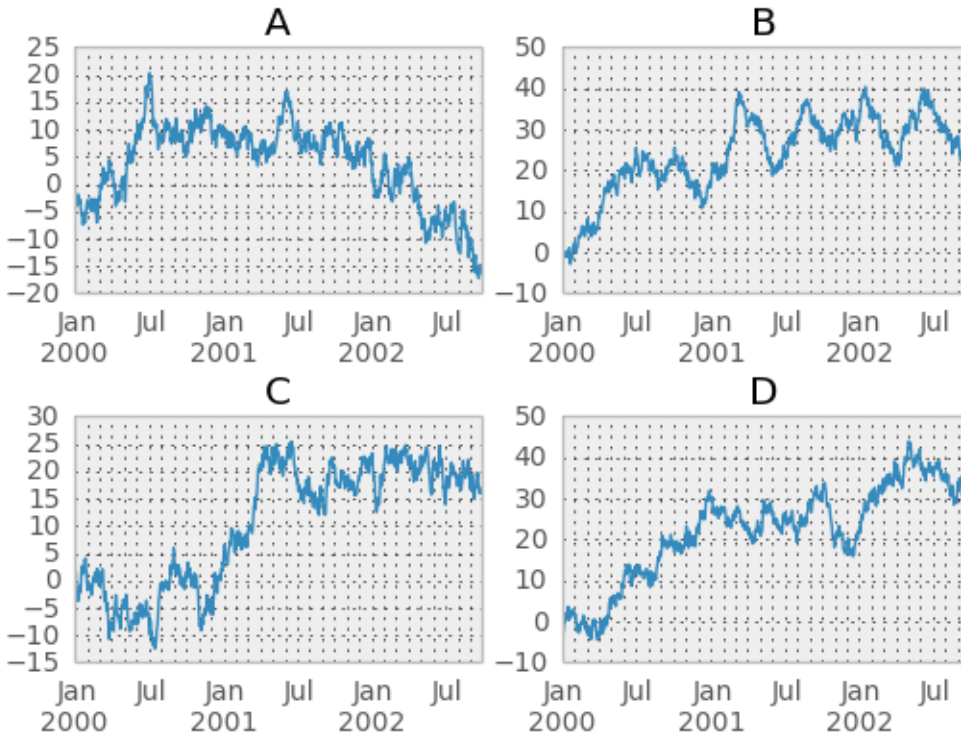
```
In [34]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [35]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A')
<matplotlib.text.Text at 0x16e559d0>

In [36]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B')
<matplotlib.text.Text at 0x16e7a3d0>

In [37]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C')
<matplotlib.text.Text at 0x16ff1890>

In [38]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D')
<matplotlib.text.Text at 0x17009a10>
```

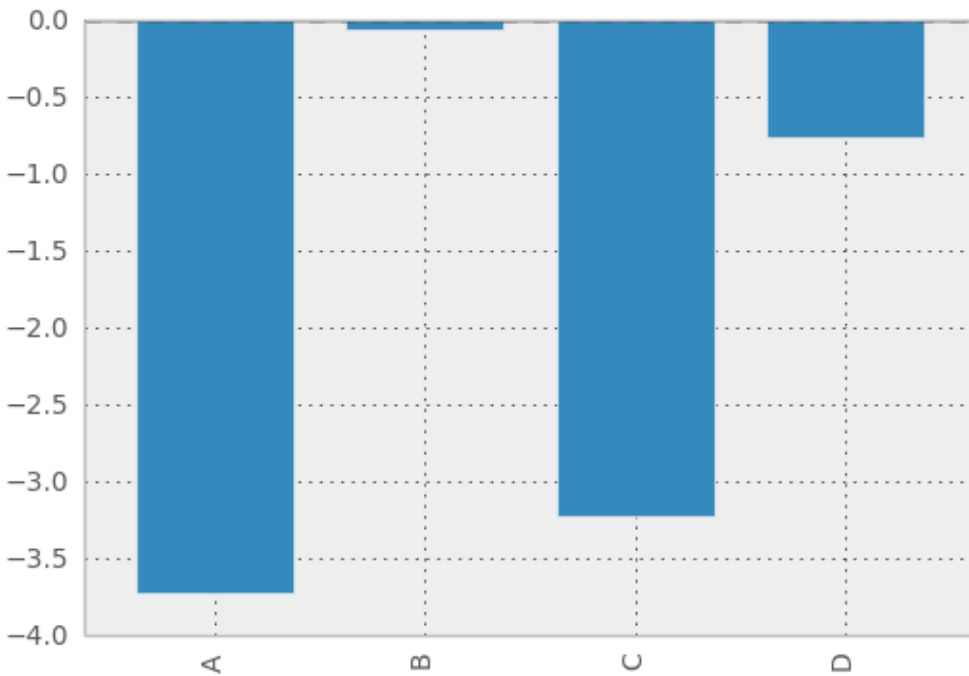


16.2 Other plotting features

16.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

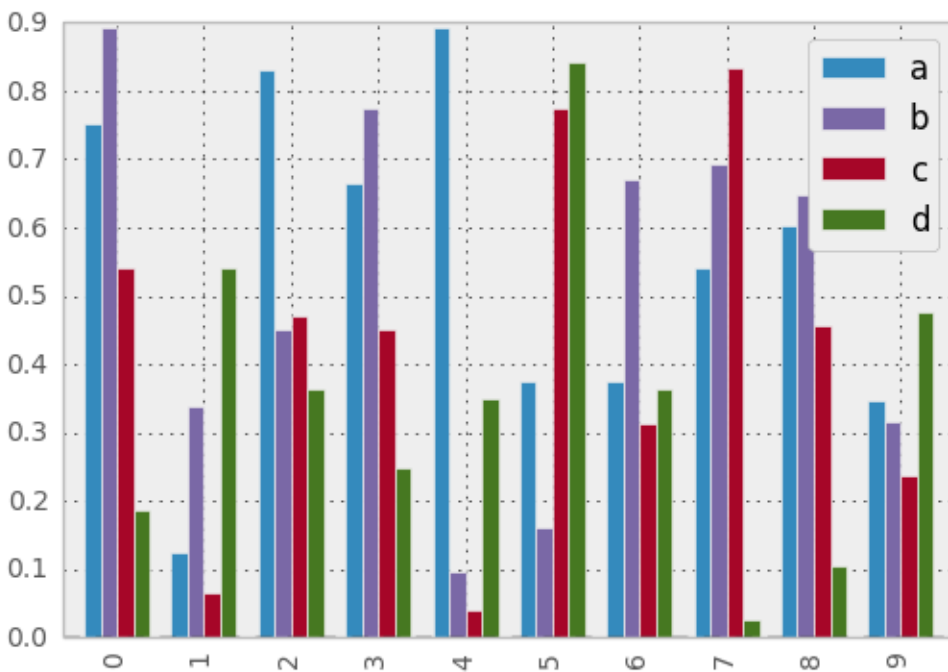
```
In [39]: plt.figure();
In [39]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
<matplotlib.lines.Line2D at 0x1796cdd0>
```



Calling a DataFrame's `plot` method with `kind='bar'` produces a multiple bar plot:

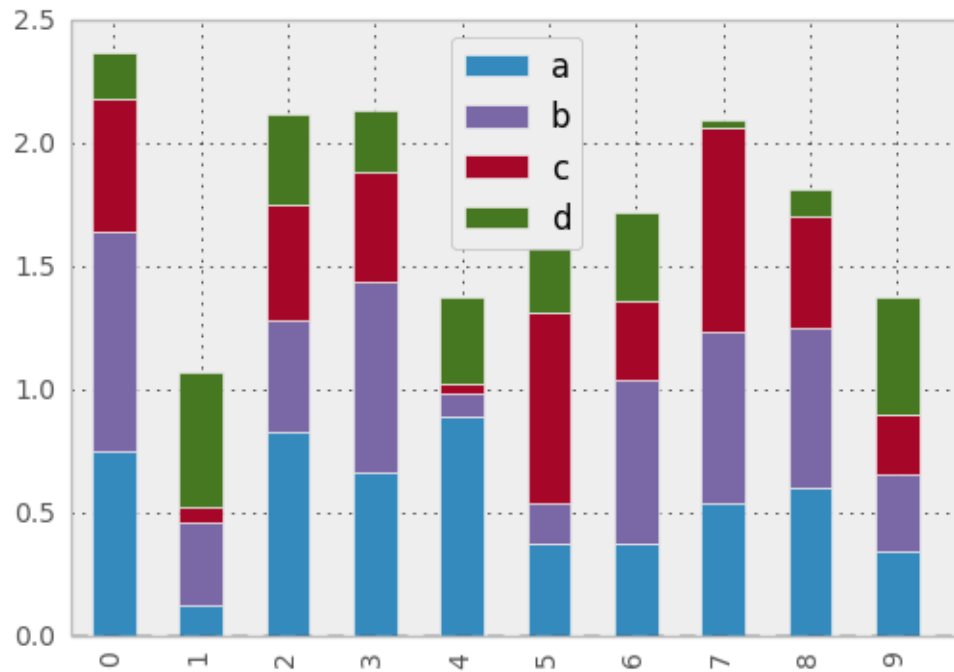
```
In [40]: df2 = DataFrame(rand(10, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [41]: df2.plot(kind='bar');
```



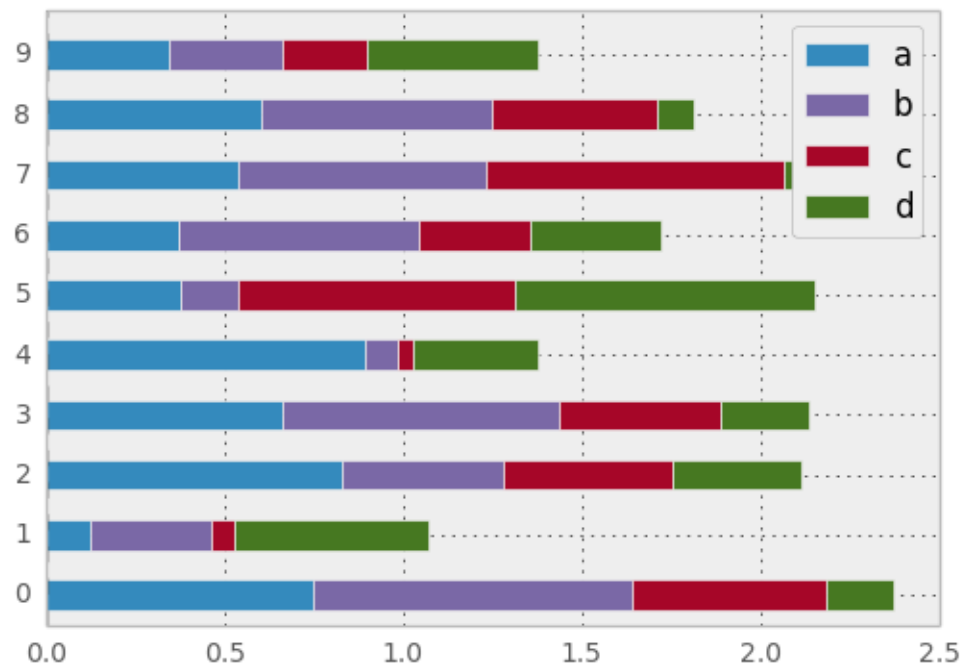
To produce a stacked bar plot, pass `stacked=True`:

```
In [41]: df2.plot(kind='bar', stacked=True);
```

To get horizontal bar plots, pass `kind='barh'`:

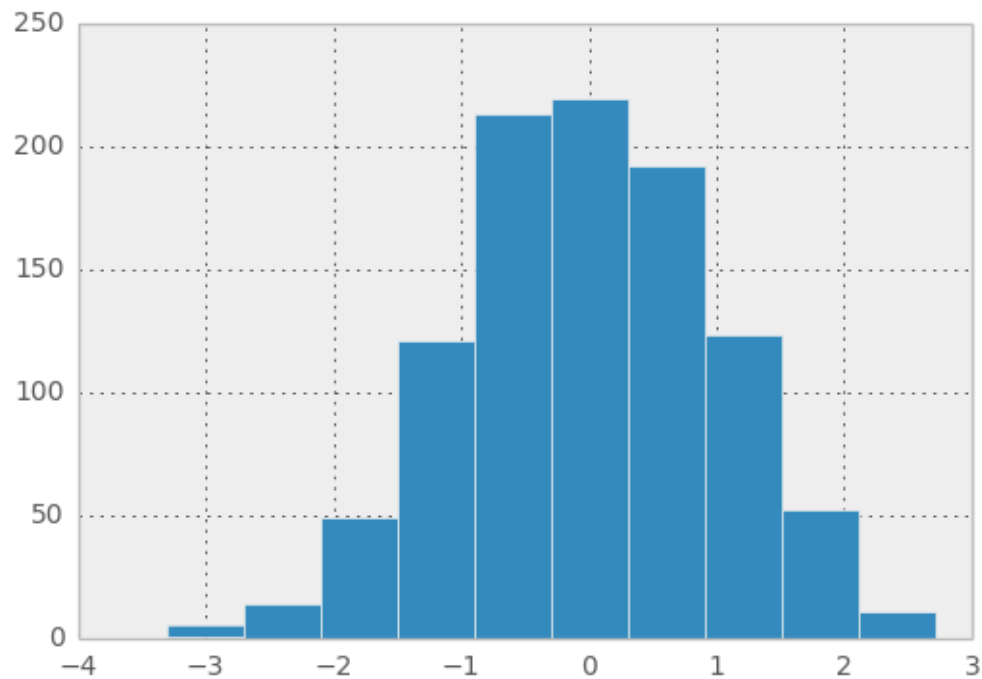
```
In [41]: df2.plot(kind='barh', stacked=True);
```



16.2.2 Histograms

```
In [41]: plt.figure();
In [41]: df['A'].diff().hist()
```

```
<matplotlib.axes.AxesSubplot at 0x18332b50>
```



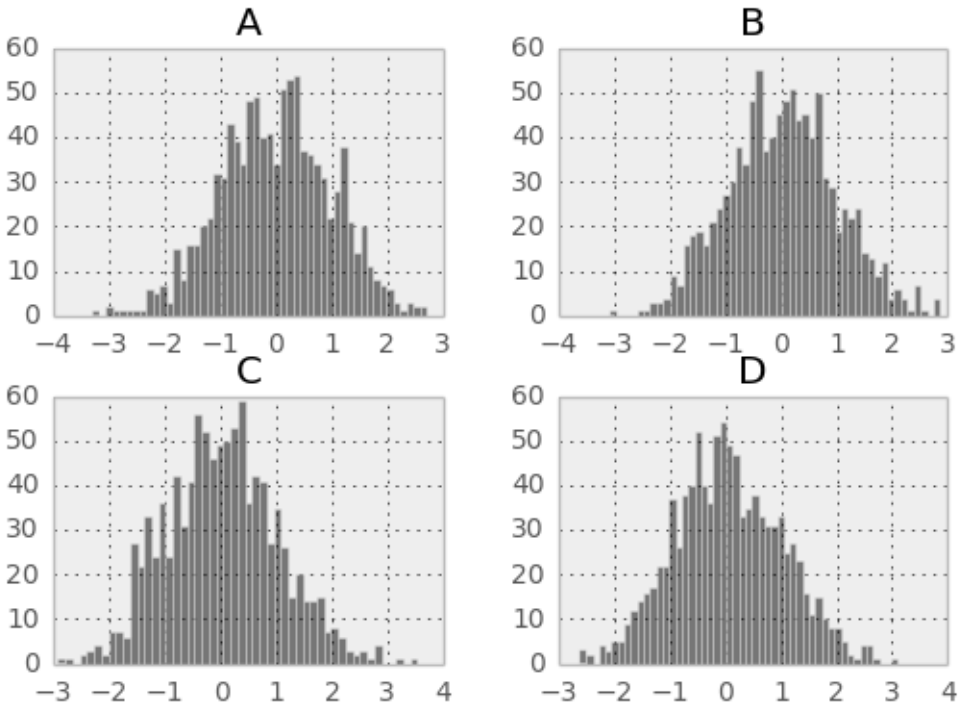
For a DataFrame, `hist` plots the histograms of the columns on multiple subplots:

```
In [42]: plt.figure()
```

```
<matplotlib.figure.Figure at 0x17fc4a90>
```

```
In [43]: df.diff().hist(color='k', alpha=0.5, bins=50)
```

```
array([[<matplotlib.axes.AxesSubplot object at 0x18777c90>,  
       <matplotlib.axes.AxesSubplot object at 0x1888e350>],  
       [<matplotlib.axes.AxesSubplot object at 0x18b3f7d0>,  
       <matplotlib.axes.AxesSubplot object at 0x18b62d90>]], dtype=object)
```

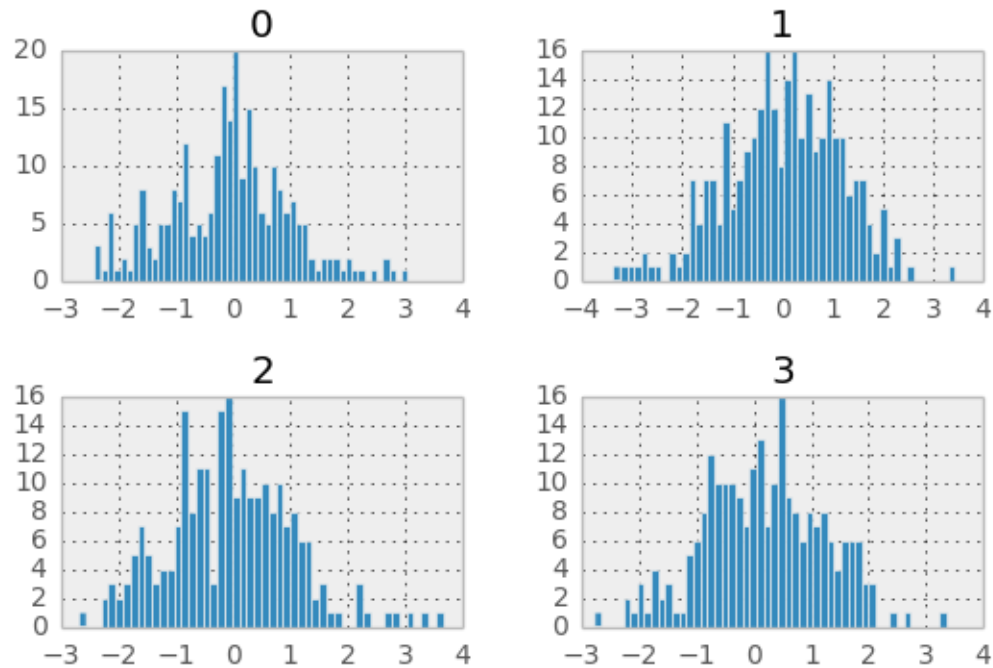


New since 0.10.0, the `by` keyword can be specified to plot grouped histograms:

```
In [44]: data = Series(randn(1000))
```

```
In [45]: data.hist(by=randint(0, 4, 1000), figsize=(6, 4))
```

```
array([[<matplotlib.axes.AxesSubplot object at 0x18de7f90>,
        <matplotlib.axes.AxesSubplot object at 0x195a47d0>],
       [<matplotlib.axes.AxesSubplot object at 0x1974e290>,
        <matplotlib.axes.AxesSubplot object at 0x195afc90>]], dtype=object)
```



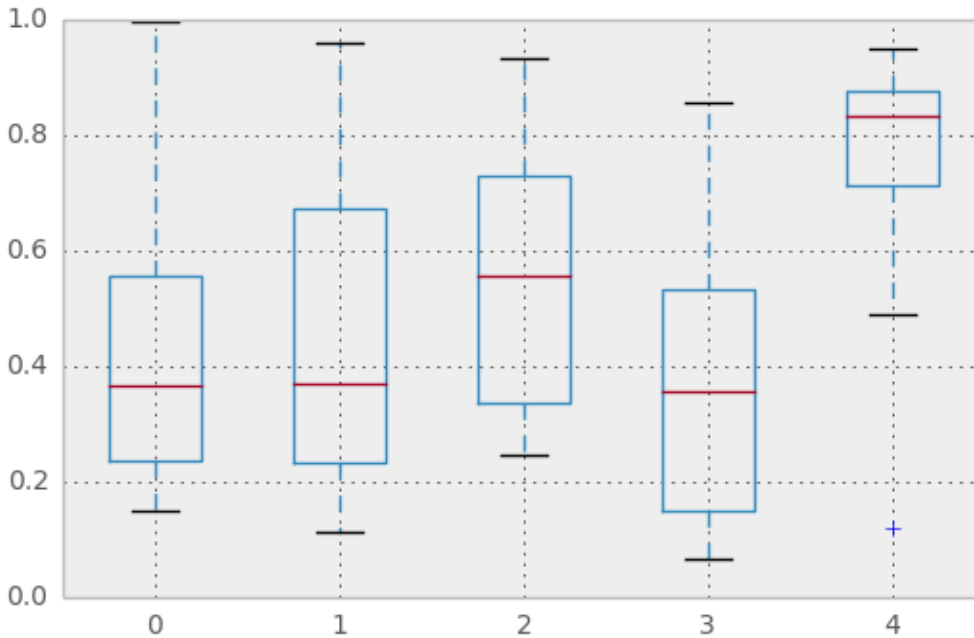
16.2.3 Box-Plotting

`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 [46]: df = DataFrame(rand(10,5))
```

```
In [47]: plt.figure();
```

```
In [47]: bp = df.boxplot()
```



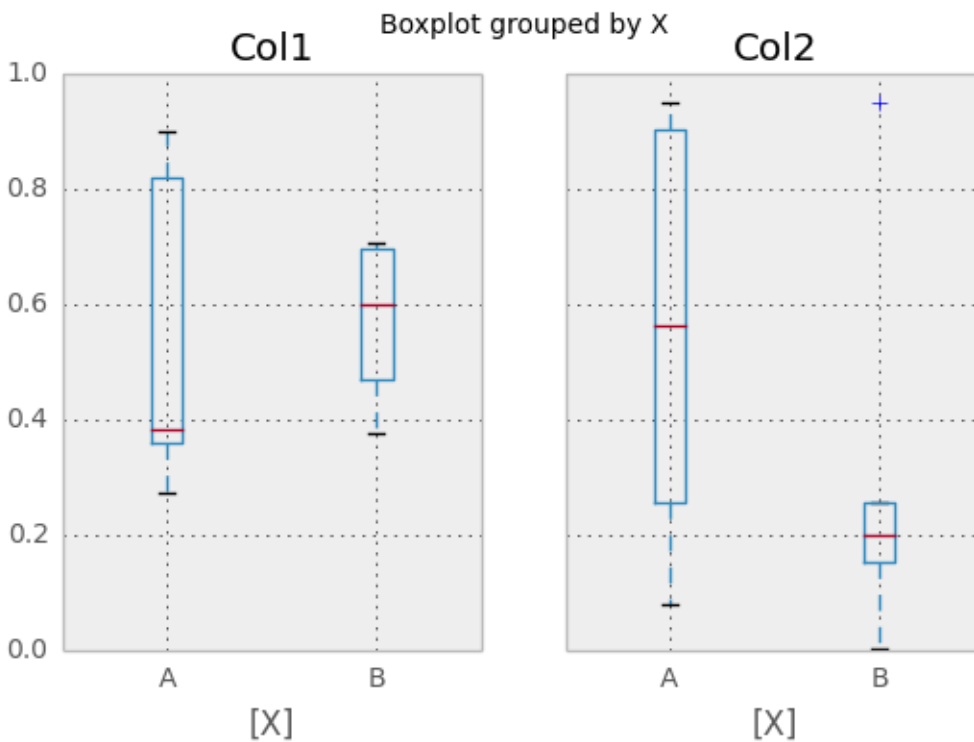
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [48]: df = DataFrame(rand(10,2), columns=['Col1', 'Col2'] )
```

```
In [49]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])
```

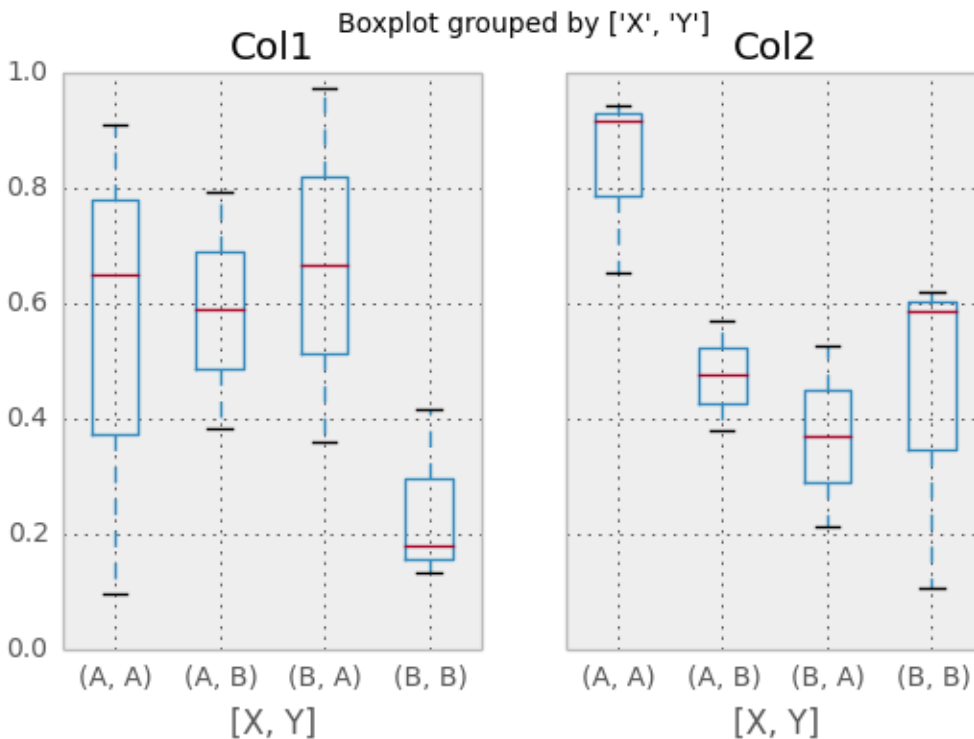
```
In [50]: plt.figure();
```

```
In [50]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [51]: df = DataFrame(rand(10,3), columns=['Col1', 'Col2', 'Col3'])
In [52]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])
In [53]: df['Y'] = Series(['A','B','A','B','A','B','A','B','A','B'])
In [54]: plt.figure();
In [54]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```



16.2.4 Scatter plot matrix

New in 0.7.3. You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.tools.plotting`:

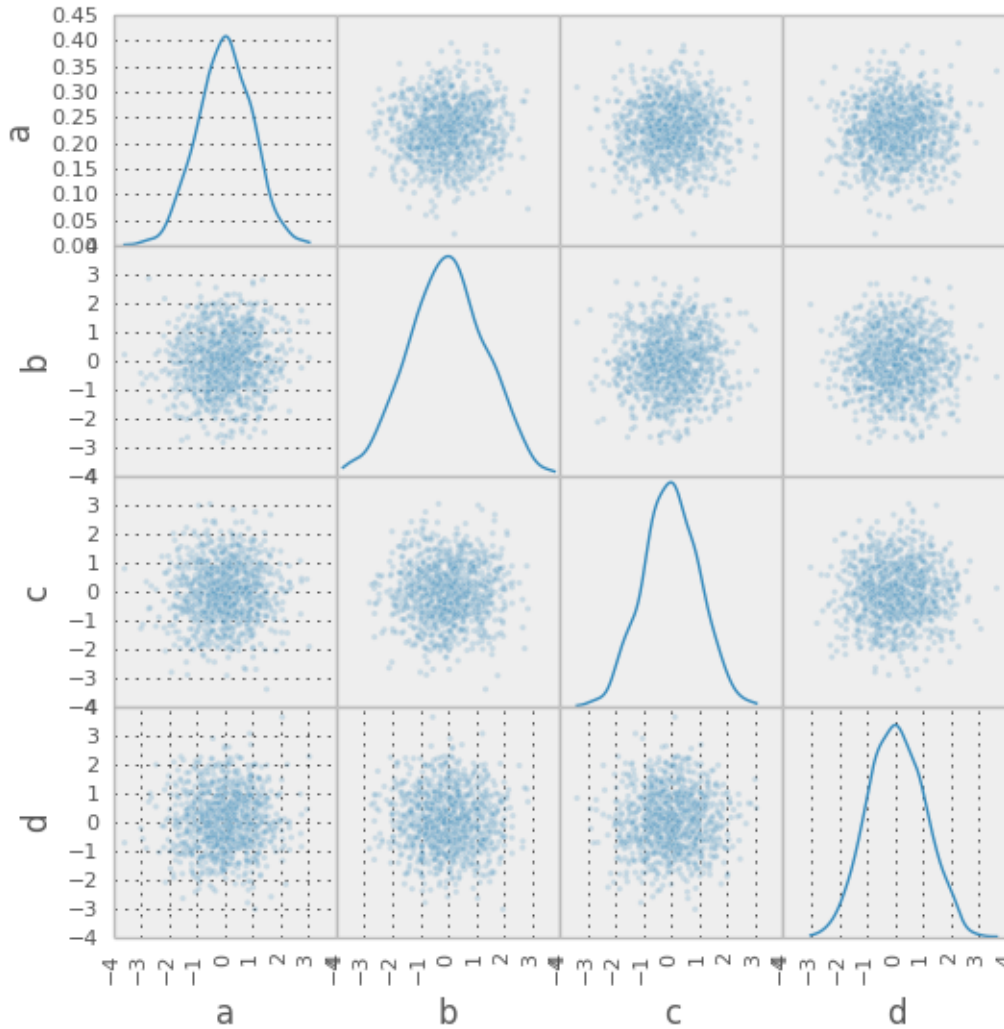
```
In [55]: from pandas.tools.plotting import scatter_matrix
In [56]: df = DataFrame(randn(1000, 4), columns=['a', 'b', 'c', 'd'])
In [57]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
```

```
array([[<matplotlib.axes.AxesSubplot object at 0x19e66850>,
        <matplotlib.axes.AxesSubplot object at 0x1acd7d90>,
        <matplotlib.axes.AxesSubplot object at 0x1adca8d0>,
        <matplotlib.axes.AxesSubplot object at 0x1b0e1610>],
       [<matplotlib.axes.AxesSubplot object at 0x1b0fef10>,
        <matplotlib.axes.AxesSubplot object at 0x1b260790>,
        <matplotlib.axes.AxesSubplot object at 0x1b3c04d0>,
        <matplotlib.axes.AxesSubplot object at 0x1b3df090>],
       [<matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>],
       [<matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>,
        <matplotlib.axes.AxesSubplot object at 0x1b548e90>]])
```

```

<matplotlib.axes.AxesSubplot object at 0x1b3caa10>,
<matplotlib.axes.AxesSubplot object at 0x1b6d1810>,
<matplotlib.axes.AxesSubplot object at 0x1b6f7210>],
[<matplotlib.axes.AxesSubplot object at 0x1b848b50>,
<matplotlib.axes.AxesSubplot object at 0x1b86c990>,
<matplotlib.axes.AxesSubplot object at 0x1b9bbad0>,
<matplotlib.axes.AxesSubplot object at 0x1b9eed10>]], dtype=object)

```

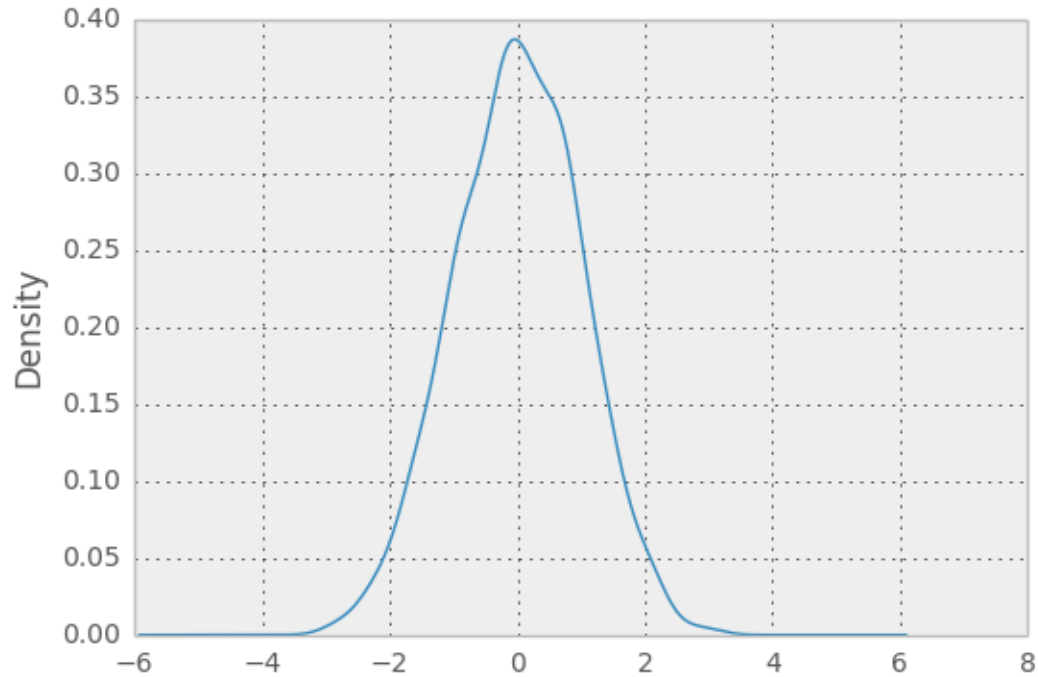


New in 0.8.0 You

can create density plots using the Series/DataFrame.plot and setting `kind='kde'`:

```
In [58]: ser = Series(randn(1000))
```

```
In [59]: ser.plot(kind='kde')
<matplotlib.axes.AxesSubplot at 0x1c5fca90>
```



16.2.5 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](#).

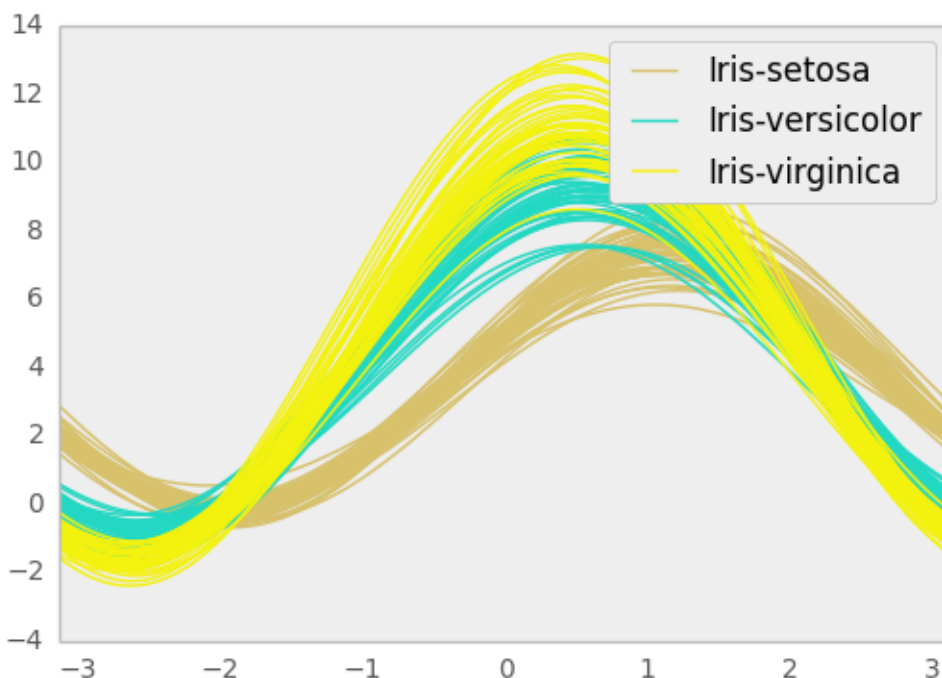
```
In [60]: from pandas import read_csv

In [61]: from pandas.tools.plotting import andrews_curves

In [62]: data = read_csv('data/iris.data')

In [63]: plt.figure()
<matplotlib.figure.Figure at 0x1c61a090>

In [64]: andrews_curves(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x1c019a90>
```

16.2.6 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.

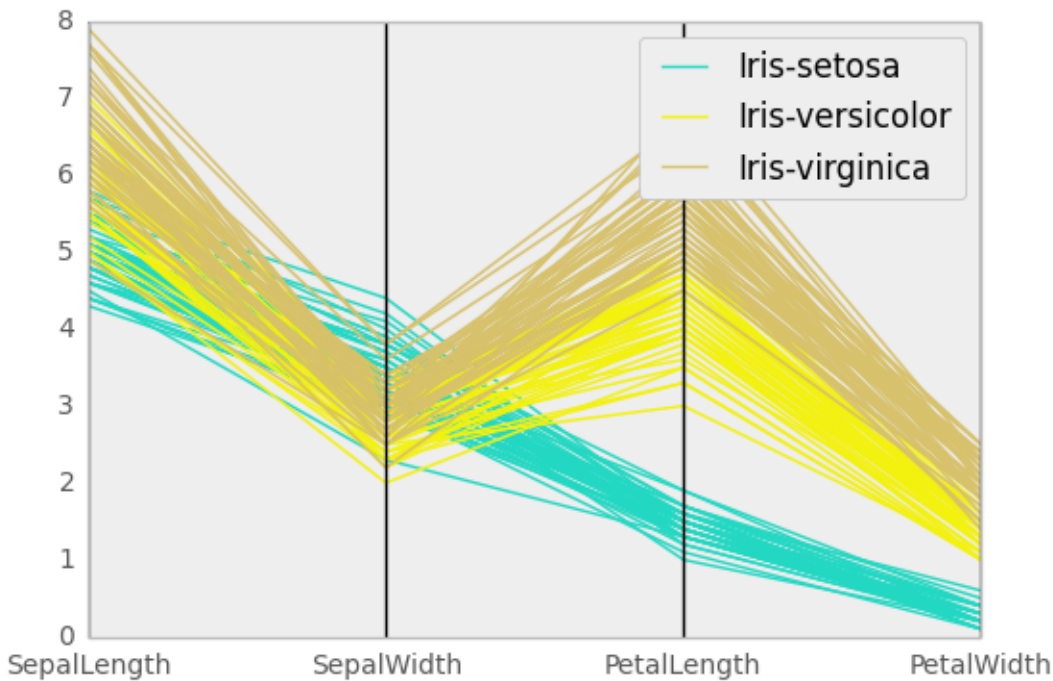
```
In [65]: from pandas import read_csv

In [66]: from pandas.tools.plotting import parallel_coordinates

In [67]: data = read_csv('data/iris.data')

In [68]: plt.figure()
<matplotlib.figure.Figure at 0x1d058450>

In [69]: parallel_coordinates(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x1d058bd0>
```



16.2.7 Lag Plot

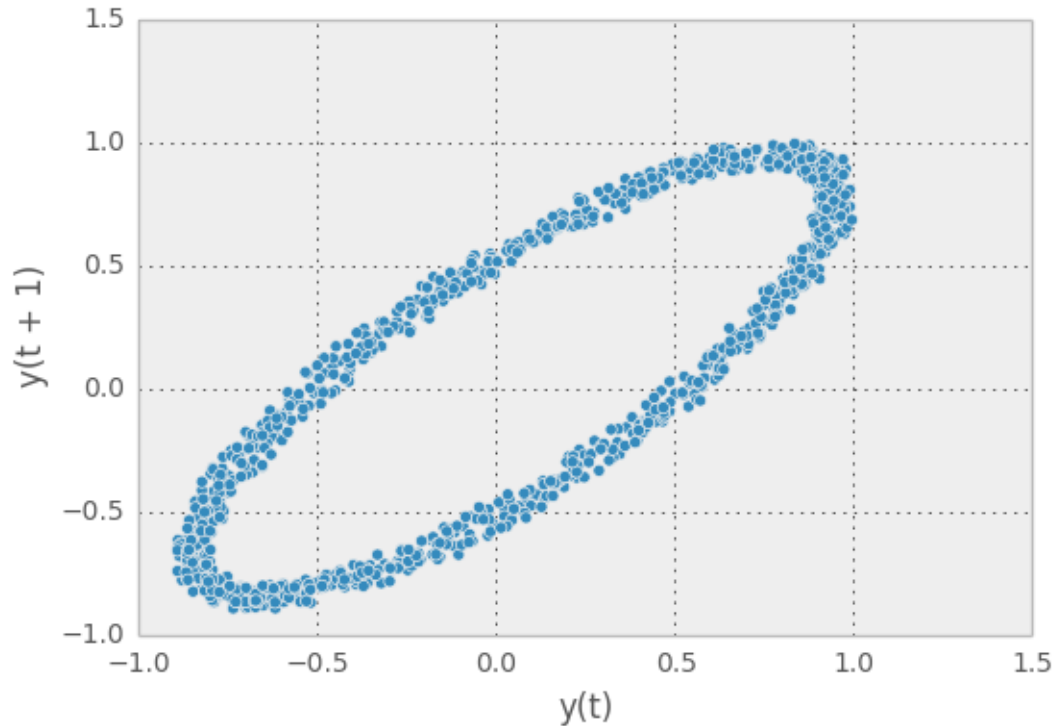
Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random.

```
In [70]: from pandas.tools.plotting import lag_plot

In [71]: plt.figure()
<matplotlib.figure.Figure at 0x1d7c6110>

In [72]: data = Series(0.1 * rand(1000) +
.....:    0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
.....:

In [73]: lag_plot(data)
<matplotlib.axes.AxesSubplot at 0x1d7c6d90>
```



16.2.8 Autocorrelation Plot

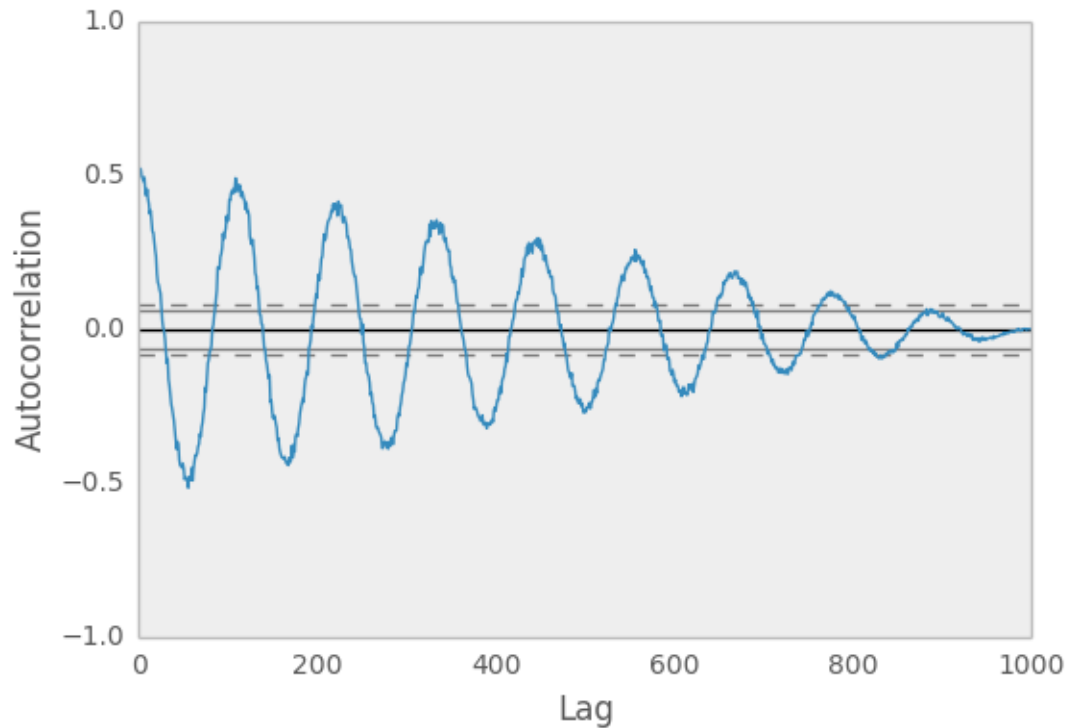
Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one 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 [74]: from pandas.tools.plotting import autocorrelation_plot
```

```
In [75]: plt.figure()
<matplotlib.figure.Figure at 0x1d627750>
```

```
In [76]: data = Series(0.7 * rand(1000) +
.....:    0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
.....:
```

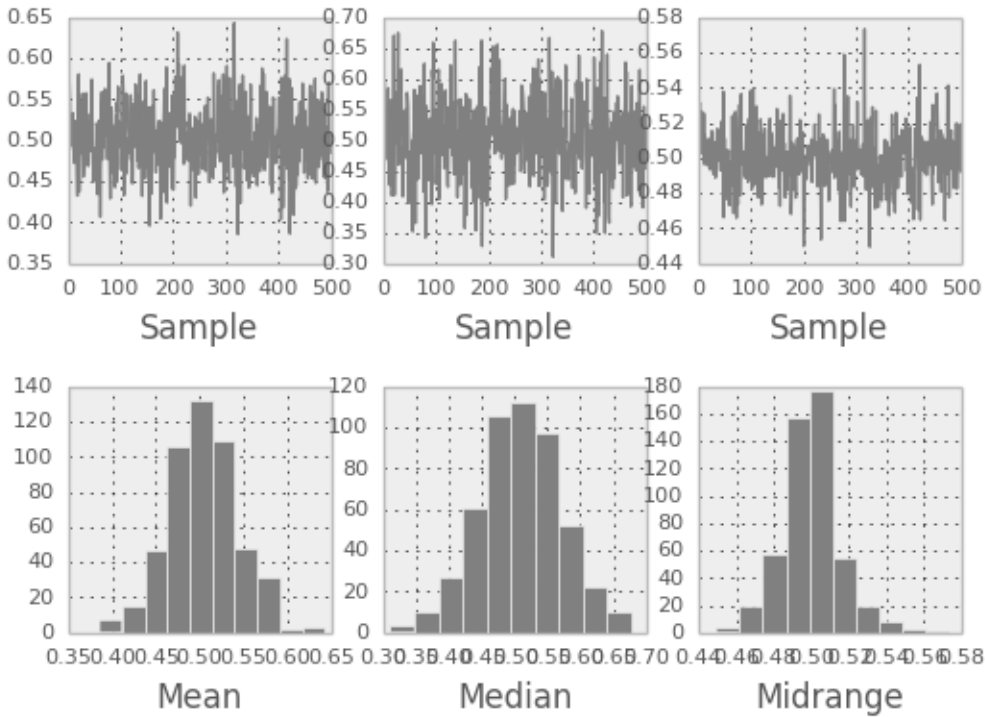
```
In [77]: autocorrelation_plot(data)
<matplotlib.axes.AxesSubplot at 0x1d4f6fd0>
```



16.2.9 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 [78]: from pandas.tools.plotting import bootstrap_plot
In [79]: data = Series(rand(1000))
In [80]: bootstrap_plot(data, size=50, samples=500, color='grey')
<matplotlib.figure.Figure at 0x1d0587d0>
```



16.2.10 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 to it will be colored differently.

Note: The “Iris” dataset is available [here](#).

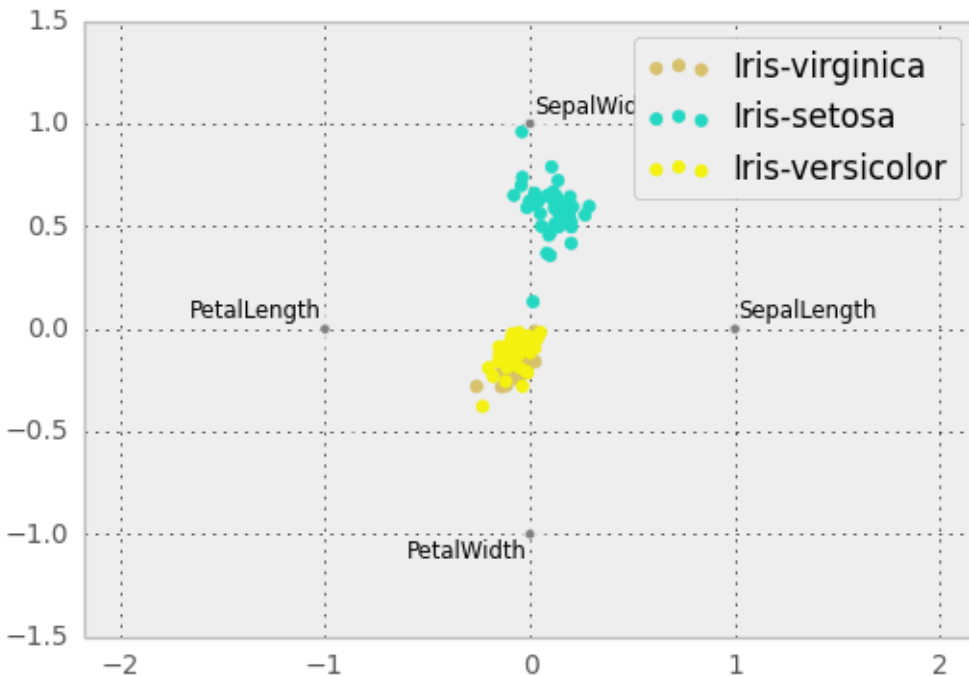
```
In [81]: from pandas import read_csv

In [82]: from pandas.tools.plotting import radviz

In [83]: data = read_csv('data/iris.data')

In [84]: plt.figure()
<matplotlib.figure.Figure at 0x1dfdc6d0>

In [85]: radviz(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x1e296910>
```



16.2.11 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.

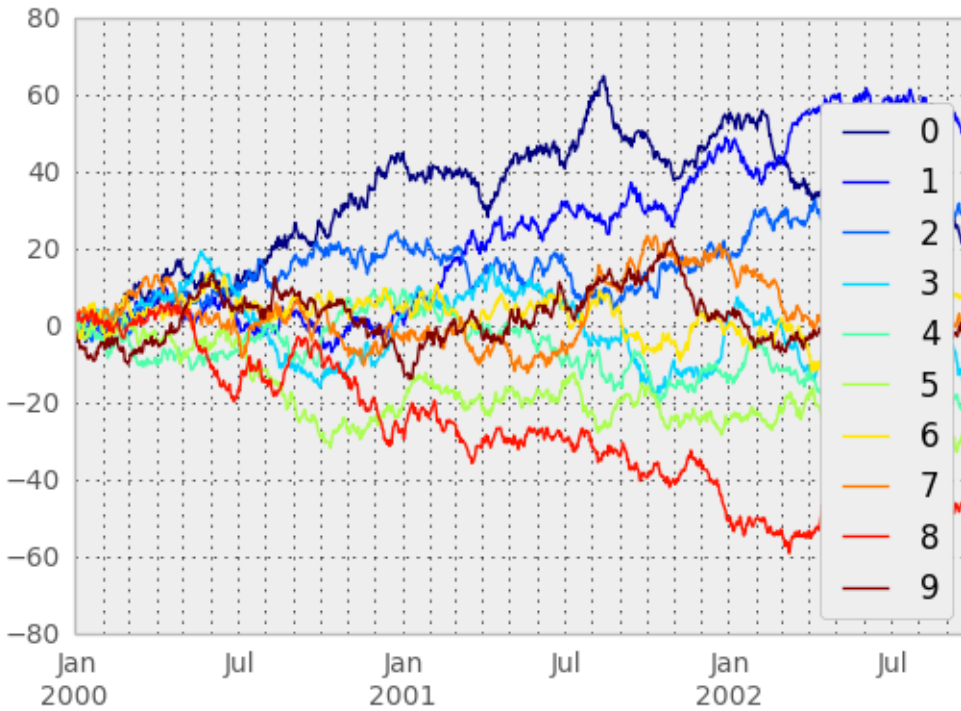
To use the jet colormap, we can simply pass `'jet'` to `colormap=`

```
In [86]: df = DataFrame(randn(1000, 10), index=ts.index)
```

```
In [87]: df = df.cumsum()
```

```
In [88]: plt.figure()
<matplotlib.figure.Figure at 0x1e7a1c90>
```

```
In [89]: df.plot(colormap='jet')
<matplotlib.axes.AxesSubplot at 0x1e7af790>
```

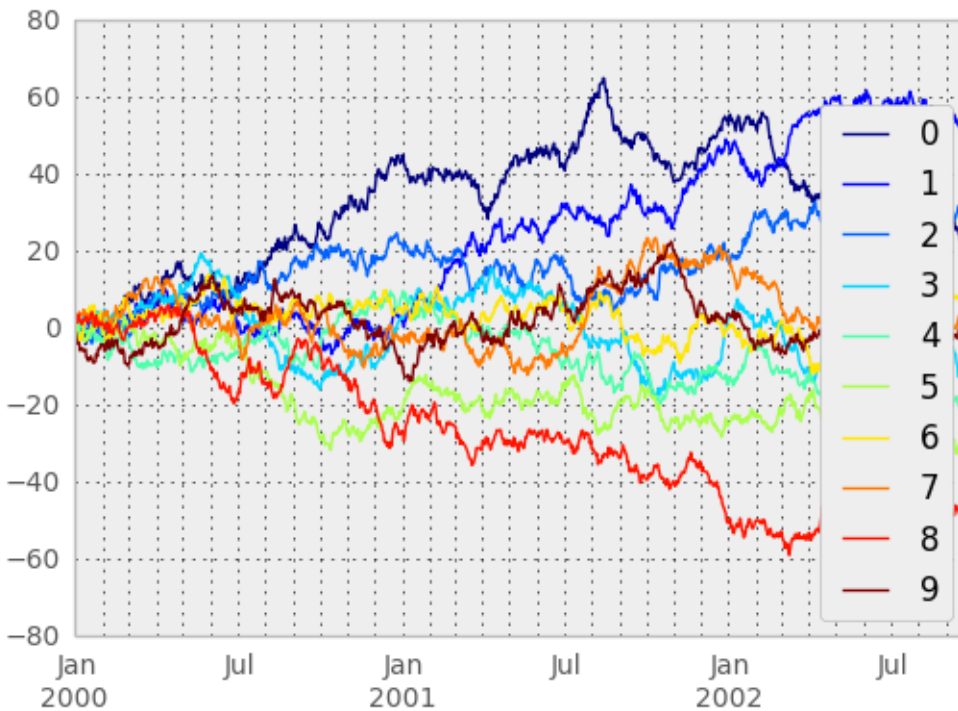


or we can pass the colormap itself

```
In [90]: from matplotlib import cm
```

```
In [91]: plt.figure()  
<matplotlib.figure.Figure at 0x1ee1ff90>
```

```
In [92]: df.plot(colormap=cm.jet)  
<matplotlib.axes.AxesSubplot at 0x1ee1ac90>
```



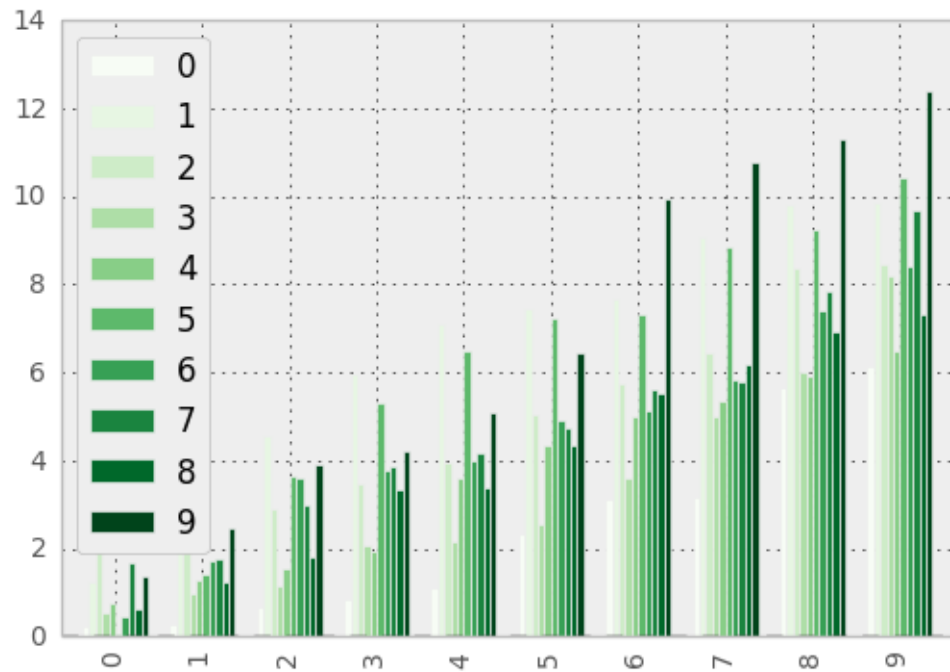
Colormaps can also be used other plot types, like bar charts:

```
In [93]: dd = DataFrame(randn(10, 10)).applymap(abs)
```

```
In [94]: dd = dd.cumsum()
```

```
In [95]: plt.figure()  
<matplotlib.figure.Figure at 0x1e7a1810>
```

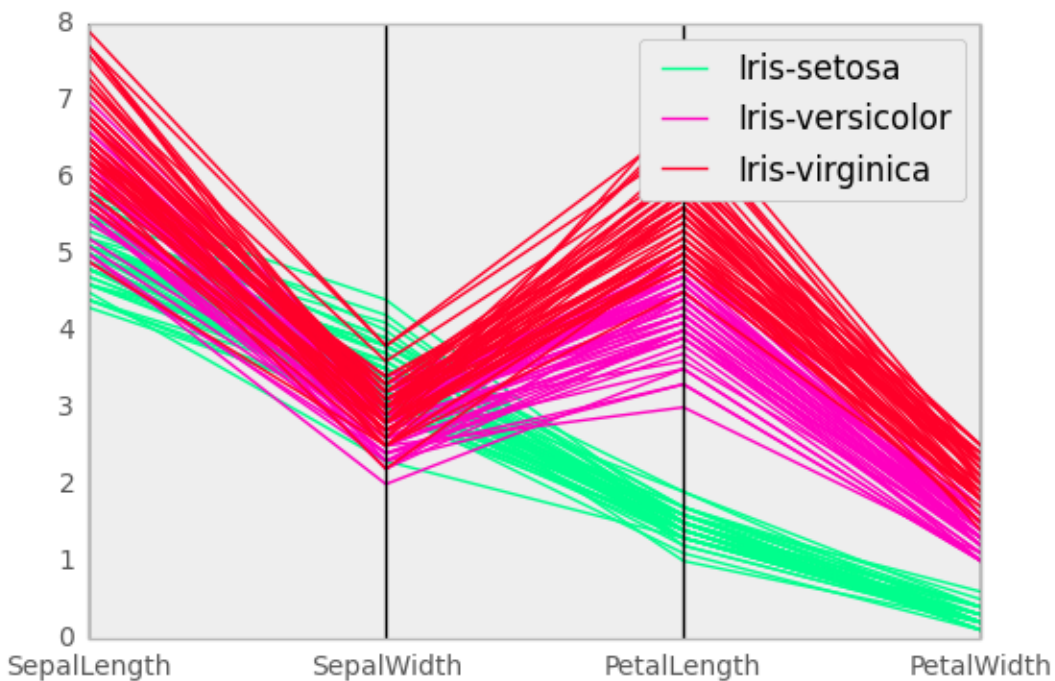
```
In [96]: dd.plot(kind='bar', colormap='Greens')  
<matplotlib.axes.AxesSubplot at 0x1eba09d0>
```

Parallel coordinates charts:

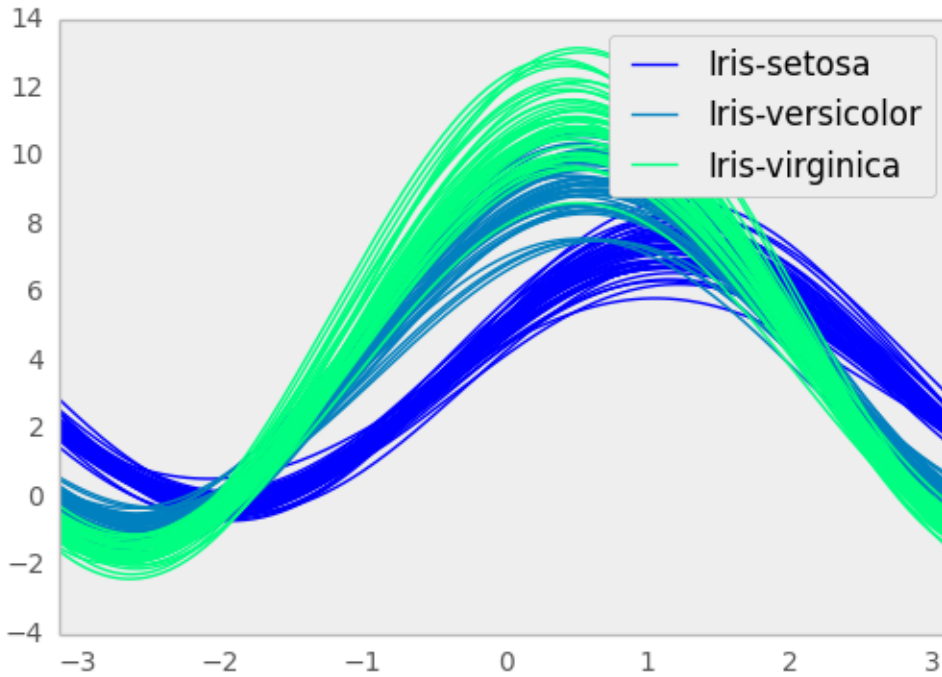
```
In [97]: plt.figure()
<matplotlib.figure.Figure at 0x1f1f13d0>
```

```
In [98]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
<matplotlib.axes.AxesSubplot at 0x1f6aa790>
```



Andrews curves charts:

```
In [99]: plt.figure()  
<matplotlib.figure.Figure at 0x1fed2250>  
  
In [100]: andrews_curves(data, 'Name', colormap='winter')  
<matplotlib.axes.AxesSubplot at 0x20013850>
```



TRELLIS PLOTTING INTERFACE

Note: The tips data set can be downloaded [here](#). Once you download it execute

```
from pandas import read_csv
tips_data = read_csv('tips.csv')
```

from the directory where you downloaded the file.

We import the rplot API:

```
In [1]: import pandas.tools.rplot as rplot
```

17.1 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.

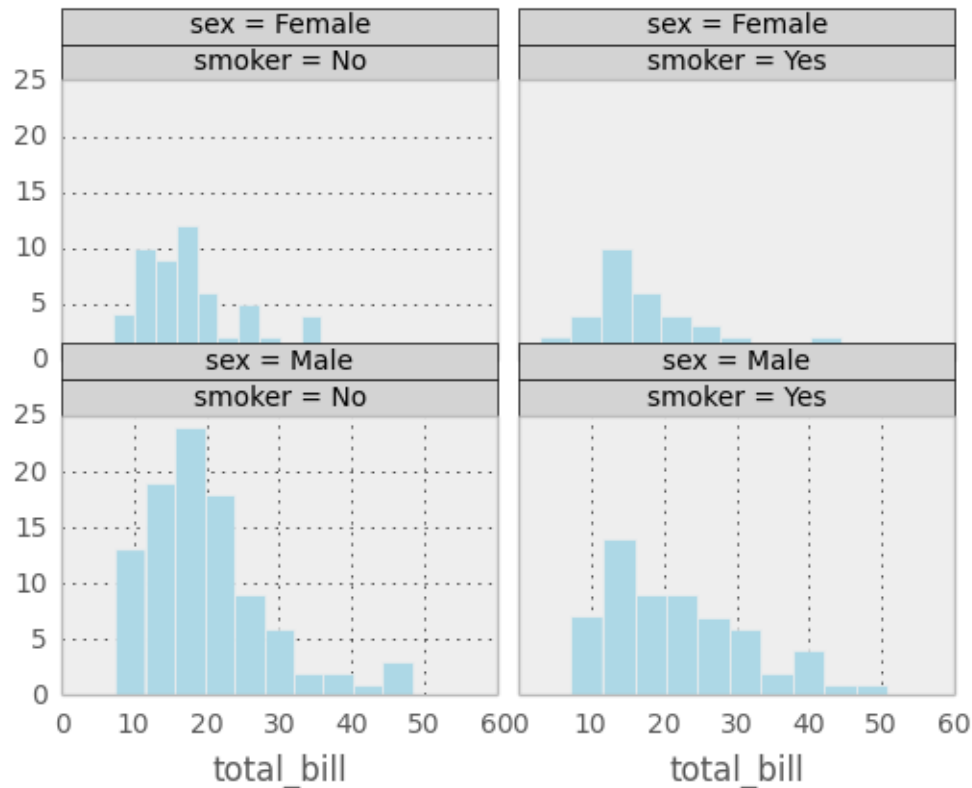
```
In [2]: plt.figure()
<matplotlib.figure.Figure at 0x6d83f50>
```

```
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.GeoHistogram())
```

```
In [6]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x6d83f50>
```



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.

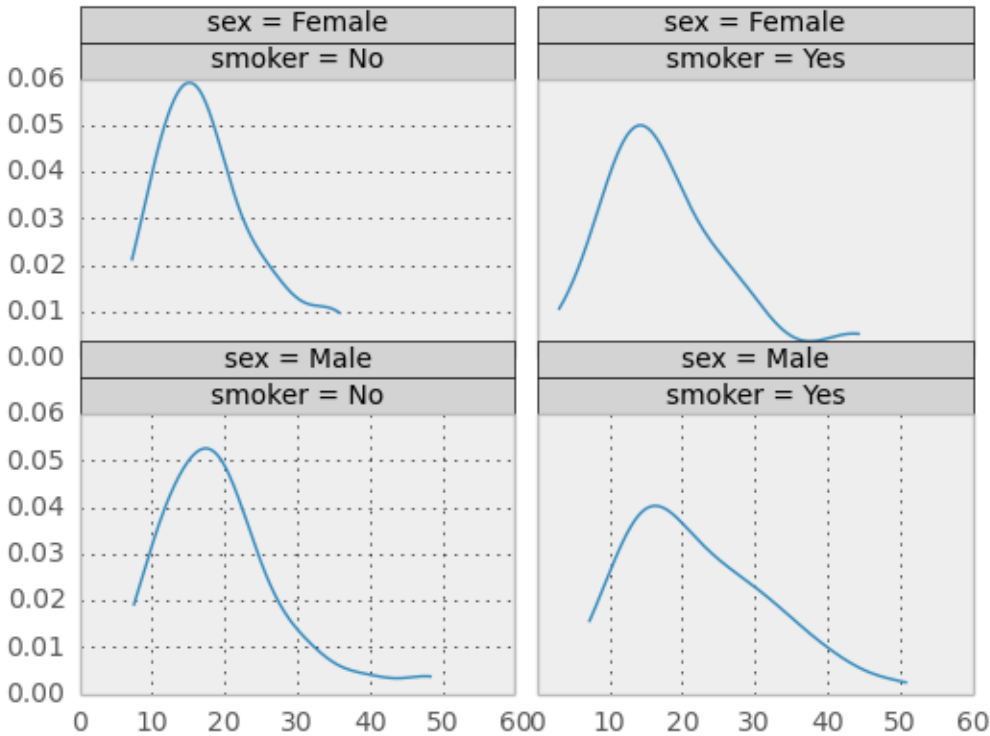
```
In [7]: plt.figure()
<matplotlib.figure.Figure at 0x6cce150>

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.GeoDensity())

In [11]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x6cce150>
```



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()
<matplotlib.figure.Figure at 0x11a38ed0>

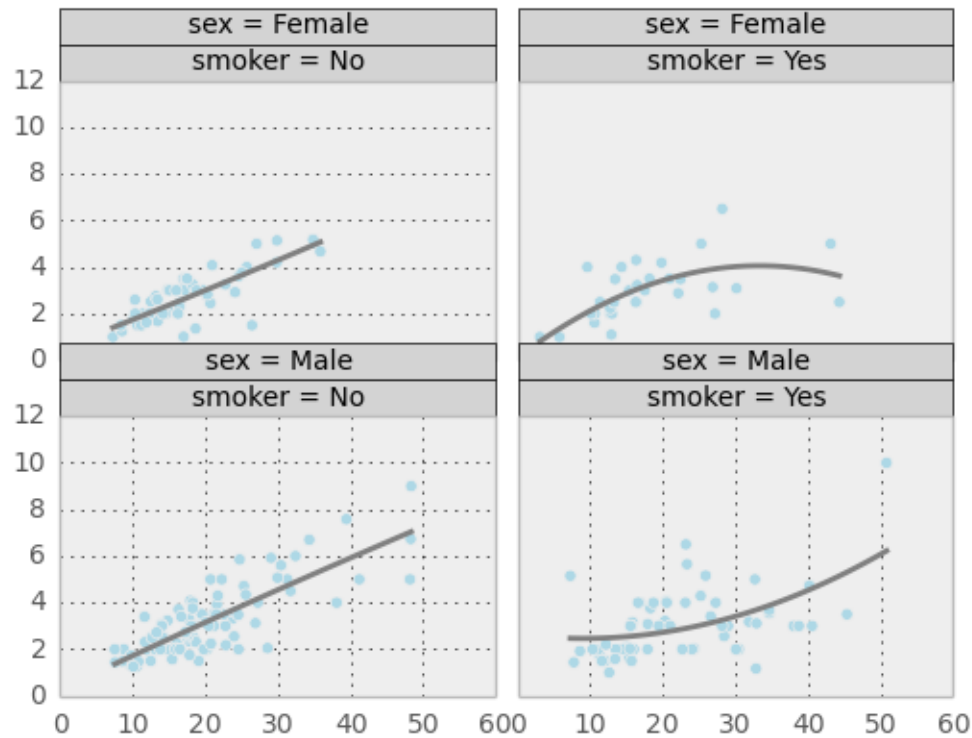
In [13]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

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())
<matplotlib.figure.Figure at 0x11a38ed0>
```



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()
<matplotlib.figure.Figure at 0x6f27dd0>

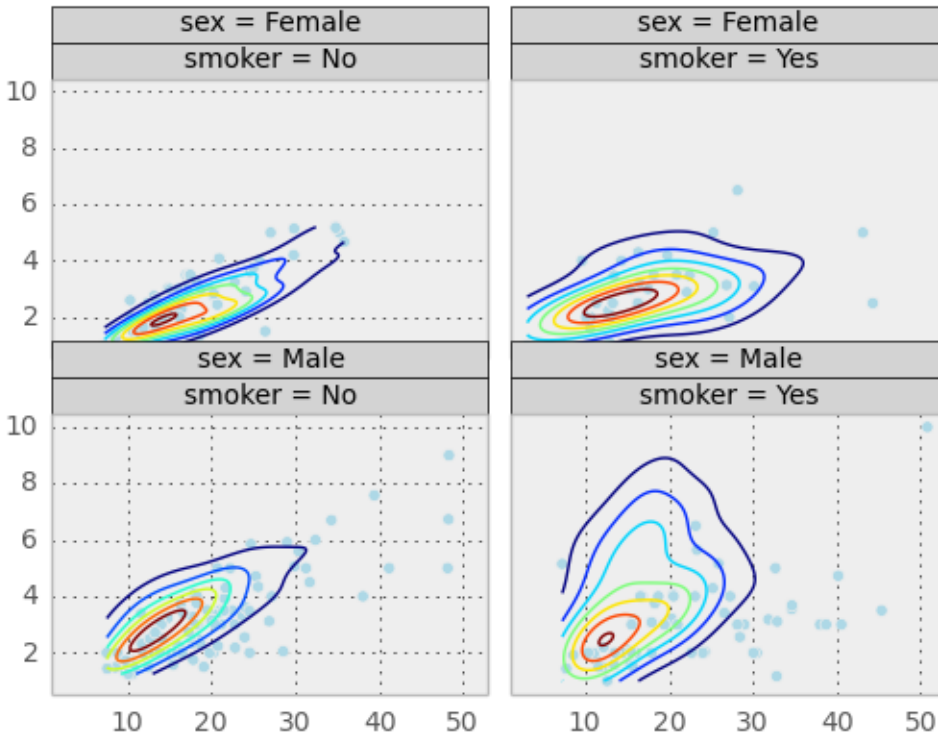
In [19]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [20]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [21]: plot.add(rplot.GeoScatter())

In [22]: plot.add(rplot.GeoDensity2D())

In [23]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x6f27dd0>
```



Above is a similar plot but with 2D kernel density estimation plot superimposed.

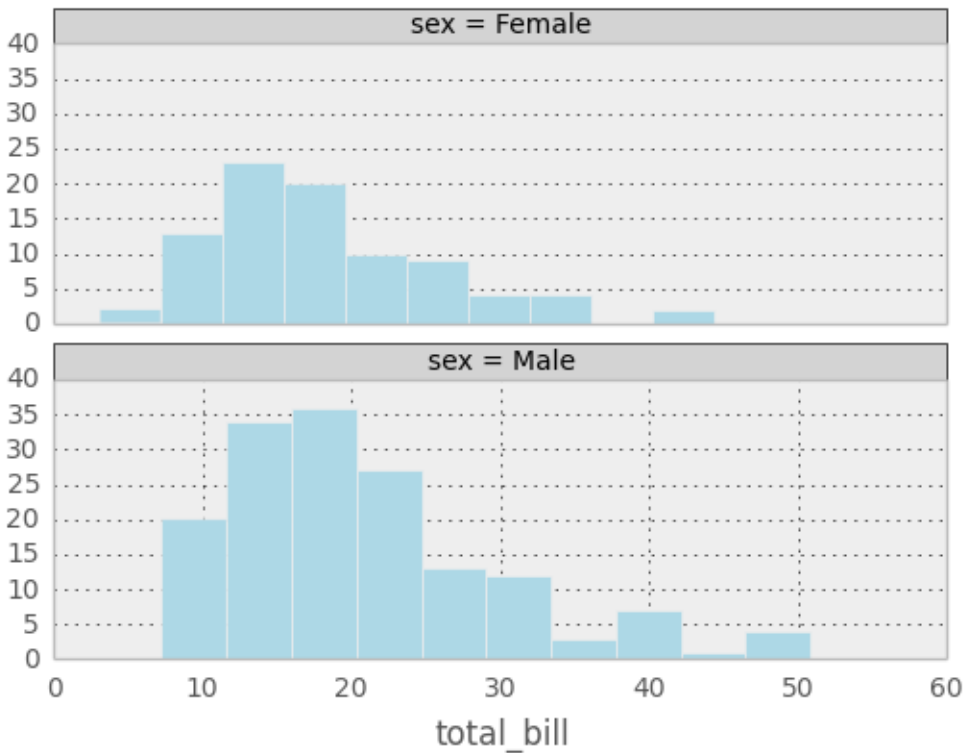
```
In [24]: plt.figure()
<matplotlib.figure.Figure at 0x1357f5d0>

In [25]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [26]: plot.add(rplot.TrellisGrid(['sex', '.']))

In [27]: plot.add(rplot.GeoHistogram())

In [28]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x1357f5d0>
```



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.

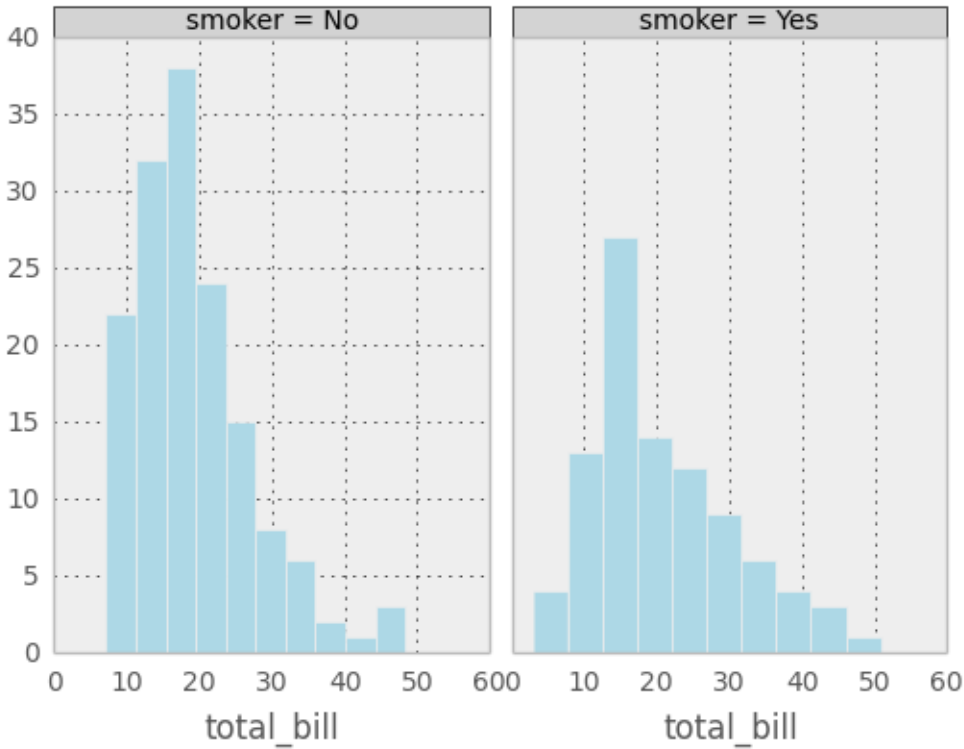
```
In [29]: plt.figure()
<matplotlib.figure.Figure at 0x13598fd0>

In [30]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [31]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [32]: plot.add(rplot.GeoHistogram())

In [33]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x13598fd0>
```

If the first grouping attribute is not specified the plots will be arranged in a row.

```
In [34]: plt.figure()
<matplotlib.figure.Figure at 0x1359c750>
```

```
In [35]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
```

```
In [36]: plot.add(rplot.TrellisGrid(['.', 'smoker']))
```

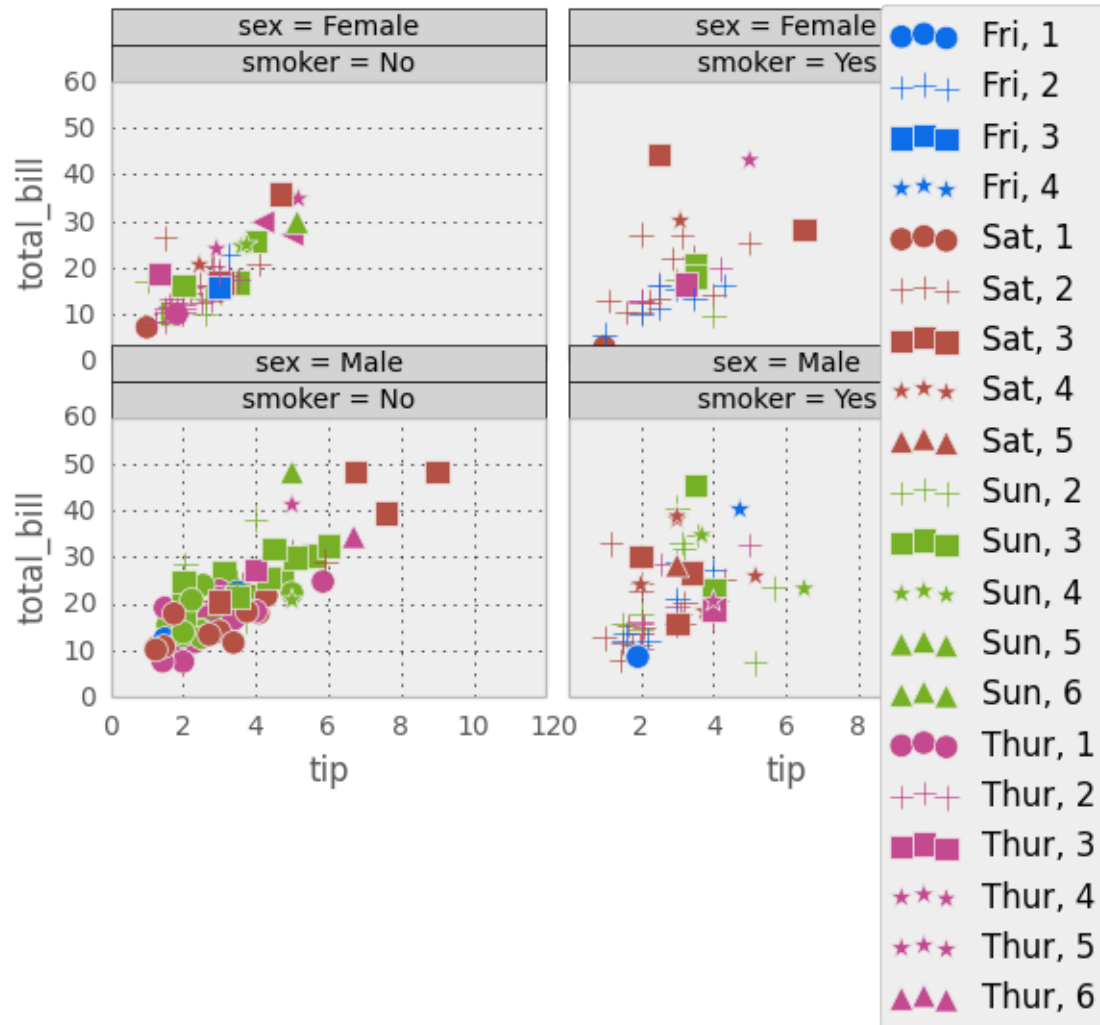
```
In [37]: plot.add(rplot.GeoHistogram())
```

```
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.GeoPoint(size=80.0, colour=rplot.ScaleRandomColour('day'), shape=rplot.ScaleRandomColour('day')))
```

```
In [41]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x1359c750>
```



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.

17.2 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*

18.1 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 `csv.Dialect` instance to expose more ways to specify the file format
- `dtype`: A data type name or a dict of column name to data type. If not specified, data types will be inferred.
- `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 `dateutil.parser`. Specifying this implicitly sets `parse_dates` as `True`. You can also use functions from community supported date converters from `date_converters.py`
- `dayfirst`: if `True` then uses the DD/MM international/European date format (This is `False` by default)

- `thousands`: specifies the thousands separator. If not `None`, this character will be stripped from numeric dtypes. However, if it is the first character in a field, that column will be imported as a string. In the `PythonParser`, if not `None`, then parser will try to look for it in the output and parse relevant data to numeric dtypes. Because it has to essentially scan through the data again, this causes a significant performance hit so only use if necessary.
- `lineterminator`: string (length 1), default `None`, Character to break file into lines. Only valid with `C` parser
- `quotechar`: string, The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.
- `quoting`: int, Controls whether quotes should be recognized. Values are taken from `csv.QUOTE_*` values. Acceptable values are 0, 1, 2, and 3 for `QUOTE_MINIMAL`, `QUOTE_ALL`, `QUOTE_NONE`, and `QUOTE_NONNUMERIC`, respectively.
- `skipinitialspace`: boolean, default `False`, Skip spaces after delimiter
- `escapechar`: string, to specify how to escape quoted data
- `comment`: 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:

```
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:

```
In [2]: pd.read_csv('foo.csv')

   date  A  B  C
0  20090101  a  1  2
```

```
1 20090102 b 3 4
2 20090103 c 4 5
[3 rows x 4 columns]
```

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)
```

```
      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')
```

```
      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'])
```

```
      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)
```

```
      label1 label2 label3
index1      "a      c      e
index2      b      d      f
[2 rows x 3 columns]
```


All of the dialect options can be specified separately by keyword arguments:

```
In [10]: data = 'a,b,c~1,2,3~4,5,6'
```

```
In [11]: pd.read_csv(StringIO(data), lineterminator='~')
```

```
   a  b  c
0  1  2  3
1  4  5  6
[2 rows x 3 columns]
```

Another common dialect option is `skipinitialspace`, to skip any whitespace after a delimiter:

```
In [12]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'
```

```
In [13]: print(data)
```

```
a, b, c
1, 2, 3
4, 5, 6
```

```
In [14]: pd.read_csv(StringIO(data), skipinitialspace=True)
```

```
   a  b  c
0  1  2  3
1  4  5  6
[2 rows x 3 columns]
```

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.

18.1.1 Specifying column data types

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

```
In [15]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'
```

```
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
```

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
[3 rows x 3 columns]
```

```
In [19]: df['a'][0]
'1'
```

```
In [20]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})
```

```
In [21]: df.dtypes
```

```
a      int64
b      object
c     float64
dtype: object
```

18.1.2 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\n1,2,3\n4,5,6\n7,8,9'
```

```
In [24]: print(data)
```

```
a,b,c
1,2,3
4,5,6
7,8,9
```

```
In [25]: pd.read_csv(StringIO(data))
```

```
   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)
```

```
   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)
```

```
   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\na,b,c\n1,2,3\n4,5,6\n7,8,9'
```

```
In [30]: pd.read_csv(StringIO(data), header=1)
```

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
[3 rows x 3 columns]
```

18.1.3 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\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'
```

```
In [32]: pd.read_csv(StringIO(data))
```

```
   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'])
```

```
   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])
```

```
   a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz
[3 rows x 3 columns]
```

18.1.4 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:

```
In [35]: data = 'word,length\nTr\u00e4umen,7\nGr\u00fc\u00dfe,5'
```

```
In [36]: df = pd.read_csv(StringIO(data), encoding='latin-1')
```

```
In [37]: df
```

```
   word  length
0  Tr\u00e4umen      7
1  Gr\u00fc\u00dfe      5
```

```
[2 rows x 2 columns]
```

```
In [38]: df['word'][1]
u'Gr\xfc\xdf'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won't parse correctly at all without specifying the encoding.

18.1.5 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:

```
In [39]: data = 'a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'
```

```
In [40]: pd.read_csv(StringIO(data))
```

```
      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\n4,apple,bat,5.7\n8,orange,cow,10'
```

```
In [42]: pd.read_csv(StringIO(data), index_col=0)
```

```
      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`:

```
In [43]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'
```

```
In [44]: print(data)
```

```
a,b,c
4,apple,bat,
8,orange,cow,
```

```
In [45]: pd.read_csv(StringIO(data))
```

```
      a      b      c
4  apple  bat  NaN
8  orange  cow  NaN
[2 rows x 3 columns]
```

```
In [46]: pd.read_csv(StringIO(data), index_col=False)
```

```
      a      b      c
0  4  apple  bat
1  8  orange  cow
[2 rows x 3 columns]
```

18.1.6 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

           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

<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
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
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 [51]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [52]: df

           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
```

```
      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
      3      4
0 18:56:00 0.81
1 19:56:00 0.01
2 20:56:00 -0.59
3 21:18:00 -0.99
4 21:56:00 -0.59
5 22:56:00 -0.59
[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
```

```
      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]
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The `index_col` specification is based off of this new set of columns rather than the original data columns:

```
In [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
```

```
      nominal      actual      0      4
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: When passing a dict as the `parse_dates` argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use `collections.OrderedDict` instead of a regular *dict* if this matters to you. Because of this, when using a dict for ‘`parse_dates`’ in conjunction with the `index_col` argument, it’s best to specify `index_col` as a column label rather than as an index on the resulting frame.

18.1.7 Date Parsing Functions

Finally, the parser allows you can specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```
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
```

```

          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.

18.1.8 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:

```
In [64]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c
```

```
In [65]: pd.read_csv('tmp.csv', parse_dates=[0])
```

```

          date  value cat
0 2000-01-06 00:00:00     5  a
1 2000-02-06 00:00:00    10  b

```

```
2 2000-03-06 00:00:00      15    c
[3 rows x 3 columns]
```

```
In [66]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
```

```
      date  value cat
0 2000-06-01 00:00:00     5  a
1 2000-06-02 00:00:00    10  b
2 2000-06-03 00:00:00    15  c
[3 rows x 3 columns]
```

18.1.9 Thousand Separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings

```
In [67]: print(open('tmp.csv').read())
```

```
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z
```

```
In [68]: df = pd.read_csv('tmp.csv', sep='|')
```

```
In [69]: df
```

```
      ID      level category
0  Patient1    123,000         x
1  Patient2     23,000         y
2  Patient3  1,234,018         z
[3 rows x 3 columns]
```

```
In [70]: df.level.dtype
dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly

```
In [71]: print(open('tmp.csv').read())
```

```
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z
```

```
In [72]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')
```

```
In [73]: df
```

```
      ID      level category
0  Patient1    123000         x
1  Patient2     23000         y
2  Patient3   1234018         z
[3 rows x 3 columns]
```

```
In [74]: df.level.dtype
dtype('int64')
```


18.1.10 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'].

```
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

18.1.11 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`.

18.1.12 Comments

Sometimes comments or meta data may be included in a file:

```
In [75]: 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 [76]: df = pd.read_csv('tmp.csv')
```

```
In [77]: df
```

```

      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 [78]: df = pd.read_csv('tmp.csv', comment='#')
```

```
In [79]: df
```

```
      ID    level category
0  Patient1  123000      x
1  Patient2   23000      y
2  Patient3 1234018      z
[3 rows x 3 columns]
```

18.1.13 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a `Series`:

```
In [80]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018
```

```
In [81]: output = pd.read_csv('tmp.csv', squeeze=True)
```

```
In [82]: output
```

```
Patient1    123000
Patient2     23000
Patient3    1234018
Name: level, dtype: int64
```

```
In [83]: type(output)
pandas.core.series.Series
```

18.1.14 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 [84]: data= 'a,b,c\n1,Yes,2\n3,No,4'
```

```
In [85]: print(data)
a,b,c
1,Yes,2
3,No,4
```

```
In [86]: pd.read_csv(StringIO(data))
```

```
   a    b    c
0  1  Yes    2
1  3   No    4
[2 rows x 3 columns]
```

```
In [87]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
```

```
   a      b    c
0  1   True    2
1  3  False    4
[2 rows x 3 columns]
```

18.1.15 Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many will cause an error by default:

```
In [27]: data = 'a,b,c\n1,2,3\n4,5,6,7\n8,9,10'
```

```
In [28]: pd.read_csv(StringIO(data))
```

```
-----
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
```

18.1.16 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 [88]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'
```

```
In [89]: print(data)
a,b
"hello, \"Bob\", nice to see you",5
```

```
In [90]: pd.read_csv(StringIO(data), escapechar='\\')
```

```

              a  b
0  hello, "Bob", nice to see you  5
[1 rows x 2 columns]
```

18.1.17 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 [91]: 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 [92]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [93]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [94]: df
```

	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 [95]: widths = [6, 14, 13, 10]

In [96]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [97]: df
```

	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 [98]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [99]: df
```

	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]

18.1.18 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

```
In [100]: 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 [101]: pd.read_csv('foo.csv')

      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 [102]: df = pd.read_csv('foo.csv', parse_dates=True)

In [103]: df.index

<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

18.1.19 Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

```
In [104]: print(open('data/minindex_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 [105]: df = pd.read_csv("data/minindex_ex.csv", index_col=[0,1])

In [106]: df
```

```
      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 [107]: df.ix[1978]
```

```
      zit  xit
indiv
A      0.2 0.06
B      0.7 0.20
C      0.8 0.30
D      0.9 0.50
E      1.4 0.90
[5 rows x 2 columns]
```

18.1.20 Reading columns with a MultiIndex

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the intervening rows. In order to have the pre-0.13 behavior of tupleizing columns, specify `tupleize_cols=True`.

```
In [108]: from pandas.util.testing import makeCustomDataframe as mkdf
```

```
In [109]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)
```

```
In [110]: df.to_csv('mi.csv')
```

```
In [111]: print(open('mi.csv').read())
```

```
C0,,C_10_g0,C_10_g1,C_10_g2
C1,,C_11_g0,C_11_g1,C_11_g2
C2,,C_12_g0,C_12_g1,C_12_g2
C3,,C_13_g0,C_13_g1,C_13_g2
R0,R1,,,
R_10_g0,R_11_g0,R0C0,R0C1,R0C2
R_10_g1,R_11_g1,R1C0,R1C1,R1C2
R_10_g2,R_11_g2,R2C0,R2C1,R2C2
R_10_g3,R_11_g3,R3C0,R3C1,R3C2
R_10_g4,R_11_g4,R4C0,R4C1,R4C2
```

```
In [112]: pd.read_csv('mi.csv',header=[0,1,2,3],index_col=[0,1])
```

```

C0          C_10_g0 C_10_g1 C_10_g2
C1          C_11_g0 C_11_g1 C_11_g2
C2          C_12_g0 C_12_g1 C_12_g2
C3          C_13_g0 C_13_g1 C_13_g2
R0      R1
R_10_g0 R_11_g0      R0C0      R0C1      R0C2
R_10_g1 R_11_g1      R1C0      R1C1      R1C2
R_10_g2 R_11_g2      R2C0      R2C1      R2C2
R_10_g3 R_11_g3      R3C0      R3C1      R3C2
R_10_g4 R_11_g4      R4C0      R4C1      R4C2
[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 [113]: 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 [114]: pd.read_csv('mi2.csv', header=[0,1], index_col=0)

```

```

      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*.

18.1.21 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 [115]: print(open('tmp2.csv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498

```

```

In [116]: pd.read_csv('tmp2.csv')

```

```

                                :0:1:2:3
0  0:0.4691122999071863:-0.2828633443286633:-1.50...
1  1:1.2121120250208506:-0.17321464905330858:0.11...
2  2:-0.8618489633477999:-2.1045692188948086:-0.4...
3  3:0.7215551622443669:-0.7067711336300845:-1.03...
4  4:-0.42497232978883753:0.567020349793672:0.276...

```

```
5 5:-0.6736897080883706:0.1136484096888855:-1.47...
6 6:0.4047052186802365:0.5770459859204836:-1.715...
7 7:-0.3706468582364464:-1.1578922506419993:-1.3...
8 8:1.0757697837155533:-0.10904997528022223:1.64...
9 9:0.35702056413309086:-0.6746001037299882:-1.7...
[10 rows x 1 columns]
```

18.1.22 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 [117]: print(open('tmp.csv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498
```

```
In [118]: table = pd.read_table('tmp.csv', sep='|')
```

```
In [119]: table
```

```
   Unnamed: 0      0      1      2      3
0      0  0.469112 -0.282863 -1.509059 -1.135632
1      1  1.212112 -0.173215  0.119209 -1.044236
2      2 -0.861849 -2.104569 -0.494929  1.071804
3      3  0.721555 -0.706771 -1.039575  0.271860
4      4 -0.424972  0.567020  0.276232 -1.087401
5      5 -0.673690  0.113648 -1.478427  0.524988
6      6  0.404705  0.577046 -1.715002 -1.039268
7      7 -0.370647 -1.157892 -1.344312  0.844885
8      8  1.075770 -0.109050  1.643563 -1.469388
9      9  0.357021 -0.674600 -1.776904 -0.968914
[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 [120]: reader = pd.read_table('tmp.csv', sep='|', chunksize=4)
```

```
In [121]: reader
<pandas.io.parsers.TextFileReader at 0xef4fd50>
```

```
In [122]: for chunk in reader:
```

```
.....:     print(chunk)
.....:
   Unnamed: 0      0      1      2      3
0      0  0.469112 -0.282863 -1.509059 -1.135632
1      1  1.212112 -0.173215  0.119209 -1.044236
2      2 -0.861849 -2.104569 -0.494929  1.071804
```



```

3          3  0.721555 -0.706771 -1.039575  0.271860
[4 rows x 5 columns]
   Unnamed: 0      0      1      2      3
0          4 -0.424972  0.567020  0.276232 -1.087401
1          5 -0.673690  0.113648 -1.478427  0.524988
2          6  0.404705  0.577046 -1.715002 -1.039268
3          7 -0.370647 -1.157892 -1.344312  0.844885
[4 rows x 5 columns]
   Unnamed: 0      0      1      2      3
0          8  1.075770 -0.10905  1.643563 -1.469388
1          9  0.357021 -0.67460 -1.776904 -0.968914
[2 rows x 5 columns]

```

Specifying `iterator=True` will also return the `TextFileReader` object:

```
In [123]: reader = pd.read_table('tmp.csv', sep='|', iterator=True)
```

```
In [124]: reader.get_chunk(5)
```

```

   Unnamed: 0      0      1      2      3
0          0  0.469112 -0.282863 -1.509059 -1.135632
1          1  1.212112 -0.173215  0.119209 -1.044236
2          2 -0.861849 -2.104569 -0.494929  1.071804
3          3  0.721555 -0.706771 -1.039575  0.271860
4          4 -0.424972  0.567020  0.276232 -1.087401
[5 rows x 5 columns]

```

18.1.23 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`

18.1.24 Writing a formatted string

The `DataFrame` object has an instance method `to_string` which allows control over the string representation of the object. All arguments are optional:

- `buf` default `None`, for example a `StringIO` object
- `columns` default `None`, which columns to write
- `col_space` default `None`, minimum width of each column.
- `na_rep` default `NaN`, representation of NA value
- `formatters` default `None`, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- `float_format` default `None`, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the `DataFrame`.
- `sparsify` default `True`, set to `False` for a `DataFrame` with a hierarchical index to print every multiindex key at each row.
- `index_names` default `True`, will print the names of the indices
- `index` default `True`, will print the index (ie, row labels)
- `header` default `True`, will print the column labels
- `justify` default `left`, will print column headers left- or right-justified

The `Series` object also has a `to_string` method, but with only the `buf`, `na_rep`, `float_format` arguments. There is also a `length` argument which, if set to `True`, will additionally output the length of the `Series`.

18.2 JSON

Read and write JSON format files and strings.

18.2.1 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

<code>split</code>	dict like <code>{index -> [index], columns -> [columns], data -> [values]}</code>
<code>records</code>	list like <code>[{column -> value}, ... , {column -> value}]</code>
<code>index</code>	dict like <code>{index -> {column -> value}}</code>
<code>columns</code>	dict like <code>{column -> {index -> value}}</code>
<code>values</code>	just the values array

- `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.

```
In [125]: dfj = DataFrame(randn(5, 2), columns=list('AB'))
```

```
In [126]: json = dfj.to_json()
```

```
In [127]: json
```

```
'{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.0061535699,"4":0.8957173022},"B":
```

Orient Options

There are a number of different options for the format of the resulting JSON file / string. Consider the following DataFrame and Series:

```
In [128]: dfjo = DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
.....:                      columns=list('ABC'), index=list('xyz'))
.....:
```

```
In [129]: dfjo
```

```
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9
[3 rows x 3 columns]
```

```
In [130]: sjo = Series(dict(x=15, y=16, z=17), name='D')
```

```
In [131]: sjo
```

```
x    15
y    16
z    17
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:

```
In [132]: dfjo.to_json(orient="columns")
'{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'
```

Index oriented (the default for Series) similar to column oriented but the index labels are now primary:

```
In [133]: dfjo.to_json(orient="index")
'{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'
```

```
In [134]: sjo.to_json(orient="index")
'{"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:

```
In [135]: dfjo.to_json(orient="records")
'[{ "A":1, "B":4, "C":7 }, { "A":2, "B":5, "C":8 }, { "A":3, "B":6, "C":9 } ]'
```

```
In [136]: sjo.to_json(orient="records")
' [15,16,17]'
```

Value oriented is a bare-bones option which serialises to nested JSON arrays of values only, column and index labels are not included:

```
In [137]: dfjo.to_json(orient="values")
'[[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`:

```
In [138]: dfjo.to_json(orient="split")
'{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,9]]}'
```

```
In [139]: sjo.to_json(orient="split")
'{"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

```
In [140]: dfd = DataFrame(randn(5, 2), columns=list('AB'))
```

```
In [141]: dfd['date'] = Timestamp('20130101')
```

```
In [142]: dfd = dfd.sort_index(1, ascending=False)
```

```
In [143]: json = dfd.to_json(date_format='iso')
```

```
In [144]: json
'{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":"2013-01-01T00:00:00.000Z"}}
```

Writing in iso date format, with microseconds

```
In [145]: json = dfd.to_json(date_format='iso', date_unit='us')
```

```
In [146]: json
'{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-01T00:00:00.000000Z","5":"2013-01-01T00:00:00.000000Z","6":"2013-01-01T00:00:00.000000Z","7":"2013-01-01T00:00:00.000000Z","8":"2013-01-01T00:00:00.000000Z","9":"2013-01-01T00:00:00.000000Z"},"id":1,"name":"2013-01-01T00:00:00.000000Z","type":"date","value":1}
```

Epoch timestamps, in seconds

```
In [147]: json = dfd.to_json(date_format='epoch', date_unit='s')
```

```
In [148]: json
'{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":1356998400},"B":{"0":2.565
```

Writing to a file, with a date index and a date column

- `filepath_or_buffer` : a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file `://localhost/path/to/table.json`
- `typ` : type of object to recover (series or frame), default 'frame'
- `orient` :

Series :

- default is `index`
- allowed values are `{split, records, index}`

DataFrame

- default is `columns`
- allowed values are `{split, records, index, columns, values}`

The format of the JSON string

<code>split</code>	dict like <code>{index -> [index], columns -> [columns], data -> [values]}</code>
<code>records</code>	list like <code>[{column -> value}, ... , {column -> value}]</code>
<code>index</code>	dict like <code>{index -> {column -> value}}</code>
<code>columns</code>	dict like <code>{column -> {index -> value}}</code>
<code>values</code>	just the values array

- `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 [159]: pd.read_json(json)
```

```

      A      B      date
0 -1.206412  2.565646 2013-01-01 00:00:00
1  1.431256  1.340309 2013-01-01 00:00:00
2 -1.170299 -0.226169 2013-01-01 00:00:00
3  0.410835  0.813850 2013-01-01 00:00:00
4  0.132003 -0.827317 2013-01-01 00:00:00
[5 rows x 3 columns]
```

Reading from a file:

```
In [160]: pd.read_json('test.json')
```

```

      A      B bools      date  ints
2013-01-01 -1.294524  0.413738  True 2013-01-01 00:00:00      0
2013-01-02  0.276662 -0.472035  True 2013-01-01 00:00:00      1
2013-01-03 -0.013960 -0.362543  True 2013-01-01 00:00:00      2
2013-01-04 -0.006154 -0.923061  True 2013-01-01 00:00:00      3
2013-01-05  0.895717  0.805244  True 2013-01-01 00:00:00      4
[5 rows x 5 columns]
```

Don't convert any data (but still convert axes and dates):

```
In [161]: pd.read_json('test.json', dtype=object).dtypes
```

```

A              object
B              object
bools          object
date    datetime64[ns]
ints              object
dtype: object
```

Specify dtypes for conversion:

```
In [162]: pd.read_json('test.json', dtype={'A' : 'float32', 'bools' : 'int8'}).dtypes
```

```

A              float32
B              float64
bools          int8
date    datetime64[ns]
ints              int64
dtype: object
```

Preserve string indicies:

```

In [163]: si = DataFrame(np.zeros((4, 4)),
.....:                  columns=list(range(4)),
.....:                  index=[str(i) for i in range(4)])
.....:
```

```
In [164]: si
```

```
   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 [165]: si.index
```

```
Index([u'0', u'1', u'2', u'3'], dtype='object')
```

```
In [166]: si.columns
```

```
Int64Index([0, 1, 2, 3], dtype='int64')
```

```
In [167]: json = si.to_json()
```

```
In [168]: sij = pd.read_json(json, convert_axes=False)
```

```
In [169]: sij
```

```
   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 [170]: sij.index
```

```
Index([u'0', u'1', u'2', u'3'], dtype='object')
```

```
In [171]: sij.columns
```

```
Index([u'0', u'1', u'2', u'3'], dtype='object')
```

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [172]: json = dfj2.to_json(date_unit='ns')
```

```
# Try to parse timestamps as milliseconds -> Won't Work
```

```
In [173]: dfju = pd.read_json(json, date_unit='ms')
```

```
In [174]: dfju
```

```
           A           B bools           date  ints
1356998400000000000 -1.294524  0.413738  True  1356998400000000000    0
1357084800000000000  0.276662 -0.472035  True  1356998400000000000    1
1357171200000000000 -0.013960 -0.362543  True  1356998400000000000    2
1357257600000000000 -0.006154 -0.923061  True  1356998400000000000    3
1357344000000000000  0.895717  0.805244  True  1356998400000000000    4
[5 rows x 5 columns]
```

```
# Let Pandas detect the correct precision
```

```
In [175]: dfju = pd.read_json(json)
```

```
In [176]: dfju
```

```
           A           B bools           date  ints
2013-01-01 -1.294524  0.413738  True  2013-01-01 00:00:00    0
```



```

2013-01-02  0.276662 -0.472035  True 2013-01-01 00:00:00    1
2013-01-03 -0.013960 -0.362543  True 2013-01-01 00:00:00    2
2013-01-04 -0.006154 -0.923061  True 2013-01-01 00:00:00    3
2013-01-05  0.895717  0.805244  True 2013-01-01 00:00:00    4
[5 rows x 5 columns]

```

Or specify that all timestamps are in nanoseconds

```
In [177]: dfju = pd.read_json(json, date_unit='ns')
```

```
In [178]: dfju
```

```

           A           B bools           date  ints
2013-01-01 -1.294524  0.413738  True 2013-01-01 00:00:00    0
2013-01-02  0.276662 -0.472035  True 2013-01-01 00:00:00    1
2013-01-03 -0.013960 -0.362543  True 2013-01-01 00:00:00    2
2013-01-04 -0.006154 -0.923061  True 2013-01-01 00:00:00    3
2013-01-05  0.895717  0.805244  True 2013-01-01 00:00:00    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 [179]: randfloats = np.random.uniform(-100, 1000, 10000)
```

```
In [180]: randfloats.shape = (1000, 10)
```

```
In [181]: dffloats = DataFrame(randfloats, columns=list('ABCDEFGHIJ'))
```

```
In [182]: jsonfloats = dffloats.to_json()
```

```
In [183]: timeit read_json(jsonfloats)
100 loops, best of 3: 11 ms per loop
```

```
In [184]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 5.88 ms per loop
```

The speedup is less noticable for smaller datasets:

```
In [185]: jsonfloats = dffloats.head(100).to_json()
```

```
In [186]: timeit read_json(jsonfloats)
100 loops, best of 3: 4.14 ms per loop
```

```
In [187]: timeit read_json(jsonfloats, numpy=True)
100 loops, best of 3: 3.09 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.

18.2.3 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 [188]: from pandas.io.json import json_normalize
```

```
In [189]: 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 [190]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor']])
```

	name	population	info.governor	state	shortname
0	Dade	12345	Rick Scott	Florida	FL
1	Broward	40000	Rick Scott	Florida	FL
2	Palm Beach	60000	Rick Scott	Florida	FL
3	Summit	1234	John Kasich	Ohio	OH
4	Cuyahoga	1337	John Kasich	Ohio	OH

[5 rows x 5 columns]

18.3 HTML

18.3.1 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 [191]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'
```

```
In [192]: dfs = read_html(url)
```

```
In [193]: dfs
```

```
[
      Bank Name      City  ST  \
0      Syringa Bank      Boise  ID
1      The Bank of Union      El Reno  OK
2      DuPage National Bank      West Chicago  IL
3      Texas Community Bank, National Association      The Woodlands  TX
4      Bank of Jackson County      Graceville  FL
5      First National Bank also operating as The Nat...      Edinburg  TX
6      The Community's Bank      Bridgeport  CT
7      Sunrise Bank of Arizona      Phoenix  AZ
8      Community South Bank      Parsons  TN
9      Bank of Wausau      Wausau  WI
10     First Community Bank of Southwest Florida (als...      Fort Myers  FL
11     Mountain National Bank      Sevierville  TN
12     1st Commerce Bank      North Las Vegas  NV
13     Banks of Wisconsin d/b/a Bank of Kenosha      Kenosha  WI
14     Central Arizona Bank      Scottsdale  AZ
      ...
      CERT      Acquiring Institution      Closing Date  \
0      34296      Sunwest Bank      2014-01-31 00:00:00
1      17967      BancFirst      2014-01-24 00:00:00
2      5732      Republic Bank of Chicago      2014-01-17 00:00:00
3      57431      Spirit of Texas Bank, SSB      2013-12-13 00:00:00
4      14794      First Federal Bank of Florida      2013-10-30 00:00:00
5      14318      PlainsCapital Bank      2013-09-13 00:00:00
6      57041      No Acquirer      2013-09-13 00:00:00
7      34707      First Fidelity Bank, National Association      2013-08-23 00:00:00
8      19849      CB&S Bank, Inc.      2013-08-23 00:00:00
9      35016      Nicolet National Bank      2013-08-09 00:00:00
10     34943      C1 Bank      2013-08-02 00:00:00
11     34789      First Tennessee Bank, National Association      2013-06-07 00:00:00
12     58358      Plaza Bank      2013-06-06 00:00:00
13     35386      North Shore Bank, FSB      2013-05-31 00:00:00
14     34527      Western State Bank      2013-05-14 00:00:00
      ...
      Updated Date  Loss Share Type
0      2014-01-31 00:00:00      NaN
1      2014-01-28 00:00:00      NaN
2      2014-01-27 00:00:00      NaN
3      2014-01-13 00:00:00      NaN
4      2013-12-09 00:00:00      none
5      2013-11-01 00:00:00      SFR/NSF
6      2013-12-20 00:00:00      none
7      2013-11-01 00:00:00      none
8      2013-11-01 00:00:00      none
9      2013-10-24 00:00:00      none
10     2013-10-24 00:00:00      none
```

```
11 2013-07-12 00:00:00      none
12 2013-07-12 00:00:00      NSF
13 2013-10-29 00:00:00      none
14 2013-07-12 00:00:00      none
    ...
[519 rows x 8 columns]]
```

Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string

```
In [194]: with open(file_path, 'r') as f:
.....:     dfs = read_html(f.read())
.....:
```

```
In [195]: dfs
```

```
[
      Bank Name      City  ST  CERT  \
0  Banks of Wisconsin d/b/a Bank of Kenosha  Kenosha  WI  35386
1      Central Arizona Bank  Scottsdale  AZ  34527
2      Sunrise Bank  Valdosta  GA  58185
3  Pisgah Community Bank  Asheville  NC  58701
4  Douglas County Bank  Douglasville  GA  21649
5      Parkway Bank  Lenoir  NC  57158
6  Chipola Community Bank  Marianna  FL  58034
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  \
0  North Shore Bank, FSB  2013-05-31 00:00:00
1  Western State Bank  2013-05-14 00:00:00
2  Synovus Bank  2013-05-10 00:00:00
3  Capital Bank, N.A.  2013-05-10 00:00:00
4  Hamilton State Bank  2013-04-26 00:00:00
5  CertusBank, National Association  2013-04-26 00:00:00
6  First Federal Bank of Florida  2013-04-19 00:00:00
7  FirstAtlantic Bank  2013-04-19 00:00:00
8  Your Community Bank  2013-04-19 00:00:00
9  First Scottsdale Bank, National Association  2013-04-05 00:00:00
10  HeritageBank of the South  2013-03-08 00:00:00
11  Liberty Bank and Trust Company  2013-02-15 00:00:00
12  First Minnesota Bank  2013-01-18 00:00:00
13  Sunwest Bank  2013-01-11 00:00:00
14  Bank of Sullivan  2012-12-14 00:00:00
    ...
      Updated Date
0  2013-05-31 00:00:00
1  2013-05-20 00:00:00
2  2013-05-21 00:00:00
3  2013-05-14 00:00:00
```

```

4 2013-05-16 00:00:00
5 2013-05-17 00:00:00
6 2013-05-16 00:00:00
7 2013-05-16 00:00:00
8 2013-04-23 00:00:00
9 2013-04-09 00:00:00
10 2013-03-26 00:00:00
11 2013-03-04 00:00:00
12 2013-02-28 00:00:00
13 2013-01-24 00:00:00
14 2013-01-24 00:00:00
...
[506 rows x 7 columns]

```

You can even pass in an instance of StringIO if you so desire

```
In [196]: from cStringIO import StringIO
```

```
In [197]: with open(file_path, 'r') as f:
.....:     sio = StringIO(f.read())
.....:
```

```
In [198]: dfs = read_html(sio)
```

```
In [199]: dfs
```

```

[
      Bank Name      City  ST  CERT \
0  Banks of Wisconsin d/b/a Bank of Kenosha  Kenosha  WI  35386
1      Central Arizona Bank  Scottsdale  AZ  34527
2      Sunrise Bank  Valdosta  GA  58185
3  Pisgah Community Bank  Asheville  NC  58701
4  Douglas County Bank  Douglasville  GA  21649
5      Parkway Bank  Lenoir  NC  57158
6  Chipola Community Bank  Marianna  FL  58034
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 \
0  North Shore Bank, FSB 2013-05-31 00:00:00
1  Western State Bank 2013-05-14 00:00:00
2  Synovus Bank 2013-05-10 00:00:00
3  Capital Bank, N.A. 2013-05-10 00:00:00
4  Hamilton State Bank 2013-04-26 00:00:00
5  CertusBank, National Association 2013-04-26 00:00:00
6  First Federal Bank of Florida 2013-04-19 00:00:00
7  FirstAtlantic Bank 2013-04-19 00:00:00
8  Your Community Bank 2013-04-19 00:00:00
9  First Scottsdale Bank, National Association 2013-04-05 00:00:00
10  HeritageBank of the South 2013-03-08 00:00:00
11  Liberty Bank and Trust Company 2013-02-15 00:00:00
12  First Minnesota Bank 2013-01-18 00:00:00
13  Sunwest Bank 2013-01-11 00:00:00

```

```
14                                     Bank of Sullivan 2012-12-14 00:00:00
                                     ...
Updated Date
0  2013-05-31 00:00:00
1  2013-05-20 00:00:00
2  2013-05-21 00:00:00
3  2013-05-14 00:00:00
4  2013-05-16 00:00:00
5  2013-05-17 00:00:00
6  2013-05-16 00:00:00
7  2013-05-16 00:00:00
8  2013-04-23 00:00:00
9  2013-04-09 00:00:00
10 2013-03-26 00:00:00
11 2013-03-04 00:00:00
12 2013-02-28 00:00:00
13 2013-01-24 00:00:00
14 2013-01-24 00:00:00
                                     ...
[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'])
```

18.3.2 Writing to HTML files

`DataFrame` objects have an instance method `to_html` which renders the contents of the `DataFrame` as an HTML table. The function arguments are as in the method `to_string` described above.

Note: Not all of the possible options for `DataFrame.to_html` are shown here for brevity's sake. See `to_html()` for the full set of options.

```
In [200]: df = DataFrame(randn(2, 2))
```

```
In [201]: df
```

```
      0      1
0 -0.184744  0.496971
1 -0.856240  1.857977
[2 rows x 2 columns]
```

```
In [202]: 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 [203]: `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 [204]: `print(df.to_html(float_format='{0:.10f}'.format))`

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.1847438576</td>
      <td>0.4969711327</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off

In [205]: `print(df.to_html(bold_rows=False))`

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>
      <td>-0.184744</td>
      <td> 0.496971</td>
    </tr>
    <tr>
      <td>1</td>
      <td>-0.856240</td>
      <td> 1.857977</td>
    </tr>
  </tbody>
</table>
```

The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing 'dataframe' class.

In [206]: `print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class']))`

```
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td> 0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td> 1.857977</td>
    </tr>
  </tbody>
</table>
```

Finally, the `escape` argument allows you to control whether the "<", ">" and "&" characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

In [207]: `df = DataFrame({'a': list('&<>'), 'b': randn(3)})`

Escaped:

```
In [208]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td> &#x26; </td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td> &#x26;lt; </td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td> &#x26;gt; </td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```
In [209]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td> &#x26; </td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td> &#x26;lt; </td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td> &#x26;gt; </td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Note: Some browsers may not show a difference in the rendering of the previous two HTML tables.

18.4 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

Note: The prior method of accessing Excel is now deprecated as of 0.12.0, this will work but will be removed in a future version.

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

Replaced by

```
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

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:

```
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```
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.

```
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.

```
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:

```
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`.

```
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.

18.4.1 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`.

```
# 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')
```

18.5 Clipboard

A handy way to grab data is to use the `read_clipboard` method, which takes the contents of the clipboard buffer and passes them to the `read_table` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a `DataFrame` by calling:

```
clipdf = pd.read_clipboard()
```

```
In [210]: clippdf
```

```

      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 [211]: df=pd.DataFrame(randn(5,3))
```

```
In [212]: df
```

```

      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 [213]: df.to_clipboard()
```

```
In [214]: pd.read_clipboard()
```

```

      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]
```

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.

18.6 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 [215]: df
```

```

      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 [216]: 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 [217]: read_pickle('foo.pkl')
```

```
      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]
```

Warning: Loading pickled data received from untrusted sources can be unsafe.
See: <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](#) for a detailed explanation.

Note: These methods were previously `pd.save` and `pd.load`, prior to 0.12.0, and are now deprecated.

18.7 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 [218]: df = DataFrame(np.random.rand(5,2), columns=list('AB'))
```

```
In [219]: df.to_msgpack('foo.msg')
```

```
In [220]: pd.read_msgpack('foo.msg')
```

```
      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]
```

```
In [221]: s = Series(np.random.rand(5), index=date_range('20130101', periods=5))
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [222]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1,2,3]), s)
```

```
In [223]: pd.read_msgpack('foo.msg')
```

```
[
      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 [224]: for o in pd.read_msgpack('foo.msg', iterator=True):
.....:     print o
.....:
```

```
      A      B
0  0.154336  0.710999
1  0.398096  0.765220
2  0.586749  0.293052
3  0.290293  0.710783
4  0.988593  0.062106
[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 [225]: df.to_msgpack('foo.msg', append=True)
```

```
In [226]: pd.read_msgpack('foo.msg')
```

```
[
      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,
      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]]
```

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 [227]: pd.to_msgpack('foo2.msg', { 'dict' : [ { 'df' : df }, { 'string' : 'foo' }, { 'scalar' : 1 }
```

```
In [228]: pd.read_msgpack('foo2.msg')
```

```
{u'dict': ({u'df':
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': 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}}}
```

18.7.1 Read/Write API

Msgpacks can also be read from and written to strings.

```
In [229]: df.to_msgpack()
'\x84\xa6blocks\x91\x86\xa5items\x85\xa5dtype\x11\xa3typ\xa5index\xa5klass\xa5Index\xa4data\x92\xa1A'
```

Furthermore you can concatenate the strings to produce a list of the original objects.

```
In [230]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
```

```
[
      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]
```

18.8 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 $\geq 0.12.0$ is required for compatibility.

```
In [231]: store = HDFStore('store.h5')
```

```
In [232]: 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 [233]: np.random.seed(1234)
```

```
In [234]: index = date_range('1/1/2000', periods=8)
```

```
In [235]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [236]: df = DataFrame(randn(8, 3), index=index,
.....:                  columns=['A', 'B', 'C'])
.....:
```

```
In [237]: 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 [238]: store['s'] = s
```

```
In [239]: store['df'] = df
```

```
In [240]: store['wp'] = wp
```

```
# the type of stored data
```

```
In [241]: store.root.wp._v_attrs.pandas_type
'wide'
```

```
In [242]: store
```

```
<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 [243]: store['df']
```

```
      A      B      C  
2000-01-01  0.887163  0.859588 -0.636524  
2000-01-02  0.015696 -2.242685  1.150036  
2000-01-03  0.991946  0.953324 -2.021255  
2000-01-04 -0.334077  0.002118  0.405453  
2000-01-05  0.289092  1.321158 -1.546906  
2000-01-06 -0.202646 -0.655969  0.193421  
2000-01-07  0.553439  1.318152 -0.469305  
2000-01-08  0.675554 -1.817027 -0.183109  
[8 rows x 3 columns]
```

```
# dotted (attribute) access provides get as well  
In [244]: store.df
```

```
      A      B      C  
2000-01-01  0.887163  0.859588 -0.636524  
2000-01-02  0.015696 -2.242685  1.150036  
2000-01-03  0.991946  0.953324 -2.021255  
2000-01-04 -0.334077  0.002118  0.405453  
2000-01-05  0.289092  1.321158 -1.546906  
2000-01-06 -0.202646 -0.655969  0.193421  
2000-01-07  0.553439  1.318152 -0.469305  
2000-01-08  0.675554 -1.817027 -0.183109  
[8 rows x 3 columns]
```

Deletion of the object specified by the key

```
# store.remove('wp') is an equivalent method  
In [245]: del store['wp']
```

```
In [246]: store
```

```
<class 'pandas.io.pytables.HDFStore'>  
File path: store.h5  
/df      frame      (shape->[8,3])  
/s       series     (shape->[5])
```

Closing a Store, Context Manager

```
In [247]: store.close()
```

```
In [248]: store
```

```
<class 'pandas.io.pytables.HDFStore'>  
File path: store.h5  
File is CLOSED
```

```
In [249]: store.is_open  
False
```

```
# Working with, and automatically closing the store with the context  
# manager
```

```
In [250]: with get_store('store.h5') as store:  
.....:     store.keys()  
.....:
```

18.8.1 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)

```
In [251]: df_t1 = DataFrame(dict(A=list(range(5)), B=list(range(5))))
```

```
In [252]: df_t1.to_hdf('store_t1.h5', 'table', append=True)
```

```
In [253]: read_hdf('store_t1.h5', 'table', where = ['index>2'])
```

```
   A  B
3  3  3
4  4  4
[2 rows x 2 columns]
```

18.8.2 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 **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`.

```
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
```

18.8.3 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.

```
In [254]: store = HDFStore('store.h5')
```

```
In [255]: df1 = df[0:4]
```

```
In [256]: df2 = df[4:]
```

```
# append data (creates a table automatically)
```

```
In [257]: store.append('df', df1)
```

```
In [258]: store.append('df', df2)

In [259]: store

<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 [260]: store.select('df')

           A          B          C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109
[8 rows x 3 columns]

# the type of stored data
In [261]: store.root.df._v_attrs.pandas_type
'frame_table'
```

Note: You can also create a table by passing `format='table'` or `format='t'` to a put operation.

18.8.4 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 with out 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 [262]: store.put('foo/bar/bah', df)

In [263]: store.append('food/orange', df)

In [264]: store.append('food/apple', df)

In [265]: store

<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 [266]: store.keys()
['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
```

```
In [267]: store.remove('food')
```

```
In [268]: store
```

```
<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])
```

18.8.5 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 [269]: 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 [270]: df_mixed.ix[3:5,['A', 'B', 'string', 'datetime64']] = np.nan
```

```
In [271]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})
```

```
In [272]: df_mixed1 = store.select('df_mixed')
```

```
In [273]: df_mixed1
```

```
      A      B      C  bool  datetime64  int  string
0  0.704721 -1.152659 -0.430096  True  2001-01-02 00:00:00    1  string
1 -0.785435  0.631979  0.767369  True  2001-01-02 00:00:00    1  string
2  0.462060  0.039513  0.984920  True  2001-01-02 00:00:00    1  string
3      NaN      NaN  0.270836  True                NaT    1      NaN
4      NaN      NaN  1.391986  True                NaT    1      NaN
5      NaN      NaN  0.079842  True                NaT    1      NaN
6  2.007843  0.152631 -0.399965  True  2001-01-02 00:00:00    1  string
7  0.226963  0.164530 -1.027851  True  2001-01-02 00:00:00    1  string
[8 rows x 7 columns]
```

```
In [274]: df_mixed1.get_dtype_counts()
```

```
bool          1
datetime64[ns] 1
float32       1
float64       2
int64         1
object        1
dtype: int64
```

```
# we have provided a minimum string column size
In [275]: store.root.df_mixed.table

/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}
```

18.8.6 Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.

```
In [276]: 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 [277]: df_mi = DataFrame(np.random.randn(10, 3), index=index,
.....:                      columns=['A', 'B', 'C'])
.....:
```

```
In [278]: df_mi
```

		A	B	C
foo	bar			
foo	one	-0.584718	0.816594	-0.081947
	two	-0.344766	0.528288	-1.068989
	three	-0.511881	0.291205	0.566534
bar	one	0.503592	0.285296	0.484288
	two	1.363482	-0.781105	-0.468018
baz	two	1.224574	-1.281108	0.875476
	three	-1.710715	-0.450765	0.749164
qux	one	-0.203933	-0.182175	0.680656
	two	-1.818499	0.047072	0.394844
	three	-0.248432	-0.617707	-0.682884

[10 rows x 3 columns]

```
In [279]: store.append('df_mi', df_mi)
```

```
In [280]: store.select('df_mi')
```

		A	B	C
foo	bar			
foo	one	-0.584718	0.816594	-0.081947

```

      two    -0.344766    0.528288   -1.068989
      three -0.511881    0.291205    0.566534
bar one     0.503592    0.285296    0.484288
      two     1.363482   -0.781105   -0.468018
baz two     1.224574   -1.281108    0.875476
      three -1.710715   -0.450765    0.749164
qux one     -0.203933   -0.182175    0.680656
      two     -1.818499    0.047072    0.394844
      three -0.248432   -0.617707   -0.682884
[10 rows x 3 columns]

# the levels are automatically included as data columns
In [281]: store.select('df_mi', 'foo=bar')

      A      B      C
foo bar
bar one  0.503592  0.285296  0.484288
      two  1.363482 -0.781105 -0.468018
[2 rows x 3 columns]

```

18.8.7 Querying a Table

Warning: This query capabilities have changed substantially starting in 0.13.0. Queries from prior version are accepted (with a `DeprecationWarning`) printed if its not string-like.

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

A query is specified using the `Term` class under the hood, as a boolean expression.

- `index` and `columns` are supported indexers of a `DataFrame`
- `major_axis`, `minor_axis`, and `items` are supported indexers of the `Panel`
- if `data_columns` are specified, these can be used as additional indexers

Valid comparison operators are:

- `=`, `==`, `!=`, `>`, `>=`, `<`, `<=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `( and )` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

Note:

- `=` will be automatically expanded to the comparison operator `==`
- `~` is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

```
• 'index>=date'
• "columns=['A', 'D']"
• "columns in ['A', 'D']"
• 'columns=A'
• 'columns==A'
• "~(columns=['A', 'B'])"
• 'index>df.index[3] & string="bar"'
• '(index>df.index[3] & index<=df.index[6]) | string="bar"'
• "ts>=Timestamp('2012-02-01')"
```

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

```
In [283]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [284]: store.select('dfq', "index>Timestamp('20130104') & columns=['A', 'B']")
```

```
          A          B
2013-01-05  1.210384  0.797435
2013-01-06 -0.850346  1.176812
2013-01-07  0.984188 -0.121728
2013-01-08  0.796595 -0.474021
2013-01-09 -0.804834 -2.123620
2013-01-10  0.334198  0.536784
[6 rows x 2 columns]
```

Use and inline column reference

```
In [285]: store.select('dfq', where="A>0 or C>0")
```

```
          A          B          C          D
2013-01-01  0.436258 -1.703013  0.393711 -0.479324
2013-01-02 -0.299016  0.694103  0.678630  0.239556
2013-01-03  0.151227  0.816127  1.893534  0.639633
2013-01-04 -0.962029 -2.085266  1.930247 -1.735349
2013-01-05  1.210384  0.797435 -0.379811  0.702562
2013-01-07  0.984188 -0.121728  2.365769  0.496143
```



```
2013-01-08  0.796595 -0.474021 -0.056696  1.357797
2013-01-10  0.334198  0.536784 -0.743830 -0.320204
[8 rows x 4 columns]
```

Works with a Panel as well.

```
In [286]: store.append('wp', wp)
```

```
In [287]: store
```

```
<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->
/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]
/wp                wide_table   (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_
/foo/bar/bah       frame        (shape->[8,3])
```

```
In [288]: store.select('wp', "major_axis>Timestamp('20000102') & minor_axis=['A', 'B']")
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B
```

The `columns` keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a `'columns=list_of_columns_to_filter'`:

```
In [289]: store.select('df', "columns=['A', 'B']")
```

```
          A          B
2000-01-01  0.887163  0.859588
2000-01-02  0.015696 -2.242685
2000-01-03  0.991946  0.953324
2000-01-04 -0.334077  0.002118
2000-01-05  0.289092  1.321158
2000-01-06 -0.202646 -0.655969
2000-01-07  0.553439  1.318152
2000-01-08  0.675554 -1.817027
[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 [290]: wp.to_frame()
```

```
          Item1  Item2
major  minor
2000-01-01 A    1.058969  0.215269
          B   -0.397840  0.841009
          C    0.337438 -1.445810
          D    1.047579 -1.401973
2000-01-02 A    1.045938 -0.100918
          B    0.863717 -0.548242
          C   -0.122092 -0.144620
          D    0.124713  0.354020
```

```
2000-01-03 A      -0.322795 -0.035513
           B       0.841675  0.565738
           C       2.390961  1.545659
           D       0.076200 -0.974236
2000-01-04 A      -0.566446 -0.070345
           B       0.036142  0.307969
           C      -2.074978 -0.208499
           ...         ...
[20 rows x 2 columns]

# limiting the search
In [291]: store.select('wp', "major_axis>20000102 & minor_axis=['A','B']",
.....:                start=0, stop=10)
.....:

<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 [292]: from datetime import timedelta
```

```
In [293]: dftd = DataFrame(dict(A = Timestamp('20130101'), B = [ Timestamp('20130101') + timedelta(d
```

```
In [294]: dftd['C'] = dftd['A']-dftd['B']
```

```
In [295]: dftd
```

```
           A                               B                               C
0 2013-01-01 00:00:00 2013-01-01 00:00:10          -00:00:10
1 2013-01-01 00:00:00 2013-01-02 00:00:10 -1 days, 00:00:10
2 2013-01-01 00:00:00 2013-01-03 00:00:10 -2 days, 00:00:10
3 2013-01-01 00:00:00 2013-01-04 00:00:10 -3 days, 00:00:10
4 2013-01-01 00:00:00 2013-01-05 00:00:10 -4 days, 00:00:10
5 2013-01-01 00:00:00 2013-01-06 00:00:10 -5 days, 00:00:10
6 2013-01-01 00:00:00 2013-01-07 00:00:10 -6 days, 00:00:10
7 2013-01-01 00:00:00 2013-01-08 00:00:10 -7 days, 00:00:10
8 2013-01-01 00:00:00 2013-01-09 00:00:10 -8 days, 00:00:10
9 2013-01-01 00:00:00 2013-01-10 00:00:10 -9 days, 00:00:10
[10 rows x 3 columns]
```

```
In [296]: store.append('dftd', dftd, data_columns=True)
```

```
In [297]: store.select('dftd', "C<' -3.5D' ")
```

```

      A      B      C
4 2013-01-01 00:00:00 2013-01-05 00:00:10 -4 days, 00:00:10
5 2013-01-01 00:00:00 2013-01-06 00:00:10 -5 days, 00:00:10
6 2013-01-01 00:00:00 2013-01-07 00:00:10 -6 days, 00:00:10
7 2013-01-01 00:00:00 2013-01-08 00:00:10 -7 days, 00:00:10
8 2013-01-01 00:00:00 2013-01-09 00:00:10 -8 days, 00:00:10
9 2013-01-01 00:00:00 2013-01-10 00:00:10 -9 days, 00:00:10
[6 rows x 3 columns]

```

18.8.8 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 automatically 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 [298]: i = store.root.df.table.cols.index.index
```

```
In [299]: i.optlevel, i.kind
(6, 'medium')
```

```
# change an index by passing new parameters
```

```
In [300]: store.create_table_index('df', optlevel=9, kind='full')
```

```
In [301]: i = store.root.df.table.cols.index.index
```

```
In [302]: i.optlevel, i.kind
(9, 'full')
```

See [here](#) for how to create a completely-sorted-index (CSI) on an existing store.

18.8.9 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 [303]: df_dc = df.copy()
```

```
In [304]: df_dc['string'] = 'foo'
```

```
In [305]: df_dc.ix[4:6, 'string'] = np.nan
```

```
In [306]: df_dc.ix[7:9, 'string'] = 'bar'
```

```
In [307]: df_dc['string2'] = 'cool'
```

```
In [308]: df_dc.ix[1:3, ['B', 'C']] = 1.0
```

```
In [309]: df_dc
```

	A	B	C	string	string2
2000-01-01	0.887163	0.859588	-0.636524	foo	cool
2000-01-02	0.015696	1.000000	1.000000	foo	cool
2000-01-03	0.991946	1.000000	1.000000	foo	cool
2000-01-04	-0.334077	0.002118	0.405453	foo	cool
2000-01-05	0.289092	1.321158	-1.546906	NaN	cool
2000-01-06	-0.202646	-0.655969	0.193421	NaN	cool
2000-01-07	0.553439	1.318152	-0.469305	foo	cool
2000-01-08	0.675554	-1.817027	-0.183109	bar	cool

[8 rows x 5 columns]

```
# on-disk operations
```

```
In [310]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])
```

```
In [311]: store.select('df_dc', [ Term('B>0') ])
```

	A	B	C	string	string2
2000-01-01	0.887163	0.859588	-0.636524	foo	cool
2000-01-02	0.015696	1.000000	1.000000	foo	cool
2000-01-03	0.991946	1.000000	1.000000	foo	cool
2000-01-04	-0.334077	0.002118	0.405453	foo	cool
2000-01-05	0.289092	1.321158	-1.546906	NaN	cool
2000-01-07	0.553439	1.318152	-0.469305	foo	cool

[6 rows x 5 columns]

```
# getting creative
```

```
In [312]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
```

	A	B	C	string	string2
2000-01-02	0.015696	1.000000	1.000000	foo	cool
2000-01-03	0.991946	1.000000	1.000000	foo	cool
2000-01-04	-0.334077	0.002118	0.405453	foo	cool

[3 rows x 5 columns]

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

```
In [313]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
```

	A	B	C	string	string2
2000-01-02	0.015696	1.000000	1.000000	foo	cool
2000-01-03	0.991946	1.000000	1.000000	foo	cool
2000-01-04	-0.334077	0.002118	0.405453	foo	cool

[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 [314]: store.root.df_dc.table
```

```
/df_dc/table (Table(8,)) ''
description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2),
  "C": Float64Col(shape=(), dflt=0.0, pos=3),
  "string": StringCol(itemsize=3, shape=(), dflt='', pos=4),
  "string2": StringCol(itemsize=4, shape=(), dflt='', pos=5)}
byteorder := 'little'
```

```
chunkshape := (1680,)
autoindex := True
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

There is some performance degradation by making lots of columns into *data columns*, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!)

18.8.10 Iterator

Starting in 0.11.0, you can pass, `iterator=True` or `chunksize=number_in_a_chunk` to select and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```
In [315]: 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]
          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]
          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.

```
for df in read_hdf('store.h5', 'df', chunksize=3):
    print(df)
```

Note, that the `chunksize` keyword applies to the **returned** rows. So if you are doing a query, then that set will be subdivided and returned in the iterator. Keep in mind that if you do not pass a `where` selection criteria then the `nrows` of the table are considered.

18.8.11 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 [316]: store.select_column('df_dc', 'index')
```

```
0    2000-01-01 00:00:00
1    2000-01-02 00:00:00
2    2000-01-03 00:00:00
3    2000-01-04 00:00:00
4    2000-01-05 00:00:00
5    2000-01-06 00:00:00
6    2000-01-07 00:00:00
7    2000-01-08 00:00:00
dtype: datetime64[ns]
```

```
In [317]: store.select_column('df_dc', 'string')
```

```
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
dtype: object
```

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent where operations.

```
In [318]: df_coord = DataFrame(np.random.randn(1000,2),index=date_range('20000101',periods=1000))
```

```
In [319]: store.append('df_coord',df_coord)
```

```
In [320]: c = store.select_as_coordinates('df_coord','index>20020101')
```

```
In [321]: c.summary()
u'Int64Index: 268 entries, 732 to 999'
```

```
In [322]: store.select('df_coord',where=c)
```

```
          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 [323]: df_mask = DataFrame(np.random.randn(1000,2), index=date_range('20000101', periods=1000))

In [324]: store.append('df_mask', df_mask)

In [325]: c = store.select_column('df_mask', 'index')

In [326]: where = c[DatetimeIndex(c).month==5].index

In [327]: store.select('df_mask', where=where)
```

```
           0          1
2000-05-01 -0.098554 -0.280782
2000-05-02  0.739851  1.627182
2000-05-03  0.030132 -0.145601
2000-05-04  0.227530  1.048856
2000-05-05  1.773939  1.116887
2000-05-06  1.081251  1.509416
2000-05-07 -0.498694 -0.913155
2000-05-08 -0.126945  0.079686
2000-05-09  1.020345 -0.790110
2000-05-10 -1.155447 -0.367505
2000-05-11  1.263914  0.827049
2000-05-12 -0.572469  1.211678
2000-05-13  1.792988  1.379125
2000-05-14  2.049917  0.992011
2000-05-15 -0.930035 -0.249394
...         ...
[93 rows x 2 columns]
```

Storer Object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [328]: store.get_storer('df_dc').nrows
8
```

18.8.12 Multiple Table Queries

New in 0.10.1 are the methods `append_to_multiple` and `select_as_multiple`, that can perform append-ing/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 [329]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
.....:                      columns=['A', 'B', 'C', 'D', 'E', 'F'])
.....:
```

```
In [330]: df_mt['foo'] = 'bar'
```

```
In [331]: df_mt.ix[1, ('A', 'B')] = np.nan
```

you can also create the tables individually

```
In [332]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None },
.....:                             df_mt, selector='df1_mt')
.....:
```

```
In [333]: store
```

```
<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,st
/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->
/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
/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,mino
/foo/bar/bah       frame        (shape->[8,3])
```

individual tables were created

```
In [334]: store.select('df1_mt')
```

```
          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 [335]: store.select('df2_mt')
```

```
          C          D          E          F  foo
2000-01-01 -1.521825 -0.428670 -1.550209  0.826839  bar
2000-01-03  1.236974 -1.328279  0.662291  1.894976  bar
2000-01-04 -0.746478  0.851039  1.415686 -0.929096  bar
2000-01-05 -1.452287  1.575492 -0.197377 -0.219901  bar
2000-01-06  1.190342  2.115021  0.148762  1.073931  bar
2000-01-07 -0.709897 -2.022441  0.714697  0.318215  bar
2000-01-08  0.852225  0.096696 -0.379903  0.929313  bar
[7 rows x 5 columns]
```



```
# as a multiple
In [336]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
.....:                               selector = 'df1_mt')
.....:

           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  0.096696 -0.379903  0.929313  bar
[2 rows x 7 columns]
```

18.8.13 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.

```
# returns the number of rows deleted
In [337]: store.remove('wp', 'major_axis>20000102' )
12

In [338]: store.select('wp')

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

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).

18.8.14 Compression

PyTables allows the stored data to be compressed. This applies to all kinds of stores, not just tables.

- Pass `complevel=int` for a compression level (1-9, with 0 being no compression, and the default)
- Pass `complib=lib` where `lib` is any of `zlib`, `bzip2`, `lzo`, `blosc` for whichever compression library you prefer.

HDFStore will use the file based compression scheme if no overriding `complib` or `complevel` options are provided. `blosc` offers very fast compression, and is my most used. Note that `lzo` and `bzip2` may not be installed (by Python) by default.

Compression for all objects within the file

- ```
store_compressed = HDFStore('store_compressed.h5', complevel=9, complib='blosc')
```

Or on-the-fly compression (this only applies to tables). You can turn off file compression for a specific table by passing `complevel=0`

- ```
store.append('df', df, complib='zlib', complevel=5)
```

ptrepack

PyTables offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied PyTables utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

- ```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

### 18.8.15 Notes & Caveats

- Once a table is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
- If a row has `np.nan` for **EVERY COLUMN** (having a `nan` in a string, or a `NaT` in a datetime-like column counts as having a value), then those rows **WILL BE DROPPED IMPLICITLY**. This limitation *may* be addressed in the future.
- HDFStore is **not-threadsafe for writing**. The underlying PyTables only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the issue (:2397) for more information.
- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- PyTables only supports fixed-width string columns in tables. The sizes of a string based indexing column (e.g. `columns` or `minor_axis`) are determined as the maximum size of the elements in that axis or by passing the parameter

**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.

### 18.8.16 DataTypes

HDFStore will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

- floating: float64, float32, float16 (*using np.nan to represent invalid values*)
- integer: int64, int32, int8, uint64, uint32, uint8
- bool
- datetime64[ns] (*using NaT to represent invalid values*)
- object: strings (*using np.nan to represent invalid values*)

Currently, unicode and datetime columns (represented with a dtype of object), **WILL FAIL**. In addition, even though a column may look like a datetime64[ns], if it contains np.nan, this **WILL FAIL**. You can try to convert datetimelike columns to proper datetime64[ns] columns, that possibly contain NaT to represent invalid values. (Some of these issues have been addressed and these conversion may not be necessary in future versions of pandas)

```
In [339]: import datetime

In [340]: df = DataFrame(dict(datelike=Series([datetime.datetime(2001, 1, 1),
.....: datetime.datetime(2001, 1, 2), np.nan])))
.....:

In [341]: df

 datelike
0 2001-01-01 00:00:00
1 2001-01-02 00:00:00
2 NaN
[3 rows x 1 columns]

In [342]: df.dtypes

datelike object
dtype: object

to convert
In [343]: df['datelike'] = Series(df['datelike'].values, dtype='M8[ns]')

In [344]: df

 datelike
0 2001-01-01 00:00:00
1 2001-01-02 00:00:00
2 NaT
[3 rows x 1 columns]

In [345]: df.dtypes

datelike datetime64[ns]
dtype: object
```

### 18.8.17 String Columns

**min\_itemsize**

The underlying implementation of `HDFStore` uses a fixed column width (`itemsizes`) 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

---

```
In [346]: dfs = DataFrame(dict(A = 'foo', B = 'bar'), index=list(range(5)))
```

```
In [347]: dfs
```

```
 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 [348]: store.append('dfs', dfs, min_itemsize = 30)
```

```
In [349]: store.get_storer('dfs').table
```

```
/dfs/table (Table(5,)) ''
description := {
 "index": Int64Col(shape=(), dflt=0, pos=0),
 "values_block_0": StringCol(itemsize=30, shape=(2,), dflt='', pos=1)}
byteorder := 'little'
chunkshape := (963,)
autoindex := True
colindexes := {
 "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

```
A is created as a data_column with a size of 30
```

```
B is size is calculated
```

```
In [350]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })
```

```
In [351]: store.get_storer('dfs2').table
```

```
/dfs2/table (Table(5,)) ''
description := {
 "index": Int64Col(shape=(), dflt=0, pos=0),
 "values_block_0": StringCol(itemsize=3, shape=(1,), dflt='', pos=1),
 "A": StringCol(itemsize=30, shape=(), dflt='', pos=2)}
byteorder := 'little'
chunkshape := (1598,)
autoindex := True
```

```
colindexes := {
 "A": Index(6, medium, shuffle, zlib(1)).is_csi=False,
 "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

### nan\_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [352]: dfss = DataFrame(dict(A = ['foo', 'bar', 'nan']))
```

```
In [353]: dfss
```

```
 A
0 foo
1 bar
2 nan
[3 rows x 1 columns]
```

```
In [354]: store.append('dfss', dfss)
```

```
In [355]: store.select('dfss')
```

```
 A
0 foo
1 bar
2 NaN
[3 rows x 1 columns]
```

*# here you need to specify a different nan\_rep*

```
In [356]: store.append('dfss2', dfss, nan_rep='_nan_')
```

```
In [357]: store.select('dfss2')
```

```
 A
0 foo
1 bar
2 nan
[3 rows x 1 columns]
```

## 18.8.18 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 [358]: store_export = HDFStore('export.h5')
```

```
In [359]: store_export.append('df_dc', df_dc, data_columns=df_dc.columns)
```

```
In [360]: store_export
```

```
<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,s
```

### 18.8.19 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 updates. The group attribute `pandas_version` contains the version information. `copy` takes a number of options, please see the docstring.

```
a legacy store
In [361]: legacy_store = HDFStore(legacy_file_path, 'r')

In [362]: legacy_store

<class 'pandas.io.pytables.HDFStore'>
File path: /home/docbuild/CI/pandas/doc/source/_static/legacy_0.10.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/df1_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 ndim_table [0.10.0] (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis,minor_axis])
/foo/bar wide (shape->[3,30,4])

copy (and return the new handle)
In [363]: new_store = legacy_store.copy('store_new.h5')

In [364]: new_store

<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/df1_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])
/foo/bar wide (shape->[3,30,4])

In [365]: new_store.close()
```

### 18.8.20 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 expect. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by `PyTables` (rather than stored as endemic types). See [Here](#) for more information and some solutions.

## 18.8.21 Experimental

HDFStore supports Panel4D storage.

```
In [366]: p4d = Panel4D({ 'l1' : wp })
```

```
In [367]: p4d
```

```
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 1 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: l1 to l1
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

```
In [368]: store.append('p4d', p4d)
```

```
In [369]: store
```

```
<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,st
/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->
/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
/dfs frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index])
/dfs2 frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])
/dfss frame_table (typ->appendable,nrows->3,ncols->1,indexers->[index])
/dfss2 frame_table (typ->appendable,nrows->3,ncols->1,indexers->[index])
/dftd frame_table (typ->appendable,nrows->10,ncols->3,indexers->[index],dc->[A,B,C
/p4d wide_table (typ->appendable,nrows->40,ncols->1,indexers->[items,major_axis
/wp wide_table (typ->appendable,nrows->8,ncols->2,indexers->[major_axis,mino
/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 [370]: store.append('p4d2', p4d, axes=['labels', 'major_axis', 'minor_axis'])
```

```
In [371]: store
```

```
<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,st
/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->
/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
```

```

/dfs frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index])
/dfs2 frame_table (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])
/dfss frame_table (typ->appendable,nrows->3,ncols->1,indexers->[index])
/dfss2 frame_table (typ->appendable,nrows->3,ncols->1,indexers->[index])
/dftd frame_table (typ->appendable,nrows->10,ncols->3,indexers->[index],dc->[A,B,C])
/p4d wide_table (typ->appendable,nrows->40,ncols->1,indexers->[items,major_axis])
/p4d2 wide_table (typ->appendable,nrows->20,ncols->2,indexers->[labels,major_axis])
/wp wide_table (typ->appendable,nrows->8,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah frame (shape->[8,3])

```

```
In [372]: store.select('p4d2', [Term('labels=l1'), Term('items=Item1'), Term('minor_axis=A_big_string')])
```

```

<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

```

## 18.9 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:

| id | Date       | Col_1 | Col_2 | Col_3 |
|----|------------|-------|-------|-------|
| 26 | 2012-10-18 | X     | 25.7  | True  |
| 42 | 2012-10-19 | Y     | -12.4 | False |
| 63 | 2012-10-20 | Z     | 5.73  | True  |

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:

```

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_frame()` function to get the query results into a `DataFrame`:

```
In [373]: sql.read_frame("SELECT * FROM data;", cnx)
```

```

 id date Col_1 Col_2 Col_3
0 26 2010-10-18 00:00:00 X 27.50 1
1 42 2010-10-19 00:00:00 Y -12.50 0
2 63 2010-10-20 00:00:00 Z 5.73 1
[3 rows x 5 columns]

```

You can also specify the name of the column as the `DataFrame` index:

```
In [374]: sql.read_frame("SELECT * FROM data;", cnx, index_col='id')
```

```

 date Col_1 Col_2 Col_3
id

```



```

26 2010-10-18 00:00:00 X 27.50 1
42 2010-10-19 00:00:00 Y -12.50 0
63 2010-10-20 00:00:00 Z 5.73 1
[3 rows x 4 columns]

```

```
In [375]: sql.read_frame("SELECT * FROM data;", cnx, index_col='date')
```

```

 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 [376]: sql.read_frame("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", cnx)
```

```

 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**.

---

## 18.10 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:

```
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*:

```
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:

```
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/big-query>> for details.

As of 10/10/13, there is a bug in Google's API preventing result sets from being larger than 100,000 rows. A patch is scheduled for the week of 10/14/13.

## 18.11 STATA Format

New in version 0.12.0.

### 18.11.1 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 [377]: df = DataFrame(randn(10, 2), columns=list('AB'))
```

```
In [378]: df.to_stata('stata.dta')
```

### 18.11.2 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 [379]: pd.read_stata('stata.dta')
```

```
 index A B
0 0 0.811031 -0.356817
1 1 1.047085 0.664705
2 2 -0.086919 0.416905
3 3 -0.764381 -0.287229
4 4 -0.089351 -1.035115
5 5 0.489131 -0.253340
6 6 -1.948100 -0.116556
7 7 0.800597 -0.796154
8 8 -0.382952 -0.397373
9 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 wheter 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



# REMOTE DATA ACCESS

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.

## 19.1 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']
```

|           |             |
|-----------|-------------|
| 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

## 19.2 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']
```

|        |          |
|--------|----------|
| Open   | 10.17    |
| High   | 10.28    |
| Low    | 10.05    |
| Close  | 10.28    |
| Volume | 60855796 |

Name: 2010-01-04 00:00:00, dtype: object

## 19.3 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']
```

|     |         |
|-----|---------|
| 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()
```

|            | CPIAUCSL | CPILFESL |
|------------|----------|----------|
| DATE       |          |          |
| 2010-01-01 | 217.478  | 220.544  |
| 2010-02-01 | 217.356  | 220.668  |
| 2010-03-01 | 217.380  | 220.749  |
| 2010-04-01 | 217.281  | 220.808  |
| 2010-05-01 | 217.230  | 221.027  |

[5 rows x 2 columns]

## 19.4 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", "famafrch")
```

```
In [23]: ip[4].ix[192607]
```

```
1 Cnsmr 5.43
2 Manuf 2.73
3 HiTec 1.83
4 Hlth 1.64
5 Other 2.15
Name: 192607, dtype: float64
```

## 19.5 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 | 43228.111147107  |
|               | 2005 | 42516.3934699993 |

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())
```

```
OLS Regression Results
```

|                   |                  |                     |         |
|-------------------|------------------|---------------------|---------|
| Dep. Variable:    | cellphone        | R-squared:          | 0.297   |
| Model:            | OLS              | Adj. R-squared:     | 0.274   |
| Method:           | Least Squares    | F-statistic:        | 13.08   |
| Date:             | Thu, 25 Jul 2013 | Prob (F-statistic): | 0.00105 |
| Time:             | 15:24:42         | Log-Likelihood:     | -139.16 |
| No. Observations: | 33               | AIC:                | 282.3   |
| Df Residuals:     | 31               | BIC:                | 285.3   |
| Df Model:         | 1                |                     |         |

```
=====
```

|             | coef    | std err | t     | P> t  | [95.0% Conf. Int.] |
|-------------|---------|---------|-------|-------|--------------------|
| Intercept   | 16.5110 | 19.071  | 0.866 | 0.393 | -22.384 55.406     |
| np.log(gdp) | 9.9333  | 2.747   | 3.616 | 0.001 | 4.331 15.535       |

```
=====
```

|                |        |                   |          |
|----------------|--------|-------------------|----------|
| Omnibus:       | 36.054 | Durbin-Watson:    | 2.071    |
| Prob(Omnibus): | 0.000  | Jarque-Bera (JB): | 119.133  |
| Skew:          | -2.314 | Prob(JB):         | 1.35e-26 |
| Kurtosis:      | 11.077 | Cond. No.         | 45.8     |



=====



# ENHANCING PERFORMANCE

## 20.1 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.

### 20.1.1 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' * 1000})
```

```
In [2]: df
```

```
 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
7 650 -1.044236 -1.118384 x
8 389 -0.861849 -0.542980 x
9 772 -2.104569 -0.994002 x
10 174 -0.494929 1.508742 x
11 394 1.071804 -0.328697 x
12 199 0.721555 -0.562235 x
13 318 -0.706771 0.001596 x
14 281 -1.039575 0.077546 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 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: 348 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)
650647 function calls in 0.526 seconds
Ordered by: internal time
List reduced from 110 to 4 due to restriction <4>
ncalls tottime percall cumtime percall filename:lineno(function)
 1000 0.230 0.000 0.353 0.000 <ipython-input-4-a877a66f40a5>:1(integrate_f)
 552423 0.117 0.000 0.117 0.000 <ipython-input-3-0b27f90c4c8a>:1(f)
 1004 0.020 0.000 0.020 0.000 {method 'reduce' of 'numpy.ufunc' objects}
 2001 0.012 0.000 0.012 0.000 {pandas.lib.infer_dtype}
```

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.

---

## 20.1.2 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)
1 loops, best of 3: 232 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

### 20.1.3 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: 125 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)
97224 function calls in 0.182 seconds
Ordered by: internal time
List reduced from 108 to 4 due to restriction <4>
ncalls tottime percall cumtime percall filename:lineno(function)
 1004 0.022 0.000 0.022 0.000 {method 'reduce' of 'numpy.ufunc' objects}
 2001 0.012 0.000 0.012 0.000 {pandas.lib.infer_dtype}
 1001 0.011 0.000 0.056 0.000 internals.py:1864(make_block)
 1001 0.010 0.000 0.113 0.000 series.py:124(__init__)
```

### 20.1.4 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 internally 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: 2.74 ms per loop
```

We've gone another three times faster! 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 0.004 seconds
Ordered by: internal time
List reduced from 13 to 4 due to restriction <4>
ncalls tottime percall cumtime percall filename:lineno(function)
1 0.004 0.004 0.004 0.004 {_cython_magic_e90d139b668b225d575329a3043b5155.apply_...
1 0.000 0.000 0.004 0.004 <string>:1(<module>)
3 0.000 0.000 0.000 0.000 frame.py:1612(__getitem__)
3 0.000 0.000 0.000 0.000 frame.py:1637(__getitem__column)
```

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.

## 20.1.5 More advanced techniques

There is still scope for improvement, here's an example of using some more advanced cython techniques:

```

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)
...: cpdef double integrate_f_typed(double a, double b, int N):
...: cdef int i
...: cdef double s, dx
...: s = 0
...: dx = (b - a) / N
...: for i in range(N):
...: s += f_typed(a + i * dx)
...: return s * dx
...: @cython.boundscheck(False)
...: @cython.wraparound(False)
...: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a, np.ndarray[double]
...: cdef Py_ssize_t i, n = len(col_a)
...: 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: 2.33 ms per loop

```

This shaves another third off!

## 20.1.6 Further topics

- Loading C modules into cython.

Read more in the [cython docs](#).

## 20.2 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()`.

---

## 20.2.1 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`.

## 20.2.2 `eval()` Examples

`eval()` works wonders for expressions containing large arrays

First let's create 4 decent-sized arrays to play with:

```
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()`:

```
In [24]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 62.6 ms per loop
```

```
In [25]: %timeit pd.eval('df1 + df2 + df3 + df4')
100 loops, best of 3: 19.8 ms per loop
```

Now let's do the same thing but with comparisons:

```
In [26]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
10 loops, best of 3: 46.4 ms per loop
```

```
In [27]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
10 loops, best of 3: 16.8 ms per loop
```

`eval()` also works with unaligned pandas objects:

```
In [28]: s = Series(randn(50))
```

```
In [29]: %timeit df1 + df2 + df3 + df4 + s
10 loops, best of 3: 93.3 ms per loop
```

```
In [30]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
10 loops, best of 3: 20.7 ms per loop
```

---

**Note:** Operations such as

```
1 and 2 # would parse to 1 & 2, but should evaluate to 2
3 or 4 # would parse to 3 | 4, but should evaluate to 3
~1 # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type `bool` or `np.bool_`. Again, you should perform these kinds of operations in plain Python.

---

### 20.2.3 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')
```

```
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 you're interested in evaluating*.

In addition, you can perform in-line assignment of columns within an expression. This can allow for *formulaic evaluation*. Only a single assignment is permitted. It can be a new column name or an existing column name. It must be a string-like.

```
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
```

```
 a b c d
0 1 5 5 10
1 1 6 7 14
2 1 7 9 18
3 1 8 11 22
4 1 9 13 26
[5 rows x 4 columns]
```

## 20.2.4 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')
```

```
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')
```

```
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')
```

```
 a b
1 -1.215949 1.670837
3 -0.835893 0.315213
4 -1.337334 0.772364
[3 rows x 2 columns]
```

---

### 20.2.5 `eval()` Parsers

There are two different parsers and two different engines you can use as the backend.

The default 'pandas' parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the `&` and `|` operators is made equal to the precedence of the corresponding boolean operations `and` and `or`.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the 'python' parser to enforce strict Python semantics.

```
In [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 & df2 > 0 & df3 > 0 & df4 > 0'
```

```
In [46]: y = pd.eval(expr_no_parens, parser='pandas')
```

```
In [47]: np.all(x == y)
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_and = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'
```

```
In [51]: y = pd.eval(expr_with_and, parser='pandas')
```

```
In [52]: np.all(x == y)
True
```

The `and` and `or` operators here have the same precedence that they would in vanilla Python.

### 20.2.6 `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'`.

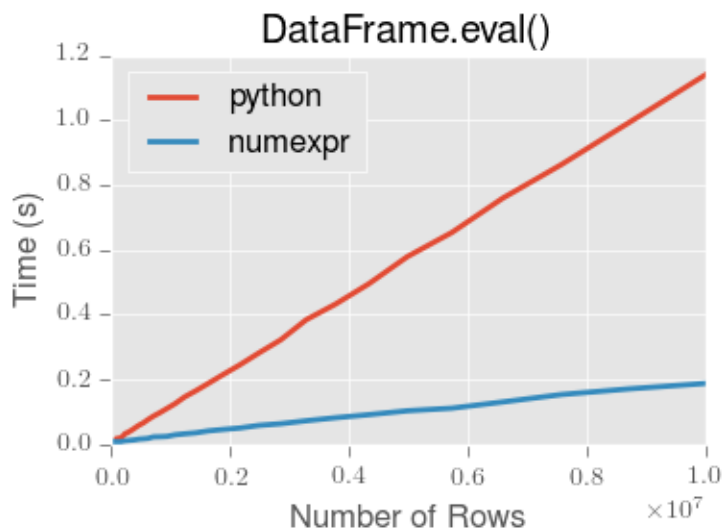
You can see this by using `eval()` with the 'python' engine is actually a bit slower (not by much) than evaluating the same expression in Python:

```
In [53]: %timeit df1 + df2 + df3 + df4
10 loops, best of 3: 60.2 ms per loop
```

```
In [54]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
10 loops, best of 3: 60.9 ms per loop
```

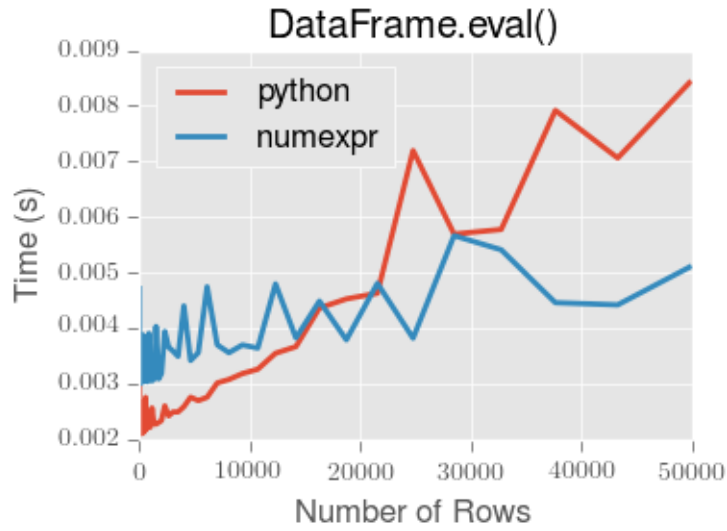
## 20.2.7 eval() Performance

`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()`.



# SPARSE DATA STRUCTURES

We have implemented “sparse” versions of Series, DataFrame, and Panel. These are not sparse in the typical “mostly 0”. You can view these objects as being “compressed” where any data matching a specific value (NaN/missing by default, though any value can be chosen) is omitted. A special `SparseIndex` object tracks where data has been “sparsified”. This will make much more sense in an example. All of the standard pandas data structures have a `to_sparse` method:

```
In [1]: ts = Series(randn(10))
```

```
In [2]: ts[2:-2] = np.nan
```

```
In [3]: sts = ts.to_sparse()
```

```
In [4]: sts
```

```
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`:

```
In [5]: ts.fillna(0).to_sparse(fill_value=0)
```

```
0 0.469112
1 -0.282863
2 0.000000
3 0.000000
4 0.000000
5 0.000000
6 0.000000
7 0.000000
8 -0.861849
9 -2.104569
```

```
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

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
```

```
 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
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()
```

```
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
```



## 21.1 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 [13]: arr[2:5] = np.nan; arr[7:8] = np.nan

In [14]: sparr = SparseArray(arr)

In [15]: sparr

[-1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.606027190513, 1.3342]
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()

array([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
 nan, 0.606 , 1.3342])
```

## 21.2 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

<pandas.sparse.list.SparseList object at 0x15582050>
```

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

<pandas.sparse.list.SparseList object at 0x15582050>
[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, 1.15893288864]
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()

[1.0, nan, nan, 2.0, 3.0, 5.0, -1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, 1.15893288864, 0.606027190513, 1.15893288864]
Fill: nan
IntIndex
Indices: array([0, 3, 4, 5, 6, 7, 11, 12, 14, 15], dtype=int32)
```

## 21.3 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 array 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.

# CAVEATS AND GOTCHAS

## 22.1 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

```
>>> 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`:

```
>>> 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`

```
>>> if pd.Series([False, True, False]) is not None:
 print("I was not None")
>>> I was not None
```

or return if any value is `True`.

```
>>> if pd.Series([False, True, False]).any():
 print("I am any")
>>> I am any
```

To evaluate single-element pandas objects in a boolean context, use the method `.bool()`:

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

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

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

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

### 22.1.1 Bitwise boolean

Bitwise boolean operators like `==` and `!=` will return a boolean `Series`, which is almost always what you want anyways.

```
>>> s = pd.Series(range(5))
>>> s == 4
0 False
1 False
2 False
3 False
4 True
dtype: bool
```

See [boolean comparisons](#) for more examples.

## 22.2 NaN, Integer NA values and NA type promotions

### 22.2.1 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.

### 22.2.2 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:

```
In [5]: s = Series([1, 2, 3, 4, 5], index=list('abcde'))
```

```
In [6]: s
```

```
a 1
b 2
c 3
d 4
e 5
dtype: int64
```

```
In [7]: s.dtype
dtype('int64')
```

```
In [8]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])
```

```
In [9]: s2
```

```
a 1
b 2
c 3
f NaN
u NaN
dtype: float64
```

```
In [10]: s2.dtype
dtype('float64')
```

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.

### 22.2.3 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:

| Typeclass | Promotion dtype for storing NAs |
|-----------|---------------------------------|
| floating  | no change                       |
| object    | no change                       |
| integer   | cast to float64                 |
| boolean   | cast to object                  |

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.

### 22.2.4 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:

| Typeclass                          | Dtypes                                                                                     |
|------------------------------------|--------------------------------------------------------------------------------------------|
| <code>numpy.floating</code>        | <code>float16</code> , <code>float32</code> , <code>float64</code> , <code>float128</code> |
| <code>numpy.integer</code>         | <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>           |
| <code>numpy.unsignedinteger</code> | <code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code>       |
| <code>numpy.object_</code>         | <code>object_</code>                                                                       |
| <code>numpy.bool_</code>           | <code>bool_</code>                                                                         |
| <code>numpy.character</code>       | <code>string_</code> , <code>unicode_</code>                                               |

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.

## 22.3 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:

```
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).

## 22.4 Label-based slicing conventions

### 22.4.1 Non-monotonic indexes require exact matches

### 22.4.2 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:

```
In [11]: s = Series(randn(6), index=list('abcdef'))
```

```
In [12]: s
```

```
a 1.337122
b -1.531095
c 1.331458
d -0.571329
e -0.026671
f -1.085663
dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be

```
In [13]: s[2:5]
```

```
c 1.331458
d -0.571329
e -0.026671
dtype: float64
```

However, if you only had `c` and `e`, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.ix['c':'e'+1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design design to make label-based slicing include both endpoints:

```
In [14]: s.ix['c':'e']
```

```
c 1.331458
d -0.571329
e -0.026671
dtype: float64
```

This is most definitely a “practicality beats purity” sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

## 22.5 Miscellaneous indexing gotchas

### 22.5.1 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
```

```
 one two three four
a -1.114738 -0.058216 -0.486768 1.685148
b 0.112572 -1.495309 0.898435 -0.148217
c -1.596070 0.159653 0.262136 0.036220
d 0.184735 -0.255069 -0.271020 1.288393
e 0.294633 -1.165787 0.846974 -0.685597
f 0.609099 -0.303961 0.625555 -0.059268
[6 rows x 4 columns]
```

```
In [17]: df.ix[['b', 'c', 'e']]
```

```
 one two three four
b 0.112572 -1.495309 0.898435 -0.148217
c -1.596070 0.159653 0.262136 0.036220
e 0.294633 -1.165787 0.846974 -0.685597
[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'])
```

```
 one two three four
b 0.112572 -1.495309 0.898435 -0.148217
c -1.596070 0.159653 0.262136 0.036220
e 0.294633 -1.165787 0.846974 -0.685597
[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]]
```

```
 one two three four
```

```
b 0.112572 -1.495309 0.898435 -0.148217
c -1.596070 0.159653 0.262136 0.036220
e 0.294633 -1.165787 0.846974 -0.685597
[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])
```

```
 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
```

```
a 1
0 2
1 3
dtype: int64
```

```
In [23]: s.ix[[0, 1]]
```

```
0 2
1 3
dtype: int64
```

```
In [24]: s.reindex([0, 1])
```

```
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.

## 22.5.2 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.



## 22.6 Timestamp limitations

### 22.6.1 Minimum and maximum timestamps

Since pandas represents timestamps in nanosecond resolution, the timespan that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [25]: begin = Timestamp.min
```

```
In [26]: begin
Timestamp('1677-09-22 00:12:43.145225', tz=None)
```

```
In [27]: end = Timestamp.max
```

```
In [28]: end
Timestamp('2262-04-11 23:47:16.854775807', tz=None)
```

If you need to represent time series data outside the nanosecond timespan, use `PeriodIndex`:

```
In [29]: span = period_range('1215-01-01', '1381-01-01', freq='D')
```

```
In [30]: span
```

```
<class 'pandas.tseries.period.PeriodIndex'>
freq: D
[1215-01-01, ..., 1381-01-01]
length: 60632
```

## 22.7 Parsing Dates from Text Files

When parsing multiple text file columns into a single date column, the new date column is prepended to the data and then `index_col` specification is indexed off of the new set of columns rather than the original ones:

```
In [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
```

```

 actual 0 1 2 3 \
nominal
```

```
1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 19990127 19:00:00 18:56:00
1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 19990127 20:00:00 19:56:00
1999-01-27 21:00:00 1999-01-27 20:56:00 KORD 19990127 21:00:00 20:56:00
1999-01-27 21:00:00 1999-01-27 21:18:00 KORD 19990127 21:00:00 21:18:00
1999-01-27 22:00:00 1999-01-27 21:56:00 KORD 19990127 22:00:00 21:56:00
1999-01-27 23:00:00 1999-01-27 22:56:00 KORD 19990127 23:00:00 22:56:00
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]
```

## 22.8 Differences with NumPy

For Series and DataFrame objects, `var` normalizes by  $N-1$  to produce unbiased estimates of the sample variance, while NumPy's `var` normalizes by  $N$ , which measures the variance of the sample. Note that `cov` normalizes by  $N-1$  in both pandas and NumPy.

## 22.9 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the `DataFrame.copy` method. If you are doing a lot of copying of DataFrame objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See [this link](#) for more information.

## 22.10 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 `read_html`.

### Issues with `lxml`

- Benefits
  - `lxml` is very fast
  - `lxml` requires Cython to install correctly.
- Drawbacks
  - `lxml` does *not* make any guarantees about the results of its parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the `lxml` backend, but **this backend will use `html5lib` if `lxml` fails to parse**
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if `lxml` fails.

### Issues with **BeautifulSoup4** using `lxml` as a backend

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

#### Issues with **BeautifulSoup4** using **html5lib** as a backend

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** generates valid HTML5 markup from invalid markup automatically. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is “correct”, since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.
- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the url over the web, i.e., IO (input-output). For very large tables, this might not be true.

#### Issues with using **Anaconda**

- **Anaconda** ships with **lxml** version 3.2.0; the following workaround for **Anaconda** was successfully used to deal with the versioning issues surrounding **lxml** and **BeautifulSoup4**.

---

**Note:** Unless you have *both*:

- A strong restriction on the upper bound of the runtime of some code that incorporates `read_html()`
- Complete knowledge that the HTML you will be parsing will be 100% valid at all times

then you should install **html5lib** and things will work swimmingly without you having to muck around with *conda*. If you want the best of both worlds then install both **html5lib** and **lxml**. If you do install **lxml** then you need to perform the following commands to ensure that **lxml** will work correctly:

```
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 'bzz+lp:beautifulsoup4'
```

Note that you need **bzr** and **git** installed to perform the last two operations.

---

## 22.11 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.

## RPY2 / R INTERFACE

---

**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.

### 23.1 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()
```

|   | education | age | parity | induced | case | spontaneous | stratum | pooled.stratum |
|---|-----------|-----|--------|---------|------|-------------|---------|----------------|
| 1 | 0-5yrs    | 26  | 6      | 1       | 1    | 2           | 1       | 3              |
| 2 | 0-5yrs    | 42  | 1      | 1       | 1    | 0           | 2       | 1              |
| 3 | 0-5yrs    | 39  | 6      | 2       | 1    | 0           | 3       | 4              |

```
4 0-5yrs 34 4 2 1 0 4 2
5 6-11yrs 35 3 1 1 1 5 32
[5 rows x 8 columns]
```

## 23.2 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**):

```
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.robjjects.vectors.DataFrame'>
```

```
In [8]: print(r_dataframe)
 A B C
one 1 4 7
two 2 5 8
three 3 6 9
```

The DataFrame's index is stored as the `rownames` attribute of the `data.frame` instance.

You can also use **convert\_to\_r\_matrix** to obtain a `Matrix` instance, but bear in mind that it will only work with homogeneously-typed DataFrames (as R matrices bear no information on the data type):

```
In [9]: r_matrix = com.convert_to_r_matrix(df)
```

```
In [10]: print(type(r_matrix))
<class 'rpy2.robjjects.vectors.Matrix'>
```

```
In [11]: print(r_matrix)
 A B C
one 1 4 7
two 2 5 8
three 3 6 9
```

## 23.3 Calling R functions with pandas objects

## 23.4 High-level interface to R estimators

# RELATED PYTHON LIBRARIES

## 24.1 `la` (larry)

Keith Goodman's excellent [labeled array package](#) is very similar to pandas in many regards, though with some key differences. The main philosophical design difference is to be a wrapper around a single NumPy `ndarray` object while adding axis labeling and label-based operations and indexing. Because of this, creating a size-mutable object with heterogeneous columns (e.g. `DataFrame`) is not possible with the `la` package.

- Provide a single n-dimensional object with labeled axes with functionally analogous data alignment semantics to pandas objects
- Advanced / label-based indexing similar to that provided in pandas but setting is not supported
- Stays much closer to NumPy arrays than pandas—`larry` objects must be homogeneously typed
- `GroupBy` support is relatively limited, but a few functions are available: `group_mean`, `group_median`, and `group_ranking`
- It has a collection of analytical functions suited to quantitative portfolio construction for financial applications
- It has a collection of moving window statistics implemented in [Bottleneck](#)

## 24.2 `statsmodels`

The main [statistics and econometrics library](#) for Python. pandas has become a dependency of this library.

## 24.3 `scikits.timeseries`

`scikits.timeseries` provides a data structure for fixed frequency time series data based on the `numpy.MaskedArray` class. For time series data, it provides some of the same functionality to the pandas `Series` class. It has many more functions for time series-specific manipulation. Also, it has support for many more frequencies, though less customizable by the user (so 5-minutely data is easier to do with pandas for example).

We are aiming to merge these libraries together in the near future.

Progress:

- It has a collection of moving window statistics implemented in [Bottleneck](#)
- [Outstanding issues](#)

Summarising, Pandas offers superior functionality due to its combination with the `pandas.DataFrame`.  
An introduction for former users of `scikits.timeseries` is provided in the *[migration guide](#)*.



# 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.

## 25.1 Base R

### 25.1.1 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:

```
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 [1]: from pandas import DataFrame
```

```
In [2]: from numpy import random
```

```
In [3]: df = DataFrame({'a': random.randn(10), 'b': random.randn(10)})
```

```
In [4]: df.query('a <= b')
```

```
 a b
0 0.026864 0.731942
1 -2.071188 -0.714342
5 -0.550596 -0.391094
[3 rows x 2 columns]
```

```
In [5]: df[df.a <= df.b]
```

```
 a b
0 0.026864 0.731942
1 -2.071188 -0.714342
5 -0.550596 -0.391094
[3 rows x 2 columns]
```

```
In [6]: df.loc[df.a <= df.b]
```

```
 a b
0 0.026864 0.731942
1 -2.071188 -0.714342
5 -0.550596 -0.391094
[3 rows x 2 columns]
```

For more details and examples see [the query documentation](#).

## 25.1.2 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:

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

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

```
In [7]: df = DataFrame({'a': random.randn(10), 'b': random.randn(10)})
```

```
In [8]: df.eval('a + b')
```

```
0 0.436549
1 -2.071980
2 -2.090933
3 0.447128
4 1.947388
5 -1.181133
6 2.974307
7 0.707994
8 1.424054
9 0.146349
dtype: float64
```

```
In [9]: df.a + df.b # same as the previous expression
```

```
0 0.436549
1 -2.071980
2 -2.090933
3 0.447128
4 1.947388
5 -1.181133
6 2.974307
7 0.707994
8 1.424054
9 0.146349
dtype: float64
```

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

## 25.2 zoo

## 25.3 xts

## 25.4 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.

| R          | Python                        |
|------------|-------------------------------|
| array      | list                          |
| lists      | dictionary or list of objects |
| data.frame | dataframe                     |

### 25.4.1 ddply

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

```
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:

```
In [10]: df = DataFrame({
.....: 'x': random.uniform(1., 168., 120),
.....: 'y': random.uniform(7., 334., 120),
.....: 'z': random.uniform(1.7, 20.7, 120),
.....: 'month': [5,6,7,8]*30,
.....: 'week': random.randint(1,4, 120)
.....: })
.....:

In [11]: grouped = df.groupby(['month', 'week'])

In [12]: print grouped['x'].agg([np.mean, np.std])
 mean std
month week
5 1 76.668813 50.666094
 2 68.651122 43.948919
```

```
 3 69.108800 42.935478
6 1 94.854115 58.036739
 2 97.374311 46.307307
 3 113.711425 50.303955
7 1 83.197105 45.647110
 2 94.956183 51.086197
 3 94.872662 47.258354
8 1 106.109188 49.332236
 2 73.117347 53.968658
 3 89.748792 53.828984
[12 rows x 2 columns]
```

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

## 25.5 reshape / reshape2

### 25.5.1 melt.array

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

```
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 [13]: a = np.array(range(1,24)+[np.NaN]).reshape(2,3,4)
```

```
In [14]: DataFrame([tuple(list(x)+[val]) for x, val in np.ndenumerate(a)])
```

```
 0 1 2 3
0 0 0 0 1
1 0 0 1 2
2 0 0 2 3
3 0 0 3 4
4 0 1 0 5
5 0 1 1 6
6 0 1 2 7
7 0 1 3 8
8 0 2 0 9
9 0 2 1 10
10 0 2 2 11
11 0 2 3 12
12 1 0 0 13
13 1 0 1 14
14 1 0 2 15
.. ..
[24 rows x 4 columns]
```

### 25.5.2 melt.list

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

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

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.

```
In [15]: a = list(enumerate(range(1,5)+[np.NaN]))
```

```
In [16]: DataFrame(a)
```

```

 0 1
0 0 1
1 1 2
2 2 3
3 3 4
4 4 NaN
[5 rows x 2 columns]
```

For more details and examples see [the \*Into to Data Structures\* documentation](#).

### 25.5.3 melt.data.frame

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

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

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

```
In [17]: cheese = DataFrame({'first' : ['John', 'Mary'],
.....: 'last' : ['Doe', 'Bo'],
.....: 'height' : [5.5, 6.0],
.....: 'weight' : [130, 150]})
.....:
```

```
In [18]: pd.melt(cheese, id_vars=['first', 'last'])
```

```

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 [19]: cheese.set_index(['first', 'last']).stack() # alternative way
```

```

first last
John Doe height 5.5
 Doe weight 130.0
Mary Bo height 6.0
 Bo weight 150.0
dtype: float64
```

For more details and examples see [the \*reshaping\* documentation](#).

## 25.5.4 cast

An expression using a data.frame called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
 x = runif(12, 1, 168),
 y = runif(12, 7, 334),
 z = runif(12, 1.7, 20.7),
 month = rep(c(5,6,7),4),
 week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

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

```
In [20]: df = DataFrame({
.....: 'x': random.uniform(1., 168., 12),
.....: 'y': random.uniform(7., 334., 12),
.....: 'z': random.uniform(1.7, 20.7, 12),
.....: 'month': [5,6,7]*4,
.....: 'week': [1,2]*6
.....: })

In [21]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [22]: pd.pivot_table(mdf, values='value', rows=['variable', 'week'],
.....: cols=['month'], aggfunc=np.mean)
.....:

month 5 6 7
variable week
x 1 81.157131 131.364666 63.034819
 2 47.736485 120.352953 99.131511
y 1 61.777944 18.588993 188.790929
 2 303.348717 174.792974 209.744832
z 1 2.795230 13.532019 12.555273
 2 13.226753 13.347298 4.249712
[6 rows x 3 columns]
```

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

# COMPARISON WITH SQL

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

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

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

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

```
In [2]: import numpy as np
```

Most of the examples will utilize the `tips` dataset found within pandas tests. We'll read the data into a `DataFrame` called `tips` and assume we have a database table of the same name and structure.

```
In [3]: url = 'https://raw.githubusercontent.com/pydata/pandas/master/pandas/tests/data/tips.csv'
```

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

```
In [5]: tips.head()
```

|   | 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]

## 26.1 SELECT

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

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

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

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
```

|  | total_bill | tip | smoker | time |
|--|------------|-----|--------|------|
|--|------------|-----|--------|------|

```
0 16.99 1.01 No Dinner
1 10.34 1.66 No Dinner
2 21.01 3.50 No Dinner
3 23.68 3.31 No Dinner
4 24.59 3.61 No Dinner
[5 rows x 4 columns]
```

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

## 26.2 WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using [boolean indexing](#).

```
In [7]: tips[tips['time'] == 'Dinner'].head(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]
```

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

```
In [8]: is_dinner = tips['time'] == 'Dinner'
```

```
In [9]: is_dinner.value_counts()
```

```
True 176
False 68
dtype: int64
```

```
In [10]: tips[is_dinner].head(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]
```

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)]
```

|     | total_bill | tip   | sex    | smoker | day | time   | size |
|-----|------------|-------|--------|--------|-----|--------|------|
| 23  | 39.42      | 7.58  | Male   | No     | Sat | Dinner | 4    |
| 44  | 30.40      | 5.60  | Male   | No     | Sun | Dinner | 4    |
| 47  | 32.40      | 6.00  | Male   | No     | Sun | Dinner | 4    |
| 52  | 34.81      | 5.20  | Female | No     | Sun | Dinner | 4    |
| 59  | 48.27      | 6.73  | Male   | No     | Sat | Dinner | 4    |
| 116 | 29.93      | 5.07  | Male   | No     | Sun | Dinner | 4    |
| 155 | 29.85      | 5.14  | Female | No     | Sun | Dinner | 5    |
| 170 | 50.81      | 10.00 | Male   | Yes    | Sat | Dinner | 3    |
| 172 | 7.25       | 5.15  | Male   | Yes    | Sun | Dinner | 2    |
| 181 | 23.33      | 5.65  | Male   | Yes    | Sun | Dinner | 2    |
| 183 | 23.17      | 6.50  | Male   | Yes    | Sun | Dinner | 4    |
| 211 | 25.89      | 5.16  | Male   | Yes    | Sat | Dinner | 4    |
| 212 | 48.33      | 9.00  | Male   | No     | Sat | Dinner | 4    |
| 214 | 28.17      | 6.50  | Female | Yes    | Sat | Dinner | 3    |
| 239 | 29.03      | 5.92  | Male   | No     | Sat | Dinner | 3    |

[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)]
```

|     | total_bill | tip   | sex    | smoker | day  | time   | size |
|-----|------------|-------|--------|--------|------|--------|------|
| 59  | 48.27      | 6.73  | Male   | No     | Sat  | Dinner | 4    |
| 125 | 29.80      | 4.20  | Female | No     | Thur | Lunch  | 6    |
| 141 | 34.30      | 6.70  | Male   | No     | Thur | Lunch  | 6    |
| 142 | 41.19      | 5.00  | Male   | No     | Thur | Lunch  | 5    |
| 143 | 27.05      | 5.00  | Female | No     | Thur | Lunch  | 6    |
| 155 | 29.85      | 5.14  | Female | No     | Sun  | Dinner | 5    |
| 156 | 48.17      | 5.00  | Male   | No     | Sun  | Dinner | 6    |
| 170 | 50.81      | 10.00 | Male   | Yes    | Sat  | Dinner | 3    |
| 182 | 45.35      | 3.50  | Male   | Yes    | Sun  | Dinner | 3    |
| 185 | 20.69      | 5.00  | Male   | No     | Sun  | Dinner | 5    |
| 187 | 30.46      | 2.00  | Male   | Yes    | Sun  | Dinner | 5    |
| 212 | 48.33      | 9.00  | Male   | No     | Sat  | Dinner | 4    |
| 216 | 28.15      | 3.00  | Male   | Yes    | Sat  | Dinner | 5    |

[13 rows x 7 columns]

NULL checking is done using the `notnull()` and `isnull()` methods.

```
In [13]: frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],
.....: 'col2': ['F', np.NaN, 'G', 'H', 'I']})
.....:
```

```
In [14]: frame
```

|   | col1 | col2 |
|---|------|------|
| 0 | A    | F    |
| 1 | B    | NaN  |
| 2 | NaN  | G    |
| 3 | C    | H    |

```
4 D I
[5 rows x 2 columns]
```

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

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

```
In [15]: frame[frame['col2'].isnull()]
```

```
 col1 col2
1 B NaN
[1 rows x 2 columns]
```

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

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

```
In [16]: frame[frame['col1'].notnull()]
```

```
 col1 col2
0 A F
1 B NaN
3 C H
4 D I
[4 rows x 2 columns]
```

## 26.3 GROUP BY

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

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

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

The pandas equivalent would be:

```
In [17]: tips.groupby('sex').size()
```

```
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()
```

```

 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()
```

```

sex
Female 87
Male 157
dtype: int64
```

Multiple functions can also be applied at once. For instance, say we'd like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

```

SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
/*
Fri 2.734737 19
Sat 2.993103 87
Sun 3.255132 76
Thur 2.771452 62
*/
```

```
In [20]: tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
```

```

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

```
In [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})
```

```
 tip
smoker day size mean
No Fri 4 2.812500
 Sat 45 3.102889
 Sun 57 3.167895
 Thur 45 2.673778
Yes Fri 15 2.714000
 Sat 42 2.875476
 Sun 19 3.516842
 Thur 17 3.030000
[8 rows x 2 columns]
```

## 26.4 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.

### 26.4.1 INNER JOIN

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

merge performs an INNER JOIN by default
In [24]: pd.merge(df1, df2, on='key')
```

```
 key value_x value_y
0 B 0.215896 1.446712
1 D 0.684420 2.047340
2 D 0.684420 0.073711
[3 rows x 3 columns]
```

`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)
```

```

 key value_x value_y
1 B 0.215896 1.446712
3 D 0.684420 2.047340
3 D 0.684420 0.073711
[3 rows x 3 columns]

```

## 26.4.2 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')

 key value_x value_y
0 A 1.152659 NaN
1 B 0.215896 1.446712
2 C 0.302647 NaN
3 D 0.684420 2.047340
4 D 0.684420 0.073711
[5 rows x 3 columns]

```

## 26.4.3 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')

 key value_x value_y
0 B 0.215896 1.446712
1 D 0.684420 2.047340
2 D 0.684420 0.073711
3 E NaN 0.444477
[4 rows x 3 columns]

```

## 26.4.4 FULL JOIN

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

```

-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
 ON df1.key = df2.key;

```

```
show all records from both frames
In [29]: pd.merge(df1, df2, on='key', how='outer')

 key value_x value_y
0 A 1.152659 NaN
1 B 0.215896 1.446712
2 C 0.302647 NaN
3 D 0.684420 2.047340
4 D 0.684420 0.073711
5 E NaN 0.444477
[6 rows x 3 columns]
```

## 26.5 UNION

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

```
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
San Francisco 2
New York City 3
 Chicago 1
 Boston 4
Los Angeles 5
*/
```

```
In [32]: pd.concat([df1, df2])
```

```
 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()
```

```
 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]
```

## 26.6 UPDATE

## 26.7 DELETE





# API REFERENCE

## 27.1 Input/Output

### 27.1.1 Pickling

---

|                                |                                                                             |
|--------------------------------|-----------------------------------------------------------------------------|
| <code>read_pickle(path)</code> | Load pickled pandas object (or any other pickled object) from the specified |
|--------------------------------|-----------------------------------------------------------------------------|

---

#### **pandas.read\_pickle**

`pandas.read_pickle(path)`

Load pickled pandas object (or any other pickled object) from the specified file path

Warning: Loading pickled data received from untrusted sources can be unsafe. See:  
<http://docs.python.org/2.7/library/pickle.html>

**Parameters** `path` : string

File path

**Returns** `unpickled` : type of object stored in file

### 27.1.2 Flat File

---

|                                                                |                                                            |
|----------------------------------------------------------------|------------------------------------------------------------|
| <code>read_table(filepath_or_buffer[, sep, ...])</code>        | Read general delimited file into DataFrame                 |
| <code>read_csv(filepath_or_buffer[, sep, dialect, ...])</code> | Read CSV (comma-separated) file into DataFrame             |
| <code>read_fwf(filepath_or_buffer[, colspecs, widths])</code>  | 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, double-
quote=True, escapechar=None, quotechar='"', quoting=0, skipinitialspace=False,
lineterminator=None, header='infer', index_col=None, names=None, pre-
fix=None, skiprows=None, skipfooter=None, skip_footer=0, na_values=None,
na_fvalues=None, true_values=None, false_values=None, delimiter=None, con-
verters=None, dtype=None, usecols=None, engine='c', delim_whitespace=False,
as_reccarray=False, na_filter=True, compact_ints=False, use_unsigned=False,
low_memory=True, buffer_lines=None, warn_bad_lines=True, er-
ror_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, man-
gle_dupes_cols=True, tupleize_cols=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 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

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

**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'

**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)

**Returns result** : DataFrame or TextParser

**pandas.read\_csv**

```
pandas.read_csv(filepath_or_buffer, sep=',', dialect=None, compression=None, double-
quote=True, escapechar=None, quotechar='"', quoting=0, skipinitialspace=False,
lineterminator=None, header='infer', index_col=None, names=None, pre-
fix=None, skiprows=None, skipfooter=None, skip_footer=0, na_values=None,
na_fvalues=None, true_values=None, false_values=None, delimiter=None, con-
verters=None, dtype=None, usecols=None, engine='c', delim_whitespace=False,
as_reccarray=False, na_filter=True, compact_ints=False, use_unsigned=False,
low_memory=True, buffer_lines=None, warn_bad_lines=True, error_bad_lines=True,
keep_default_na=True, thousands=None, comment=None, decimal='.',
parse_dates=False, keep_date_col=False, dayfirst=False, date_parser=None,
memory_map=False, nrows=None, iterator=False, chunksize=None, verbose=False,
encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_cols=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

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

**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'

**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)

**Returns result** : DataFrame or TextParser

## pandas.read\_fwf

`pandas.read_fwf` (*filepath\_or\_buffer*, *colspecs*='infer', *widths*=None, *\*\*kws*)

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

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

**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'

**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)

**Returns** **result** : DataFrame or TextParser

Also, 'delimiter' is used to specify the filler character of the fields if it is not spaces (e.g., '~').

|                        |
|------------------------|
| Continued on next page |
|------------------------|

Table 27.3 – continued from previous page

### 27.1.3 Clipboard

---

|                                       |                                                                |
|---------------------------------------|----------------------------------------------------------------|
| <code>read_clipboard(**kwargs)</code> | Read text from clipboard and pass to <code>read_table</code> . |
|---------------------------------------|----------------------------------------------------------------|

---

#### **pandas.read\_clipboard**

`pandas.read_clipboard(**kwargs)`

Read text from clipboard and pass to `read_table`. See `read_table` for the full argument list

**Returns** `parsed` : DataFrame

### 27.1.4 Excel

---

|                                                        |                                             |
|--------------------------------------------------------|---------------------------------------------|
| <code>read_excel(io, sheetname, **kwargs)</code>       | Read an Excel table into a pandas DataFrame |
| <code>ExcelFile.parse(sheetname[, header, ...])</code> | Read an Excel table into DataFrame          |

---

#### **pandas.read\_excel**

`pandas.read_excel(io, sheetname, **kwargs)`

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)

**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

**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, **kwargs)`  
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 → 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

## 27.1.5 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

## 27.1.6 HTML

---

`read_html(io[, match, flavor, header, ...])` Read HTML tables into a list of DataFrame objects.

---

**pandas.read\_html**

```
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 BeautifulSoup 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, and 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 BeautifulSoup. 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.

## 27.1.7 HDFStore: PyTables (HDF5)

|                                                              |                                                                  |
|--------------------------------------------------------------|------------------------------------------------------------------|
| <code>read_hdf(path_or_buf, key, **kwargs)</code>            | read from the store, close it if we opened it                    |
| <code>HDFStore.put(key, value[, format, append])</code>      | Store object in HDFStore                                         |
| <code>HDFStore.append(key, value[, format, ...])</code>      | Append to Table in file. Node must already exist and be Table    |
| <code>HDFStore.get(key)</code>                               | Retrieve pandas object stored in file                            |
| <code>HDFStore.select(key[, where, start, stop, ...])</code> | Retrieve pandas object stored in file, optionally based on where |



**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 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.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 representation

**chunksize** : size to chunk the writing

**expectedrows** : expected TOTAL row size of this table

**encoding** : default None, provide an encoding for strings

**dropna** : boolean, default True, do not write an ALL nan row to the store settable by the option ‘io.hdf.dropna\_table’

**Notes**

—

**Does *\*not\** check if data being appended overlaps with existing data in the table, so be careful**

## pandas.HDFStore.get

HDFStore.get(*key*)

Retrieve pandas object stored in file

**Parameters** **key** : object

**Returns** **obj** : type of object stored in file

## pandas.HDFStore.select

HDFStore.select(*key*, *where=None*, *start=None*, *stop=None*, *columns=None*, *iterator=False*, *chunksize=None*, *auto\_close=False*, *\*\*kwargs*)

Retrieve pandas object stored in file, optionally based on where criteria

**Parameters** **key** : object

**where** : list of Term (or 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** : n rows to include in iteration, return an iterator

**auto\_close** : boolean, should automatically close the store when finished, default is False

**Returns** The selected object

## 27.1.8 SQL

---

|                                                   |                                                                  |
|---------------------------------------------------|------------------------------------------------------------------|
| <code>read_sql(sql, con[, index_col, ...])</code> | Returns a DataFrame corresponding to the result set of the query |
|---------------------------------------------------|------------------------------------------------------------------|

---

### pandas.read\_sql

`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.

---

|                                                           |                                                                  |
|-----------------------------------------------------------|------------------------------------------------------------------|
| <code>read_frame(sql, con[, index_col, ...])</code>       | Returns a DataFrame corresponding to the result set of the query |
| <code>write_frame(frame, name, con[, flavor, ...])</code> | 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.

## 27.1.9 Google BigQuery

|                                                          |                                               |
|----------------------------------------------------------|-----------------------------------------------|
| <code>read_gbq(query[, project_id, ...])</code>          | Load data from Google BigQuery.               |
| <code>to_gbq(dataframe, destination_table[, ...])</code> | 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.

The main method a user calls to load data from Google BigQuery into a pandas DataFrame. This is a simple wrapper for Google's `bq.py` and `bigquery_client.py`, which we use to get the source data. Because of this, this script respects the user's bq settings file, `~/bigqueryrc`, if it exists. Such a file can be generated using `'bq init'`. Further, additional parameters for the query can be specified as either `**kwargs` in the command, or using `FLAGS` provided in the `'gflags'` module. Particular options can be found in `bigquery_client.py`.

**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** :

To be passed to `bq.Client.Create()`. Particularly: 'trace', 'sync', 'api', 'api\_version'

**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.

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** **dataframe** : DataFrame

DataFrame to be written

**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

## 27.1.10 STATA

---

|                                                    |                                |
|----------------------------------------------------|--------------------------------|
| <code>read_stata(filepath_or_buffer[, ...])</code> | 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

|                                                     |                                                                      |
|-----------------------------------------------------|----------------------------------------------------------------------|
| <code>StataReader.data([convert_dates, ...])</code> | Reads observations from Stata file, converting them into a dataframe |
| <code>StataReader.data_label()</code>               | Returns data label of Stata file                                     |
| <code>StataReader.value_labels()</code>             | Returns a dict, associating each variable name a dict, associating   |
| <code>StataReader.variable_labels()</code>          | Returns variable labels as a dict, associating each variable name    |
| <code>StataWriter.write_file()</code>               |                                                                      |

---

**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()`

## 27.2 General functions

### 27.2.1 Data manipulations

|                                                                |                                                                                                                                                                                                        |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>melt(frame[, id_vars, value_vars, var_name, ...])</code> | “Unpivots” a DataFrame from wide format to long format, optionally leaving id variables set                                                                                                            |
| <code>pivot_table(data[, values, rows, cols, ...])</code>      | Create a spreadsheet-style pivot table as a DataFrame. The levels in the index are given in the variable <code>rows</code> and the levels in the columns are given in the variable <code>cols</code> . |
| <code>crosstab(rows, cols[, values, rownames, ...])</code>     | Compute a simple cross-tabulation of two (or more) factors.                                                                                                                                            |
| <code>cut(x, bins[, right, labels, retbins, ...])</code>       | Return indices of half-open bins to which each value of <i>x</i> belongs.                                                                                                                              |
| <code>qcut(x, q[, labels, retbins, precision])</code>          | Quantile-based discretization function.                                                                                                                                                                |
| <code>merge(left, right[, how, on, left_on, ...])</code>       | Merge DataFrame objects by performing a database-style join operation by columns.                                                                                                                      |
| <code>concat(objs[, axis, join, join_axes, ...])</code>        | Concatenate pandas objects along a particular axis with optional set logic along the other axis.                                                                                                       |
| <code>get_dummies(data[, prefix, prefix_sep, dummy_na])</code> | Convert categorical variable into dummy/indicator variables                                                                                                                                            |

**pandas.melt**

`pandas.melt(frame, id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None)`

“Unpivots” a DataFrame from wide format to long format, optionally leaving 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**

```
>>> 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 B 1
1 b B 3
2 c B 5

>>> melt(df, id_vars=['A'], value_vars=['B'],
... var_name='myVarname', value_name='myValname')
 A myVarname myValname
0 a B 1
1 b B 3
2 c B 5

>>> 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 E 1
1 b B E 3
2 c 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
7 bar two small 6
8 bar two large 7

>>> table = pivot_table(df, values='D', rows=['A', 'B'],
... cols=['C'], aggfunc=np.sum)
>>> table
 small large
foo one 1 4
 two 6 NaN
bar one 5 4
 two 6 7
```

### pandas.crosstab

`pandas.crosstab`(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

```
>>> a
array([foo, foo, foo, foo, bar, bar,
 bar, bar, foo, foo, foo], dtype=object)
>>> b
array([one, one, one, two, one, one,
 one, two, two, two, one], dtype=object)
>>> c
array([dull, dull, shiny, dull, dull, shiny,
 shiny, dull, shiny, shiny, shiny], dtype=object)

>>> crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
b one two
c dull shiny dull shiny
a
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

```
>>> 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 >>> B
 lkey value rkey value
0 foo 1 0 foo 5
1 bar 2 1 bar 6
2 baz 3 2 qux 7
3 foo 4 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.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

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

```
>>> 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
```

## 27.2.2 Top-level missing data

|                           |                                                                                                                     |
|---------------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>isnull(obj)</code>  | Detect missing values (NaN in numeric arrays, None/NaN in object arrays)                                            |
| <code>notnull(obj)</code> | Replacement for <code>numpy.isfinite</code> / <code>-numpy.isnan</code> which is suitable for use on object arrays. |

### 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.notnull

`pandas.notnull(obj)`

Replacement for `numpy.isfinite` / `-numpy.isnan` which is suitable for use on object arrays.

**Parameters** `arr` : ndarray or object value

Object to check for *not*-null-ness

**Returns** `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.

## 27.2.3 Top-level dealing with datetimes

|                                                                |                                                                             |
|----------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>to_datetime(arg[, errors, dayfirst, utc, ...])</code>    | Convert argument to datetime                                                |
| <code>to_timedelta(arg[, box, unit])</code>                    | Convert argument to timedelta                                               |
| <code>date_range([start, end, periods, freq, tz, ...])</code>  | Return a fixed frequency datetime index, with day (calendar) as the default |
| <code>bdate_range([start, end, periods, freq, tz, ...])</code> | Return a fixed frequency datetime index, with business day as the default   |
| <code>period_range([start, end, periods, freq, name])</code>   | 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')`

Convert argument to datetime

**Parameters** `arg` : string, datetime, array of strings (with possible NAs)

**errors** : {'ignore', 'raise'}, default 'ignore'

Errors are ignored by default (values left untouched)

**dayfirst** : boolean, default False



If True parses dates with the day first, eg 20/01/2005 Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug).

**utc** : boolean, default None

Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well)

**box** : boolean, default True

If True returns a DatetimeIndex, if False returns ndarray of values

**format** : string, default None

strftime to parse time, eg “%d/%m/%Y”

**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

**Returns** **ret** : datetime if parsing succeeded

## 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

Name of the resulting index

**closed** : string or None, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

**Returns** **rng** : DatetimeIndex

#### Notes

2 of start, end, or periods must be specified

### **pandas.bdate\_range**

`pandas.bdate_range` (*start=None, end=None, periods=None, freq='B', tz=None, normalize=True, name=None, closed=None*)

Return a fixed frequency datetime index, with business day as the default frequency

**Parameters** **start** : string or datetime-like, default None

Left bound for generating dates

**end** : string or datetime-like, default None

Right bound for generating dates

**periods** : integer or None, default None

If None, must specify start and end

**freq** : string or DateOffset, default 'B' (business daily)

Frequency strings can have multiples, e.g. '5H'

**tz** : string or None

Time zone name for returning localized DatetimeIndex, for example Asia/Beijing

**normalize** : bool, default False

Normalize start/end dates to midnight before generating date range

**name** : str, default None

Name for the resulting index

**closed** : string or None, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

**Returns** **rng** : DatetimeIndex

#### Notes

2 of start, end, or periods must be specified

## pandas.period\_range

pandas.**period\_range** (*start=None, end=None, periods=None, freq='D', name=None*)

Return a fixed frequency datetime index, with day (calendar) as the default frequency

**Parameters** **start** :

**end** :

**periods** : int, default None

Number of periods in the index

**freq** : str/DateOffset, default 'D'

Frequency alias

**name** : str, default None

Name for the resulting PeriodIndex

**Returns** **prng** : PeriodIndex

## 27.2.4 Top-level evaluation

---

`eval(expr[, parser, engine, truediv, ...])` Evaluate a Python expression as a string using various backends.

---

### pandas.eval

pandas.**eval** (*expr, parser='pandas', engine='numexpr', truediv=True, local\_dict=None, global\_dict=None, resolvers=None, level=2, target=None*)

Evaluate a Python expression as a string using various backends.

The following arithmetic operations are supported: +, -, \*, /, \*\*, %, // (python engine only) along with the following boolean operations: | (or), & (and), and ~ (not). Additionally, the 'pandas' parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators. `Series` and `DataFrame` objects are supported and behave as they would with plain ol' Python evaluation.

**Parameters** **expr** : str or unicode

The expression to evaluate. This string cannot contain any Python `statements`, only Python `expressions`.

**parser** : string, default 'pandas', {'pandas', 'python'}

The parser to use to construct the syntax tree from the expression. The default of 'pandas' parses code slightly different than standard Python. Alternatively, you can parse an expression using the 'python' parser to retain strict Python semantics. See the [enhancing performance](#) documentation for more details.

**engine** : string, default 'numexpr', {'python', 'numexpr'}

The engine used to evaluate the expression. Supported engines are

- **'numexpr'** : This default engine evaluates pandas objects using `numexpr` for large speed ups in complex expressions with large frames.
- **'python'** : Performs operations as if you had `eval'd` in top level python. This engine is generally not that useful.

More backends may be available in the future.

**truediv** : bool, optional

Whether to use true division, like in Python `>= 3`

**local\_dict** : dict or None, optional

A dictionary of local variables, taken from `locals()` by default.

**global\_dict** : dict or None, optional

A dictionary of global variables, taken from `globals()` by default.

**resolvers** : list of dict-like or None, optional

A list of objects implementing the `__getitem__` special method that you can use to inject an additional collection of namespaces to use for variable lookup. For example, this is used in the `query()` method to inject the `index` and `columns` variables that refer to their respective `DataFrame` instance attributes.

**level** : int, optional

The number of prior stack frames to traverse and add to the current scope. Most users will **not** need to change this parameter.

**target** : a target object for assignment, optional, default is None

essentially this is a passed in resolver

**Returns** ndarray, numeric scalar, DataFrame, Series

**See Also:**

`pandas.DataFrame.query`, `pandas.DataFrame.eval`

#### Notes

The dtype of any objects involved in an arithmetic `%` operation are recursively cast to `float64`.

See the *enhancing performance* documentation for more details.

## 27.2.5 Standard moving window functions

|                                                               |                                                                                                                             |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <code>rolling_count(arg, window[, freq, center, ...])</code>  | Rolling count of number of non-NaN observations inside provided window                                                      |
| <code>rolling_sum(arg, window[, min_periods, ...])</code>     | Moving sum                                                                                                                  |
| <code>rolling_mean(arg, window[, min_periods, ...])</code>    | Moving mean                                                                                                                 |
| <code>rolling_median(arg, window[, min_periods, ...])</code>  | $O(N \log(\text{window}))$ implementation using skip list                                                                   |
| <code>rolling_var(arg, window[, min_periods, ...])</code>     | Unbiased moving variance                                                                                                    |
| <code>rolling_std(arg, window[, min_periods, ...])</code>     | Unbiased moving standard deviation                                                                                          |
| <code>rolling_min(arg, window[, min_periods, ...])</code>     | Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs.                                                         |
| <code>rolling_max(arg, window[, min_periods, ...])</code>     | Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs.                                                         |
| <code>rolling_corr(arg1, arg2, window[, ...])</code>          | Moving sample correlation                                                                                                   |
| <code>rolling_corr_pairwise(df, window[, min_periods])</code> | Computes pairwise rolling correlation matrices as Panel whose items are DataFrames                                          |
| <code>rolling_cov(arg1, arg2, window[, ...])</code>           | Unbiased moving covariance                                                                                                  |
| <code>rolling_skew(arg, window[, min_periods, ...])</code>    | Unbiased moving skewness                                                                                                    |
| <code>rolling_kurt(arg, window[, min_periods, ...])</code>    | Unbiased moving kurtosis                                                                                                    |
| <code>rolling_apply(arg, window, func[, ...])</code>          | Generic moving function application                                                                                         |
| <code>rolling_quantile(arg, window, quantile[, ...])</code>   | Moving quantile                                                                                                             |
| <code>rolling_window(arg[, window, win_type, ...])</code>     | Applies a moving window of type <code>window_type</code> and size <code>window</code> over each element of <code>arg</code> |

### 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

`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

`pandas.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.rolling\_std

`pandas.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.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.rolling\_corr\_pairwise

`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

`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

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.rolling\_skew

`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

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\_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* is a 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** :  $0 \leq \text{quantile} \leq 1$

**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).

## 27.2.6 Standard expanding window functions

|                                                              |                                                                           |
|--------------------------------------------------------------|---------------------------------------------------------------------------|
| <code>expanding_count(arg[, freq, center, time_rule])</code> | Expanding count of number of non-NaN observations.                        |
| <code>expanding_sum(arg[, min_periods, freq, ...])</code>    | Expanding sum                                                             |
| <code>expanding_mean(arg[, min_periods, freq, ...])</code>   | Expanding mean                                                            |
| <code>expanding_median(arg[, min_periods, freq, ...])</code> | O(N log(window)) implementation using skip list                           |
| <code>expanding_var(arg[, min_periods, freq, ...])</code>    | Unbiased expanding variance                                               |
| <code>expanding_std(arg[, min_periods, freq, ...])</code>    | Unbiased expanding standard deviation                                     |
| <code>expanding_min(arg[, min_periods, freq, ...])</code>    | Moving min of 1d array of dtype=float64 along axis=0 ignoring NaNs.       |
| <code>expanding_max(arg[, min_periods, freq, ...])</code>    | Moving max of 1d array of dtype=float64 along axis=0 ignoring NaNs.       |
| <code>expanding_corr(arg1, arg2[, min_periods, ...])</code>  | Expanding sample correlation                                              |
| <code>expanding_corr_pairwise(df[, min_periods])</code>      | Computes pairwise expanding correlation matrices as Panel whose items are |
| <code>expanding_cov(arg1, arg2[, min_periods, ...])</code>   | Unbiased expanding covariance                                             |
| <code>expanding_skew(arg[, min_periods, freq, ...])</code>   | Unbiased expanding skewness                                               |
| <code>expanding_kurt(arg[, min_periods, freq, ...])</code>   | Unbiased expanding kurtosis                                               |
| <code>expanding_apply(arg, func[, min_periods, ...])</code>  | Generic expanding function application                                    |
| <code>expanding_quantile(arg, quantile[, ...])</code>        | Expanding quantile                                                        |

### 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**

`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**

`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**

`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**

`pandas.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.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

Frequency to conform to before computing statistic

**Returns** *y* : type of input argument

### **pandas.expanding\_kurt**

`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**

`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**

`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

## 27.2.7 Exponentially-weighted moving window functions

|                                                              |                                           |
|--------------------------------------------------------------|-------------------------------------------|
| <code>ewma(arg[, com, span, halflife, ...])</code>           | Exponentially-weighted moving average     |
| <code>ewmstd(arg[, com, span, halflife, ...])</code>         | Exponentially-weighted moving std         |
| <code>ewmvar(arg[, com, span, halflife, ...])</code>         | Exponentially-weighted moving variance    |
| <code>ewmcorr(arg1, arg2[, com, span, halflife, ...])</code> | Exponentially-weighted moving correlation |
| <code>ewmcov(arg1, arg2[, com, span, halflife, ...])</code>  | Exponentially-weighted moving covariance  |

### 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

`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

`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, :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 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, :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.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:  $\alpha = 1/(1 + com)$ ,

**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.

27.3 Series

27.3.1 Constructor

|                                                            |                                                                   |
|------------------------------------------------------------|-------------------------------------------------------------------|
| <code>Series([data, index, dtype, name, copy, ...])</code> | One-dimensional ndarray with axis labels (including time series). |
|------------------------------------------------------------|-------------------------------------------------------------------|

pandas.Series

**class** `pandas.Series` (*data=None, index=None, dtype=None, name=None, copy=False, fastpath=False*)  
One-dimensional ndarray with axis labels (including time series). Labels need not be unique but must be any hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN)  
  
Operations between Series (+, -, /, \*, ...) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

- Parameters**
- data** : array-like, dict, or scalar value  
Contains data stored in Series
  - index** : array-like or Index (1d)  
Values must be unique and hashable, same length as data. Index object (or other iterable of same length as data) Will default to `np.arange(len(data))` if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.
  - dtype** : numpy.dtype or None  
If None, dtype will be inferred
  - copy** : boolean, default False, copy input data

Attributes

|                        |                                                                  |
|------------------------|------------------------------------------------------------------|
| <code>T</code>         | support for compatibility                                        |
| <code>at</code>        |                                                                  |
| <code>axes</code>      |                                                                  |
| <code>base</code>      |                                                                  |
| <code>blocks</code>    | Internal property, property synonym for <code>as_blocks()</code> |
| <code>data</code>      |                                                                  |
| <code>dtype</code>     |                                                                  |
| <code>empty</code>     | True if NDFrame is entirely empty [no items]                     |
| <code>flags</code>     |                                                                  |
| <code>ftype</code>     |                                                                  |
| <code>iat</code>       |                                                                  |
| <code>iloc</code>      |                                                                  |
| <code>imag</code>      |                                                                  |
| Continued on next page |                                                                  |

Table 27.21 – continued from previous page

|                             |                          |
|-----------------------------|--------------------------|
| <code>is_time_series</code> |                          |
| <code>ix</code>             |                          |
| <code>loc</code>            |                          |
| <code>ndim</code>           |                          |
| <code>real</code>           |                          |
| <code>shape</code>          |                          |
| <code>size</code>           |                          |
| <code>strides</code>        |                          |
| <code>values</code>         | Return Series as ndarray |
| <code>weekday</code>        |                          |

**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.empty**

`Series.empty`  
True if NDFrame is entirely empty [no items]

**pandas.Series.flags**

`Series.flags`

**pandas.Series.ftype**

`Series.ftype`

**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`

|         |  |
|---------|--|
| is_copy |  |
|---------|--|

**Methods**

|                                                           |                                                                                            |
|-----------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>abs()</code>                                        | Return an object with absolute value taken.                                                |
| <code>add(other[, level, fill_value, axis])</code>        | Binary operator add with support to substitute a <code>fill_value</code> for missing data  |
| <code>add_prefix(prefix)</code>                           | Concatenate prefix string with panel items names.                                          |
| <code>add_suffix(suffix)</code>                           | Concatenate suffix string with panel items names                                           |
| <code>align(other[, join, axis, level, copy, ...])</code> | Align two object on their axes with the                                                    |
| <code>all([axis, out])</code>                             | Returns True if all elements evaluate to True.                                             |
| <code>any([axis, out])</code>                             | Returns True if any of the elements of <i>a</i> evaluate to True.                          |
| <code>append(to_append[, verify_integrity])</code>        | Concatenate two or more Series. The indexes must not overlap                               |
| <code>apply(func[, convert_dtype, args])</code>           | Invoke function on values of Series. Can be <code>ufunc</code> (a NumPy function           |
| <code>argmax([axis, out, skipna])</code>                  | Index of first occurrence of maximum of values.                                            |
| <code>argmin([axis, out, skipna])</code>                  | Index of first occurrence of minimum of values.                                            |
| <code>argsort([axis, kind, order])</code>                 | Overrides <code>ndarray.argsort</code> .                                                   |
| <code>as_blocks([columns])</code>                         | Convert the frame to a dict of dtype -> Constructor Types that each has                    |
| <code>as_matrix([columns])</code>                         | Convert the frame to its Numpy-array matrix representation. Columns                        |
| <code>asfreq(freq[, method, how, normalize])</code>       | Convert all TimeSeries inside to specified frequency using <code>DateOffset</code>         |
| <code>asof(where)</code>                                  | Return last good (non-NaN) value in TimeSeries if value is NaN for                         |
| <code>astype(dtype[, copy, raise_on_error])</code>        | Cast object to input <code>numpy.dtype</code>                                              |
| <code>at_time(time[, asof])</code>                        | Select values at particular time of day (e.g.                                              |
| <code>autocorr()</code>                                   | Lag-1 autocorrelation                                                                      |
| <code>between(left, right[, inclusive])</code>            | Return boolean Series equivalent to <code>left &lt;= series &lt;= right</code> . NA values |
| <code>between_time(start_time, end_time[, ...])</code>    | Select values between particular times of the day (e.g., 9:00-9:30 AM)                     |
| <code>bfill([axis, inplace, limit, downcast])</code>      | Synonym for <code>NDFrame.fillna(method='bfill')</code>                                    |
| <code>bool()</code>                                       | Return the bool of a single element <code>PandasObject</code>                              |
| <code>clip([lower, upper, out])</code>                    | Trim values at input threshold(s)                                                          |
| <code>clip_lower(threshold)</code>                        | Return copy of the input with values below given value truncated                           |

Continued c

Table 27.22 – continued from previous page

|                                                               |                                                                                                             |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>clip_upper(threshold)</code>                            | Return copy of input with values above given value truncated                                                |
| <code>combine(other, func[, fill_value])</code>               | Perform elementwise binary operation on two Series using given function                                     |
| <code>combine_first(other)</code>                             | Combine Series values, choosing the calling Series's values                                                 |
| <code>compound([axis, skipna, level])</code>                  | Return the compound percentage of the values for the requested axis                                         |
| <code>consolidate([inplace])</code>                           | Compute NDFrame with “consolidated” internals (data of each dtype)                                          |
| <code>convert_objects([convert_dates, ...])</code>            | Attempt to infer better dtype for object columns                                                            |
| <code>copy([deep])</code>                                     | Make a copy of this object                                                                                  |
| <code>corr(other[, method, min_periods])</code>               | Compute correlation with <i>other</i> Series, excluding missing values                                      |
| <code>count([level])</code>                                   | Return number of non-NA/null observations in the Series                                                     |
| <code>cov(other[, min_periods])</code>                        | Compute covariance with Series, excluding missing values                                                    |
| <code>cummax([axis, dtype, out, skipna])</code>               | Return cumulative max over requested axis.                                                                  |
| <code>cummin([axis, dtype, out, skipna])</code>               | Return cumulative min over requested axis.                                                                  |
| <code>cumprod([axis, dtype, out, skipna])</code>              | Return cumulative prod over requested axis.                                                                 |
| <code>cumsum([axis, dtype, out, skipna])</code>               | Return cumulative sum over requested axis.                                                                  |
| <code>describe([percentile_width])</code>                     | Generate various summary statistics of Series, excluding NaN                                                |
| <code>diff([periods])</code>                                  | 1st discrete difference of object                                                                           |
| <code>div(other[, level, fill_value, axis])</code>            | Binary operator <code>truediv</code> with support to substitute a <code>fill_value</code> for missing data  |
| <code>divide(other[, level, fill_value, axis])</code>         | Binary operator <code>truediv</code> with support to substitute a <code>fill_value</code> for missing data  |
| <code>dot(other)</code>                                       | Matrix multiplication with DataFrame or inner-product with Series                                           |
| <code>drop(labels[, axis, level, inplace])</code>             | Return new object with labels in requested axis removed                                                     |
| <code>drop_duplicates([take_last, inplace])</code>            | Return Series with duplicate values removed                                                                 |
| <code>dropna([axis, inplace])</code>                          | Return Series without null values                                                                           |
| <code>duplicated([take_last])</code>                          | Return boolean Series denoting duplicate values                                                             |
| <code>eq(other)</code>                                        |                                                                                                             |
| <code>ffill([axis, inplace, limit, downcast])</code>          | Synonym for <code>NDFrame.fillna(method='ffill')</code>                                                     |
| <code>fillna([value, method, axis, inplace, ...])</code>      | Fill NA/NaN values using the specified method                                                               |
| <code>filter([items, like, regex, axis])</code>               | Restrict the info axis to set of items or wildcard                                                          |
| <code>first(offset)</code>                                    | Convenience method for subsetting initial periods of time series data                                       |
| <code>first_valid_index()</code>                              | Return label for first non-NA/null value                                                                    |
| <code>floordiv(other[, level, fill_value, axis])</code>       | Binary operator <code>floordiv</code> with support to substitute a <code>fill_value</code> for missing data |
| <code>from_array(arr[, index, name, copy, fastpath])</code>   |                                                                                                             |
| <code>from_csv(path[, sep, parse_dates, header, ...])</code>  | Read delimited file into Series                                                                             |
| <code>ge(other)</code>                                        |                                                                                                             |
| <code>get(label[, default])</code>                            | Returns value occupying requested label, default to specified missing value if                              |
| <code>get_dtype_counts()</code>                               | return the counts of dtypes in this frame                                                                   |
| <code>get_ftype_counts()</code>                               | return the counts of ftypes in this frame                                                                   |
| <code>get_value(label)</code>                                 | Quickly retrieve single value at passed index label                                                         |
| <code>get_values()</code>                                     | same as <code>values</code> (but handles sparseness conversions); is a view                                 |
| <code>groupby([by, axis, level, as_index, sort, ...])</code>  | Group series using mapper (dict or key function, apply given function                                       |
| <code>gt(other)</code>                                        |                                                                                                             |
| <code>head([n])</code>                                        | Returns first n rows                                                                                        |
| <code>hist([by, ax, grid, xlabelsize, xrot, ...])</code>      | Draw histogram of the input series using matplotlib                                                         |
| <code>idxmax([axis, out, skipna])</code>                      | Index of first occurrence of maximum of values.                                                             |
| <code>idxmin([axis, out, skipna])</code>                      | Index of first occurrence of minimum of values.                                                             |
| <code>iget(i[, axis])</code>                                  | Return the i-th value or values in the Series by location                                                   |
| <code>iget_value(i[, axis])</code>                            | Return the i-th value or values in the Series by location                                                   |
| <code>interpolate([method, axis, limit, inplace, ...])</code> | Interpolate values according to different methods.                                                          |
| <code>irow(i[, axis])</code>                                  | Return the i-th value or values in the Series by location                                                   |
| <code>isin(values)</code>                                     | Return a boolean <i>Series</i> showing whether each element                                                 |
| <code>isnull()</code>                                         | Return a boolean same-sized object indicating if the values are null                                        |

Continued c



Table 27.22 – continued from previous page

|                                                                |                                                                                   |
|----------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>item()</code>                                            |                                                                                   |
| <code>iteritems()</code>                                       | Lazily iterate over (index, value) tuples                                         |
| <code>iterkv(*args, **kwargs)</code>                           | iteritems alias used to get around 2to3. Deprecated                               |
| <code>keys()</code>                                            | Alias for index                                                                   |
| <code>kurt([axis, skipna, level, numeric_only])</code>         | Return unbiased kurtosis over requested axis                                      |
| <code>kurtosis([axis, skipna, level, numeric_only])</code>     | Return unbiased kurtosis over requested axis                                      |
| <code>last(offset)</code>                                      | Convenience method for subsetting final periods of time series data               |
| <code>last_valid_index()</code>                                | Return label for last non-NA/null value                                           |
| <code>le(other)</code>                                         |                                                                                   |
| <code>load(path)</code>                                        | Deprecated.                                                                       |
| <code>lt(other)</code>                                         |                                                                                   |
| <code>mad([axis, skipna, level])</code>                        | Return the mean absolute deviation of the values for the requested axis           |
| <code>map(arg[, na_action])</code>                             | Map values of Series using input correspondence (which can be                     |
| <code>mask(cond)</code>                                        | Returns copy whose values are replaced with nan if the                            |
| <code>max([axis, skipna, level, numeric_only])</code>          | This method returns the maximum of the values in the object.                      |
| <code>mean([axis, skipna, level, numeric_only])</code>         | Return the mean of the values for the requested axis                              |
| <code>median([axis, skipna, level, numeric_only])</code>       | Return the median of the values for the requested axis                            |
| <code>min([axis, skipna, level, numeric_only])</code>          | This method returns the minimum of the values in the object.                      |
| <code>mod(other[, level, fill_value, axis])</code>             | Binary operator mod with support to substitute a fill_value for missing data      |
| <code>mode()</code>                                            | Returns the mode(s) of the dataset.                                               |
| <code>mul(other[, level, fill_value, axis])</code>             | Binary operator mul with support to substitute a fill_value for missing data      |
| <code>multiply(other[, level, fill_value, axis])</code>        | Binary operator mul with support to substitute a fill_value for missing data      |
| <code>ne(other)</code>                                         |                                                                                   |
| <code>nonzero()</code>                                         | numpy like, returns same as nonzero                                               |
| <code>notnull()</code>                                         | Return a boolean same-sized object indicating if the values are                   |
| <code>nunique()</code>                                         | Return count of unique elements in the Series                                     |
| <code>order([na_last, ascending, kind])</code>                 | Sorts Series object, by value, maintaining index-value link                       |
| <code>pct_change([periods, fill_method, limit, freq])</code>   | Percent change over given number of periods                                       |
| <code>plot(series[, label, kind, use_index, rot, ...])</code>  | Plot the input series with the index on the x-axis using matplotlib               |
| <code>pop(item)</code>                                         | Return item and drop from frame.                                                  |
| <code>pow(other[, level, fill_value, axis])</code>             | Binary operator pow with support to substitute a fill_value for missing data      |
| <code>prod([axis, skipna, level, numeric_only])</code>         | Return the product of the values for the requested axis                           |
| <code>product([axis, skipna, level, numeric_only])</code>      | Return the product of the values for the requested axis                           |
| <code>ptp([axis, out])</code>                                  |                                                                                   |
| <code>put(*args, **kwargs)</code>                              |                                                                                   |
| <code>quantile([q])</code>                                     | Return value at the given quantile, a la scoreatpercentile in                     |
| <code>radd(other[, level, fill_value, axis])</code>            | Binary operator radd with support to substitute a fill_value for missing data     |
| <code>rank([method, na_option, ascending])</code>              | Compute data ranks (1 through n).                                                 |
| <code>ravel([order])</code>                                    |                                                                                   |
| <code>rdiv(other[, level, fill_value, axis])</code>            | Binary operator rtruediv with support to substitute a fill_value for missing data |
| <code>reindex([index])</code>                                  | Conform Series to new index with optional filling logic, placing                  |
| <code>reindex_axis(labels[, axis])</code>                      | for compatibility with higher dims                                                |
| <code>reindex_like(other[, method, copy, limit])</code>        | return an object with matching indicies to myself                                 |
| <code>rename([index])</code>                                   | Alter axes input function or functions.                                           |
| <code>rename_axis(mapper[, axis, copy, inplace])</code>        | Alter index and / or columns using input function or functions.                   |
| <code>reorder_levels(order)</code>                             | Rearrange index levels using input order.                                         |
| <code>repeat(reps)</code>                                      | See ndarray.repeat                                                                |
| <code>replace([to_replace, value, inplace, limit, ...])</code> | Replace values given in 'to_replace' with 'value'.                                |
| <code>resample(rule[, how, axis, fill_method, ...])</code>     | Convenience method for frequency conversion and resampling of regular time        |
| <code>reset_index([level, drop, name, inplace])</code>         | Analogous to the DataFrame.reset_index function, see docstring there.             |

Continued c

Table 27.22 – continued from previous page

|                                                               |                                                                                                              |
|---------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <code>reshape(*args, **kwargs)</code>                         | See <code>numpy.ndarray.reshape</code>                                                                       |
| <code>rfloordiv(other[, level, fill_value, axis])</code>      | Binary operator <code>rfloordiv</code> with support to substitute a <code>fill_value</code> for missing data |
| <code>rmod(other[, level, fill_value, axis])</code>           | Binary operator <code>rmod</code> with support to substitute a <code>fill_value</code> for missing data      |
| <code>rmul(other[, level, fill_value, axis])</code>           | Binary operator <code>rmul</code> with support to substitute a <code>fill_value</code> for missing data      |
| <code>round([decimals, out])</code>                           | Return <i>a</i> with each element rounded to the given number of decimals.                                   |
| <code>rpow(other[, level, fill_value, axis])</code>           | Binary operator <code>rpow</code> with support to substitute a <code>fill_value</code> for missing data      |
| <code>rsub(other[, level, fill_value, axis])</code>           | Binary operator <code>rsub</code> with support to substitute a <code>fill_value</code> for missing data      |
| <code>rtruediv(other[, level, fill_value, axis])</code>       | Binary operator <code>rtruediv</code> with support to substitute a <code>fill_value</code> for missing data  |
| <code>save(path)</code>                                       | Deprecated.                                                                                                  |
| <code>select(crit[, axis])</code>                             | Return data corresponding to axis labels matching criteria                                                   |
| <code>set_value(label, value)</code>                          | Quickly set single value at passed label.                                                                    |
| <code>shift([periods, freq, axis])</code>                     | Shift index by desired number of periods with an optional time freq                                          |
| <code>skew([axis, skipna, level, numeric_only])</code>        | Return unbiased skew over requested axis                                                                     |
| <code>sort([axis, kind, order, ascending])</code>             | Sort values and index labels by value, in place.                                                             |
| <code>sort_index([ascending])</code>                          | Sort object by labels (along an axis)                                                                        |
| <code>sortlevel([level, ascending])</code>                    | Sort Series with MultiIndex by chosen level. Data will be                                                    |
| <code>squeeze()</code>                                        | squeeze length 1 dimensions                                                                                  |
| <code>std([axis, skipna, level, ddof])</code>                 | Return unbiased standard deviation over requested axis                                                       |
| <code>sub(other[, level, fill_value, axis])</code>            | Binary operator <code>sub</code> with support to substitute a <code>fill_value</code> for missing data       |
| <code>subtract(other[, level, fill_value, axis])</code>       | Binary operator <code>sub</code> with support to substitute a <code>fill_value</code> for missing data       |
| <code>sum([axis, skipna, level, numeric_only])</code>         | Return the sum of the values for the requested axis                                                          |
| <code>swapaxes(axis1, axis2[, copy])</code>                   | Interchange axes and swap values axes appropriately                                                          |
| <code>swaplevel(i, j[, copy])</code>                          | Swap levels <i>i</i> and <i>j</i> in a MultiIndex                                                            |
| <code>tail([n])</code>                                        | Returns last <i>n</i> rows                                                                                   |
| <code>take(indices[, axis, convert])</code>                   | Analogous to <code>ndarray.take</code> , return Series corresponding to requested                            |
| <code>to_clipboard([excel, sep])</code>                       | Attempt to write text representation of object to the system clipboard                                       |
| <code>to_csv(path[, index, sep, na_rep, ...])</code>          | Write Series to a comma-separated values (csv) file                                                          |
| <code>to_dense()</code>                                       | Return dense representation of NDFrame (as opposed to sparse)                                                |
| <code>to_dict()</code>                                        | Convert Series to {label -> value} dict                                                                      |
| <code>to_frame([name])</code>                                 | Convert Series to DataFrame                                                                                  |
| <code>to_hdf(path_or_buf, key, **kwargs)</code>               | activate the HDFStore                                                                                        |
| <code>to_json([path_or_buf, orient, date_format, ...])</code> | Convert the object to a JSON string.                                                                         |
| <code>to_msgpack([path_or_buf])</code>                        | msgpack (serialize) object to input file path                                                                |
| <code>to_period([freq, copy])</code>                          | Convert TimeSeries from DatetimeIndex to PeriodIndex with desired                                            |
| <code>to_pickle(path)</code>                                  | Pickle (serialize) object to input file path                                                                 |
| <code>to_sparse([kind, fill_value])</code>                    | Convert Series to SparseSeries                                                                               |
| <code>to_string([buf, na_rep, float_format, ...])</code>      | Render a string representation of the Series                                                                 |
| <code>to_timestamp([freq, how, copy])</code>                  | Cast to datetimeindex of timestamps, at <i>beginning</i> of period                                           |
| <code>tolist()</code>                                         | Convert Series to a nested list                                                                              |
| <code>transpose()</code>                                      | support for compatibility                                                                                    |
| <code>truediv(other[, level, fill_value, axis])</code>        | Binary operator <code>truediv</code> with support to substitute a <code>fill_value</code> for missing data   |
| <code>truncate([before, after, axis, copy])</code>            | Truncates a sorted NDFrame before and/or after some particular                                               |
| <code>tshift([periods, freq, axis])</code>                    | Shift the time index, using the index's frequency if available                                               |
| <code>tz_convert(tz[, copy])</code>                           | Convert TimeSeries to target time zone                                                                       |
| <code>tz_localize(tz[, copy, infer_dst])</code>               | Localize tz-naive TimeSeries to target time zone                                                             |
| <code>unique()</code>                                         | Return array of unique values in the Series. Significantly faster than                                       |
| <code>unstack([level])</code>                                 | Unstack, a.k.a.                                                                                              |
| <code>update(other)</code>                                    | Modify Series in place using non-NA values from passed                                                       |
| <code>valid([inplace])</code>                                 |                                                                                                              |
| <code>value_counts([normalize, sort, ascending, bins])</code> | Returns Series containing counts of unique values. The resulting Series                                      |

Continued c

Table 27.22 – continued from previous page

|                                                              |                                                                |
|--------------------------------------------------------------|----------------------------------------------------------------|
| <code>var([axis, skipna, level, ddof])</code>                | Return unbiased variance over requested axis                   |
| <code>view([dtype])</code>                                   |                                                                |
| <code>where(cond[, other, inplace, axis, level, ...])</code> | Return an object of same shape as self and whose corresponding |
| <code>xs(key[, axis, level, copy])</code>                    |                                                                |

**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

**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=()*, *\*\*kws*)

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.argmax`.

## 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', '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 methdo

**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**

**Series.convert\_objects** (*convert\_dates=True, convert\_numeric=False, 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

**copy** : Boolean, if True, return copy, default is True

**Returns** **converted** : asm as input object

#### **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.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** `dified` : 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 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.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** : bool (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.ffill**`Series.fffll` (*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 NoneMethod 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-&gt;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.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

### **pandas.Series.first**

`Series.first(offset)`

Convenience method for subsetting initial periods of time series data based on a date offset

**Parameters** **offset** : string, DateOffset, dateutil.relativedelta

**Returns** **subset** : type of caller

### Examples

ts.last('10D') -> First 10 days

### **pandas.Series.first\_valid\_index**

`Series.first_valid_index()`

Return label for first non-NA/null value

### **pandas.Series.floordiv**

`Series.floordiv(other, level=None, fill_value=None, axis=0)`

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)

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

**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 frame

### **pandas.Series.get\_ftype\_counts**

**Series.get\_ftype\_counts()**  
return the counts of ftypes in this frame

### **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

### **pandas.Series.groupby**

**Series.groupby**(*by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns

**Parameters** **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

**axis** : int, default 0

**level** : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

**as\_index** : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as\_index=False is effectively “SQL-style” grouped output

**sort** : boolean, default True

Sort group keys. Get better performance by turning this off

**group\_keys** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**Returns** GroupBy object

## Examples

```
DataFrame 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.Series.gt

`Series.gt` (*other*)

## pandas.Series.head

`Series.head` (*n=5*)  
Returns first *n* rows

## pandas.Series.hist

`Series.hist` (*by=None, ax=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, figsize=None, \*\*kwargs*)  
Draw histogram of the input series using matplotlib

**Parameters** *by* : object, optional

If passed, then used to form histograms for separate groups

**ax** : matplotlib axis object

If not passed, uses `gca()`

**grid** : 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

**kwargs** : 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 an 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 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`:

```
from pandas import Series
s = Series(list('abc'))
s.isin('a')
```

Instead, turn a single string into a list of one element:

```
from pandas import Series
s = Series(list('abc'))
s.isin(['a'])
```

**Returns** `isin` : Series (bool dtype)

**Raises** `TypeError`

- If `values` is a string

**See Also:**

`pandas.DataFrame.isin`

### 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** : Series

same index as caller

### Examples

```
>>> 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** **cond** : boolean NDFrame or array

**Returns** **wh**: 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** **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.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** **nunique** : 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

**Returns** `y` : Series

**pandas.Series.pct\_change**

`Series.pct_change` (*periods=1, fill\_method='pad', limit=None, freq=None, \*\*kws*)

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, \*\*kws*)

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

$0 \leq q \leq 1$

**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 NoneMethod 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

```
>>> 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

`Series.repeat` (*reps*)

See ndarray.repeat

### pandas.Series.replace

`Series.replace` (*to\_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None*)

Replace values given in 'to\_replace' with 'value'.

**Parameters** **to\_replace** : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - str: string exactly matching *to\_replace* will be replaced with *value*
  - regex: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if *regex=True* then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.

- Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- None:
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If `value` is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill

**regex** : bool or same types as `to_replace`, default False

Whether to interpret `to_replace` and/or `value` as regular expressions. If this is True then `to_replace` must be a string. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method** : string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a list.

**Returns** `filled` : 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.resample

`Series.resample` (*rule*, *how=None*, *axis=0*, *fill\_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*)

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** **rule** : 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, fill\_method for upsampling, default None

**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 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.

**See Also:**

`numpy.around` equivalent function

#### **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.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.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 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.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, \*\*kws*)

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 is 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)}**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.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:

`pandas.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, an identifier for the group in the store  
**mode** : optional, {'a', 'w', 'r', 'r+'}, default 'a'  
    **'r'** Read-only; no data can be modified.  
    **'w'** Write; a new file is created (an existing file with the same name would be deleted).  
    **'a'** Append; an existing file is opened for reading and writing, and if the file does not exist it is created.  
    **'r+'** It is similar to 'a', but the file must already exist.  
**format** : 'fixed(f)|table(t)', default is 'fixed'  
    **fixed(f)** [Fixed format] Fast writing/reading. Not-appendable, nor searchable  
    **table(t)** [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data  
**append** : boolean, default False  
    For Table formats, append the input data to the existing  
**complevel** : int, 1-9, default 0  
    If a complib is specified compression will be applied where possible  
**complib** : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None  
    If complevel is > 0 apply compression to objects written in the store wherever possible  
**fletcher32** : bool, default False  
    If applying compression use the fletcher32 checksum

**pandas.Series.to\_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** : type of date conversion, epoch or iso

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 NDFrame before and/or after some particular dates.

**Parameters** **before** : date

Truncate before date

**after** : date

Truncate after date

**axis** : the truncation axis, defaults to the stat axis

**copy** : boolean, default is True,

return a copy of the truncated section

**Returns** **truncated** : type of caller

#### **pandas.Series.tshift**

**Series.tshift** (*periods=1, freq=None, axis=0, \*\*kws*)

Shift the time index, using the index's frequency if available

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, default None

Increment to use from datetools module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns** **shifted** : NDFrame

#### **Notes**

If freq is not specified then tries to use the freq or inferred\_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

#### **pandas.Series.tz\_convert**

**Series.tz\_convert** (*tz, 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

### `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

```
>>> 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*)

## 27.3.2 Attributes and underlying data

### Axes

- **index**: axis labels

|                               |                                                                      |
|-------------------------------|----------------------------------------------------------------------|
| <code>Series.values</code>    | Return Series as ndarray                                             |
| <code>Series.dtype</code>     |                                                                      |
| <code>Series.isnull()</code>  | Return a boolean same-sized object indicating if the values are null |
| <code>Series.notnull()</code> | Return a boolean same-sized object indicating if the values are      |

## pandas.Series.values

`Series.values`

Return Series as ndarray

**Returns** `arr` : numpy.ndarray

## pandas.Series.dtype

`Series.dtype`

## 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

## 27.3.3 Conversion

|                                                           |                                                                      |
|-----------------------------------------------------------|----------------------------------------------------------------------|
| <code>Series.astype(dtype[, copy, raise_on_error])</code> | Cast object to input numpy.dtype                                     |
| <code>Series.copy([deep])</code>                          | Make a copy of this object                                           |
| <code>Series.isnull()</code>                              | Return a boolean same-sized object indicating if the values are null |
| <code>Series.notnull()</code>                             | Return a boolean same-sized object indicating if the values are      |

## 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

**27.3.4 Indexing, iteration**

|                                           |                                                                                             |
|-------------------------------------------|---------------------------------------------------------------------------------------------|
| <code>Series.get(label[, default])</code> | Returns value occupying requested label, default to specified missing value if not present. |
| <code>Series.at</code>                    |                                                                                             |
| <code>Series.iat</code>                   |                                                                                             |
| <code>Series.ix</code>                    |                                                                                             |
| <code>Series.loc</code>                   |                                                                                             |
| <code>Series.iloc</code>                  |                                                                                             |
| <code>Series.__iter__()</code>            |                                                                                             |
| <code>Series.iteritems()</code>           | Lazily iterate over (index, value) tuples                                                   |

**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*.

## 27.3.5 Binary operator functions

|                                                   |                                                                                    |
|---------------------------------------------------|------------------------------------------------------------------------------------|
| Series.add(other[, level, fill_value, axis])      | Binary operator add with support to substitute a fill_value for missing data       |
| Series.sub(other[, level, fill_value, axis])      | Binary operator sub with support to substitute a fill_value for missing data       |
| Series.mul(other[, level, fill_value, axis])      | Binary operator mul with support to substitute a fill_value for missing data       |
| Series.div(other[, level, fill_value, axis])      | Binary operator truediv with support to substitute a fill_value for missing data   |
| Series.truediv(other[, level, fill_value, axis])  | Binary operator truediv with support to substitute a fill_value for missing data   |
| Series.floordiv(other[, level, fill_value, axis]) | Binary operator floordiv with support to substitute a fill_value for missing data  |
| Series.mod(other[, level, fill_value, axis])      | Binary operator mod with support to substitute a fill_value for missing data       |
| Series.pow(other[, level, fill_value, axis])      | Binary operator pow with support to substitute a fill_value for missing data       |
| Series.radd(other[, level, fill_value, axis])     | Binary operator radd with support to substitute a fill_value for missing data      |
| Series.rsub(other[, level, fill_value, axis])     | Binary operator rsub with support to substitute a fill_value for missing data      |
| Series.rmul(other[, level, fill_value, axis])     | Binary operator rmul with support to substitute a fill_value for missing data      |
| Series.rdiv(other[, level, fill_value, axis])     | Binary operator rtruediv with support to substitute a fill_value for missing data  |
| Series.rtruediv(other[, level, fill_value, axis]) | Binary operator rtruediv with support to substitute a fill_value for missing data  |
| Series.rfloordiv(other[, level, fill_value, ...]) | Binary operator rfloordiv with support to substitute a fill_value for missing data |
| Series.rmod(other[, level, fill_value, axis])     | Binary operator rmod with support to substitute a fill_value for missing data      |
| Series.rpow(other[, level, fill_value, axis])     | Binary operator rpow with support to substitute a fill_value for missing data      |
| Series.combine(other, func[, fill_value])         | Perform elementwise binary operation on two Series using given function            |
| Series.combine_first(other)                       | Combine Series values, choosing the calling Series's values                        |
| Series.round([decimals, out])                     | Return a with each element rounded to the given number of decimals.                |
| Series.lt(other)                                  |                                                                                    |
| Series.gt(other)                                  |                                                                                    |
| Series.le(other)                                  |                                                                                    |
| Series.ge(other)                                  |                                                                                    |
| Series.ne(other)                                  |                                                                                    |
| Series.eq(other)                                  |                                                                                    |

## 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.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.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.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 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.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.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*)

## **27.3.6 Function application, GroupBy**

|                                                               |                                                                       |
|---------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>Series.apply(func[, convert_dtype, args])</code>        | Invoke function on values of Series. Can be ufunc (a NumPy function   |
| <code>Series.map(arg[, na_action])</code>                     | Map values of Series using input correspondence (which can be         |
| <code>Series.groupby([by, axis, level, as_index, ...])</code> | Group series using mapper (dict or key function, apply given function |

### **pandas.Series.apply**

`Series.apply` (*func*, *convert\_dtype=True*, *args=()*, *\*\*kws*)

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

**Parameters** **func** : function

**convert\_dtype** : boolean, default True

Try to find better dtype for elementwise function results. If False, leave as dtype=object

**args** : tuple

Positional arguments to pass to function in addition to the value

**Additional keyword arguments will be passed as keywords to the function**

**Returns** *y* : Series or DataFrame if func returns a Series

**See Also:**

[`Series.map`](#) For element-wise operations

### `pandas.Series.map`

`Series.map` (*arg*, *na\_action=None*)

Map values of Series using input correspondence (which can be a dict, Series, or function)

**Parameters** *arg* : function, dict, or Series

*na\_action* : {None, 'ignore'}

If 'ignore', propagate NA values

**Returns** *y* : Series

same index as caller

### Examples

```
>>> 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

```
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()
```

## 27.3.7 Computations / Descriptive Stats

|                                                               |                                                                                            |
|---------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>Series.abs()</code>                                     | Return an object with absolute value taken.                                                |
| <code>Series.any([axis, out])</code>                          | Returns True if any of the elements of <i>a</i> evaluate to True.                          |
| <code>Series.autocorr()</code>                                | Lag-1 autocorrelation                                                                      |
| <code>Series.between(left, right[, inclusive])</code>         | Return boolean Series equivalent to <code>left &lt;= series &lt;= right</code> . NA values |
| <code>Series.clip([lower, upper, out])</code>                 | Trim values at input threshold(s)                                                          |
| <code>Series.clip_lower(threshold)</code>                     | Return copy of the input with values below given value truncated                           |
| <code>Series.clip_upper(threshold)</code>                     | Return copy of input with values above given value truncated                               |
| <code>Series.corr(other[, method, min_periods])</code>        | Compute correlation with <i>other</i> Series, excluding missing values                     |
| <code>Series.count([level])</code>                            | Return number of non-NA/null observations in the Series                                    |
| <code>Series.cov(other[, min_periods])</code>                 | Compute covariance with Series, excluding missing values                                   |
| <code>Series.cummax([axis, dtype, out, skipna])</code>        | Return cumulative max over requested axis.                                                 |
| <code>Series.cummin([axis, dtype, out, skipna])</code>        | Return cumulative min over requested axis.                                                 |
| <code>Series.cumprod([axis, dtype, out, skipna])</code>       | Return cumulative prod over requested axis.                                                |
| <code>Series.cumsum([axis, dtype, out, skipna])</code>        | Return cumulative sum over requested axis.                                                 |
| <code>Series.describe([percentile_width])</code>              | Generate various summary statistics of Series, excluding NaN                               |
| <code>Series.diff([periods])</code>                           | 1st discrete difference of object                                                          |
| <code>Series.kurt([axis, skipna, level, numeric_only])</code> | Return unbiased kurtosis over requested axis                                               |
| <code>Series.mad([axis, skipna, level])</code>                | Return the mean absolute deviation of the values for the requested axis                    |
| <code>Series.max([axis, skipna, level, numeric_only])</code>  | This method returns the maximum of the values in the object.                               |
| <code>Series.mean([axis, skipna, level, numeric_only])</code> | Return the mean of the values for the requested axis                                       |
| <code>Series.median([axis, skipna, level, ...])</code>        | Return the median of the values for the requested axis                                     |
| <code>Series.min([axis, skipna, level, numeric_only])</code>  | This method returns the minimum of the values in the object.                               |
| <code>Series.mode()</code>                                    | Returns the mode(s) of the dataset.                                                        |
| <code>Series.nunique()</code>                                 | Return count of unique elements in the Series                                              |
| <code>Series.pct_change([periods, fill_method, ...])</code>   | Percent change over given number of periods                                                |
| <code>Series.prod([axis, skipna, level, numeric_only])</code> | Return the product of the values for the requested axis                                    |

Continued on next page



Table 27.28 – continued from previous page

|                                                               |                                                                         |
|---------------------------------------------------------------|-------------------------------------------------------------------------|
| <code>Series.quantile([q])</code>                             | Return value at the given quantile, a la scoreatpercentile in           |
| <code>Series.rank([method, na_option, ascending])</code>      | Compute data ranks (1 through n).                                       |
| <code>Series.skew([axis, skipna, level, numeric_only])</code> | Return unbiased skew over requested axis                                |
| <code>Series.std([axis, skipna, level, ddof])</code>          | Return unbiased standard deviation over requested axis                  |
| <code>Series.sum([axis, skipna, level, numeric_only])</code>  | Return the sum of the values for the requested axis                     |
| <code>Series.unique()</code>                                  | Return array of unique values in the Series. Significantly faster than  |
| <code>Series.var([axis, skipna, level, ddof])</code>          | Return unbiased variance over requested axis                            |
| <code>Series.value_counts([normalize, sort, ...])</code>      | Returns Series containing counts of unique values. The resulting Series |

**pandas.Series.abs**`Series.abs()`

Return an object with absolute value taken. Only applicable to objects that are all numeric

**Returns** abs: type of caller**pandas.Series.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.autocorr**`Series.autocorr()`

Lag-1 autocorrelation

**Returns** autocorr : float**pandas.Series.between**`Series.between (left, right, inclusive=True)`Return boolean Series equivalent to  $\text{left} \leq \text{series} \leq \text{right}$ . NA values will be treated as False**Parameters** left : scalar

Left boundary

**right** : scalar

Right boundary

**Returns** is\_between : Series**pandas.Series.clip**`Series.clip (lower=None, upper=None, out=None)`

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.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 a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

`numeric_only` : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns** `mad` : scalar or Series (if level specified)

**pandas.Series.max**

`Series.max` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

**Parameters** `axis` : {index (0)}

`skipna` : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

`level` : int, 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 `sort=False` is not defined.

**Returns** `modes` : Series (sorted)

### **pandas.Series.nunique**

`Series.nunique()`

Return count of unique elements in the Series

**Returns** `nunique` : int

### **pandas.Series.pct\_change**

`Series.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)`

Percent change over given number of periods

**Parameters** `periods` : int, default 1

Periods to shift for forming percent change

**fill\_method** : str, default 'pad'

How to handle NAs before computing percent changes

**limit** : int, default None

The number of consecutive NAs to fill before stopping

**freq** : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay())

**Returns** `chg` : same type as caller

### **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.quantile

`Series.quantile(q=0.5)`

Return value at the given quantile, a la `scoreatpercentile` in `scipy.stats`

**Parameters** `q` : quantile

$0 \leq q \leq 1$

**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

## 27.3.8 Reindexing / Selection / Label manipulation

|                                                                 |                                                                                    |
|-----------------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>Series.align</code> (other[, join, axis, level, ...])     | Align two object on their axes with the                                            |
| <code>Series.drop</code> (labels[, axis, level, inplace])       | Return new object with labels in requested axis removed                            |
| <code>Series.first</code> (offset)                              | Convenience method for subsetting initial periods of time series data              |
| <code>Series.head</code> ([n])                                  | Returns first n rows                                                               |
| <code>Series.idxmax</code> ([axis, out, skipna])                | Index of first occurrence of maximum of values.                                    |
| <code>Series.idxmin</code> ([axis, out, skipna])                | Index of first occurrence of minimum of values.                                    |
| <code>Series.isin</code> (values)                               | Return a boolean <code>Series</code> showing whether each element                  |
| <code>Series.last</code> (offset)                               | Convenience method for subsetting final periods of time series data                |
| <code>Series.reindex</code> ([index])                           | Conform Series to new index with optional filling logic, placing                   |
| <code>Series.reindex_like</code> (other[, method, copy, limit]) | return an object with matching indicies to myself                                  |
| <code>Series.rename</code> ([index])                            | Alter axes input function or functions.                                            |
| <code>Series.reset_index</code> ([level, drop, name, inplace])  | Analogous to the <code>DataFrame.reset_index</code> function, see docstring there. |
| <code>Series.select</code> (crit[, axis])                       | Return data corresponding to axis labels matching criteria                         |
| <code>Series.take</code> (indices[, axis, convert])             | Analogous to <code>ndarray.take</code> , return Series corresponding to requested  |
| <code>Series.tail</code> ([n])                                  | Returns last n rows                                                                |
| <code>Series.truncate</code> ([before, after, axis, copy])      | Truncates a sorted <code>NDFrame</code> 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.first` (*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset

**Parameters** **offset** : string, DateOffset, dateutil.relativedelta

**Returns** `subset` : type of caller

### Examples

`ts.last('10D')` -> First 10 days

### pandas.Series.head

`Series.head(n=5)`  
Returns first n rows

### pandas.Series.idxmax

`Series.idxmax(axis=None, out=None, skipna=True)`  
Index of first occurrence of maximum of values.

**Parameters** `skipna` : boolean, default True

Exclude NA/null values

**Returns** `idxmax` : Index of 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.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`:

```
from pandas import Series
s = Series(list('abc'))
s.isin('a')
```

Instead, turn a single string into a list of one element:

```
from pandas import Series
s = Series(list('abc'))
s.isin(['a'])
```

**Returns** `isin` : `Series` (bool dtype)

**Raises** `TypeError`

- If `values` is a string

**See Also:**

`pandas.DataFrame.isin`

## 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.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

```
>>> 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 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

**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

## 27.3.9 Missing data handling

|                                                             |                                                    |
|-------------------------------------------------------------|----------------------------------------------------|
| <code>Series.dropna([axis, inplace])</code>                 | Return Series without null values                  |
| <code>Series.fillna([value, method, axis, ...])</code>      | Fill NA/NaN values using the specified method      |
| <code>Series.interpolate([method, axis, limit, ...])</code> | 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** : bool (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', '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.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 an 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 1 2 2 3 3 dtype: float64
```

## 27.3.10 Reshaping, sorting

|                                                          |                                                             |
|----------------------------------------------------------|-------------------------------------------------------------|
| <code>Series.argsort([axis, kind, order])</code>         | Overrides ndarray.argsort.                                  |
| <code>Series.order([na_last, ascending, kind])</code>    | Sorts Series object, by value, maintaining index-value link |
| <code>Series.reorder_levels(order)</code>                | Rearrange index levels using input order.                   |
| <code>Series.sort([axis, kind, order, ascending])</code> | Sort values and index labels by value, in place.            |
| <code>Series.sort_index([ascending])</code>              | Sort object by labels (along an axis)                       |
| <code>Series.sortlevel([level, ascending])</code>        | Sort Series with MultiIndex by chosen level. Data will be   |
| <code>Series.swaplevel(i, j[, copy])</code>              | Swap levels i and j in a MultiIndex                         |
| <code>Series.unstack([level])</code>                     | Unstack, a.k.a.                                             |

### pandas.Series.argsort

`Series.argsort` (*axis=0, kind='quicksort', order=None*)

Overrides ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values

**Parameters** **axis** : int (can only be zero)

**kind** : { 'mergesort', 'quicksort', 'heapsort' }, default 'quicksort'

Choice of sorting algorithm. See `np.sort` for more information. 'mergesort' is the only stable algorithm

**order** : ignored

**Returns** **argsorted** : Series, with -1 indicated where nan values are present

### pandas.Series.order

`Series.order` (*na\_last=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:**

`pandas.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.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

```
>>> 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.
```

## 27.3.11 Combining / joining / merging

---

|                                                                |                                                              |
|----------------------------------------------------------------|--------------------------------------------------------------|
| <code>Series.append(to_append[, verify_integrity])</code>      | Concatenate two or more Series. The indexes must not overlap |
| <code>Series.replace([to_replace, value, inplace, ...])</code> | Replace values given in 'to_replace' with 'value'.           |
| <code>Series.update(other)</code>                              | Modify Series in place using non-NA values from passed       |

---

## 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.replace

`Series.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)`

Replace values given in 'to\_replace' with 'value'.

**Parameters** `to_replace` : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - str: string exactly matching *to\_replace* will be replaced with *value*
  - regex: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., `{ 'a': { 'b': nan } }`, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- None:
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also None then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If `True`, in place. Note: this will modify any other views on this object (e.g. a column 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`

## 27.3.12 Time series-related

|                                                            |                                                                             |
|------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>Series.asfreq(freq[, method, how, normalize])</code> | Convert all TimeSeries inside to specified frequency using DateOffset       |
| <code>Series.asof(where)</code>                            | Return last good (non-NaN) value in TimeSeries if value is NaN for          |
| <code>Series.shift([periods, freq, axis])</code>           | Shift index by desired number of periods with an optional time freq         |
| <code>Series.first_valid_index()</code>                    | Return label for first non-NA/null value                                    |
| <code>Series.last_valid_index()</code>                     | Return label for last non-NA/null value                                     |
| <code>Series.weekday</code>                                |                                                                             |
| <code>Series.resample(rule[, how, axis, ...])</code>       | Convenience method for frequency conversion and resampling of regular time- |
| <code>Series.tz_convert(tz[, copy])</code>                 | Convert TimeSeries to target time zone                                      |
| <code>Series.tz_localize(tz[, copy, infer_dst])</code>     | Localize tz-naive TimeSeries to target time zone                            |

### pandas.Series.asfreq

`Series.asfreq(freq, method=None, how=None, normalize=False)`

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters** **freq** : DateOffset object, or string

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill methdo

**how** : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

**normalize** : bool, default False

Whether to reset output index to midnight

**Returns** **converted** : type of caller

### pandas.Series.asof

`Series.asof(where)`

Return last good (non-NaN) value in TimeSeries if value is NaN for requested date.

If there is no good value, NaN is returned.

**Parameters** **where** : date or array of dates

**Returns** value or NaN

#### Notes

Dates are assumed to be sorted

### pandas.Series.shift

`Series.shift(periods=1, freq=None, axis=0, **kws)`

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 is not realigned

### pandas.Series.first\_valid\_index

`Series.first_valid_index()`

Return label for first non-NA/null value

### pandas.Series.last\_valid\_index

`Series.last_valid_index()`

Return label for last non-NA/null value

### pandas.Series.weekday

`Series.weekday`

### pandas.Series.resample

`Series.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0)`

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** **rule** : 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, fill\_method for upsampling, default None

**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.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

## 27.3.13 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.

|                                                             |                                                                             |
|-------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>StringMethods.cat([others, sep, na_rep])</code>       | Concatenate arrays of strings with given separator                          |
| <code>StringMethods.center(width)</code>                    | “Center” strings, filling left and right side with additional whitespace    |
| <code>StringMethods.contains(pat[, case, flags, na])</code> | Check whether given pattern is contained in each string in the array        |
| <code>StringMethods.count(pat[, flags])</code>              | Count occurrences of pattern in each string                                 |
| <code>StringMethods.decode(encoding[, errors])</code>       | Decode character string to unicode using indicated encoding                 |
| <code>StringMethods.encode(encoding[, errors])</code>       | Encode character string to some other encoding using indicated encoding     |
| <code>StringMethods.endswith(pat[, na])</code>              | Return boolean array indicating whether each string ends with passed        |
| <code>StringMethods.extract(pat[, flags])</code>            | Find groups in each string using passed regular expression                  |
| <code>StringMethods.findall(pat[, flags])</code>            | Find all occurrences of pattern or regular expression                       |
| <code>StringMethods.get(i)</code>                           | Extract element from lists, tuples, or strings in each element in the array |
| <code>StringMethods.join(sep)</code>                        | Join lists contained as elements in array, a la str.join                    |

Continued on next page

Table 27.34 – continued from previous page

|                                                               |                                                                                           |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>StringMethods.len()</code>                              | Compute length of each string in array.                                                   |
| <code>StringMethods.lower()</code>                            | Convert strings in array to lowercase                                                     |
| <code>StringMethods.lstrip([to_strip])</code>                 | Strip whitespace (including newlines) from left side of each string in the                |
| <code>StringMethods.match(pat[, flags])</code>                | Deprecated: Find groups in each string using passed regular expression.                   |
| <code>StringMethods.pad(width[, side])</code>                 | Pad strings with whitespace                                                               |
| <code>StringMethods.repeat(repeats)</code>                    | Duplicate each string in the array by indicated number of times                           |
| <code>StringMethods.replace(pat, repl[, n, case, ...])</code> | Replace                                                                                   |
| <code>StringMethods.rstrip([to_strip])</code>                 | Strip whitespace (including newlines) from right side of each string in the               |
| <code>StringMethods.slice([start, stop, step])</code>         | Slice substrings from each element in array                                               |
| <code>StringMethods.slice_replace([i, j])</code>              | Slice substrings from each element in array                                               |
| <code>StringMethods.split([pat, n])</code>                    | Split each string (a la <code>re.split</code> ) in array by given pattern, propagating NA |
| <code>StringMethods.startswith(pat[, na])</code>              | Return boolean array indicating whether each string starts with passed                    |
| <code>StringMethods.strip([to_strip])</code>                  | Strip whitespace (including newlines) from each string in the array                       |
| <code>StringMethods.title()</code>                            | Convert strings to titlecased version                                                     |
| <code>StringMethods.upper()</code>                            | Convert strings in array to uppercase                                                     |

### **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*)

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.

### **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

### **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)

#### Notes

Compare to the string method `match`, which returns `re.match` objects.

### 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** `matches` : boolean array (if `as_indexer=True`)**matches** : array of tuples (if `as_indexer=False`, default but deprecated)**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.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**`StringMethods.strip(to_strip=None)`

Strip whitespace (including newlines) from each string in the array

**Parameters** `to_strip` : str or unicode**Returns** `stripped` : array**pandas.core.strings.StringMethods.title**`StringMethods.title()`

Convert strings to titlecased version

**Returns** `titled` : array**pandas.core.strings.StringMethods.upper**`StringMethods.upper()`

Convert strings in array to uppercase

**Returns** `uppercase` : array**27.3.14 Plotting**

---

|                                                           |                                                                     |
|-----------------------------------------------------------|---------------------------------------------------------------------|
| <code>Series.hist([by, ax, grid, xlabelsize, ...])</code> | Draw histogram of the input series using matplotlib                 |
| <code>Series.plot(series[, label, kind, ...])</code>      | 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 objectIf 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

**kwargs** : keywords

To be passed to the actual plotting function

## Notes

See matplotlib documentation online for more on this

## 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*, *\*\*kwargs*)

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

## 27.3.15 Serialization / IO / Conversion

|                                                             |                                                                        |
|-------------------------------------------------------------|------------------------------------------------------------------------|
| <code>Series.from_csv(path[, sep, parse_dates, ...])</code> | Read delimited file into Series                                        |
| <code>Series.to_pickle(path)</code>                         | Pickle (serialize) object to input file path                           |
| <code>Series.to_csv(path[, index, sep, na_rep, ...])</code> | Write Series to a comma-separated values (csv) file                    |
| <code>Series.to_dict()</code>                               | Convert Series to {label -> value} dict                                |
| <code>Series.to_frame([name])</code>                        | Convert Series to DataFrame                                            |
| <code>Series.to_hdf(path_or_buf, key, **kwargs)</code>      | activate the HDFStore                                                  |
| <code>Series.to_json([path_or_buf, orient, ...])</code>     | Convert the object to a JSON string.                                   |
| <code>Series.to_sparse([kind, fill_value])</code>           | Convert Series to SparseSeries                                         |
| <code>Series.to_string([buf, na_rep, ...])</code>           | Render a string representation of the Series                           |
| <code>Series.to_clipboard([excel, sep])</code>              | Attempt to write text representation of object to the system clipboard |

### pandas.Series.from\_csv

**classmethod** `Series.from_csv` (*path*, *sep*=' ', *parse\_dates*=True, *header*=None, *index\_col*=0, *encoding*=None)

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

**Returns** `y` : Series

### **pandas.Series.to\_pickle**

`Series.to_pickle(path)`

Pickle (serialize) object to input file path

**Parameters** `path` : string

File path

### **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\_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, an identifier for the group in the store

**mode** : optional, {'a', 'w', 'r', 'r+'}, default 'a'

'r' Read-only; no data can be modified.

'w' Write; a new file is created (an existing file with the same name would be deleted).

'a' Append; an existing file is opened for reading and writing, and if the file does not exist it is created.

'r+' It is similar to 'a', but the file must already exist.

**format** : 'fixed(f)|table(t)', default is 'fixed'

**fixed(f)** [Fixed format] Fast writing/reading. Not-appendable, nor searchable

**table(t)** [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

**append** : boolean, default False

For Table formats, append the input data to the existing

**complevel** : int, 1-9, default 0

If a complib is specified compression will be applied where possible

**complib** : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

**fletcher32** : bool, default False

If applying compression use the fletcher32 checksum

## pandas.Series.to\_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** : type of date conversion, epoch or iso

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)

**Returns** **formatted** : string (if not buffer passed)

## pandas.Series.to\_clipboard

`Series.to_clipboard(excel=None, sep=None, **kwargs)`

Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

**Parameters** **excel** : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard

**sep** : optional, defaults to tab

**other keywords are passed to to\_csv**

### Notes

#### Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

## 27.4 DataFrame

### 27.4.1 Constructor

---

`DataFrame([data, index, columns, dtype, copy])` Two-dimensional size-mutable, potentially heterogeneous tabular data structure v

---

## pandas.DataFrame

**class** `pandas.DataFrame` (*data=None, index=None, columns=None, dtype=None, copy=False*)

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure

**Parameters** **data** : numpy ndarray (structured or homogeneous), dict, or DataFrame

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

```
>>> 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

|                        |                             |
|------------------------|-----------------------------|
| <code>T</code>         | Transpose index and columns |
| <code>at</code>        |                             |
| <code>axes</code>      |                             |
| Continued on next page |                             |

Table 27.38 – continued from previous page

|                     |                                                                  |
|---------------------|------------------------------------------------------------------|
| <code>blocks</code> | Internal property, property synonym for <code>as_blocks()</code> |
| <code>dtypes</code> |                                                                  |
| <code>empty</code>  | True if NDFrame is entirely empty [no items]                     |
| <code>ftypes</code> |                                                                  |
| <code>iat</code>    |                                                                  |
| <code>iloc</code>   |                                                                  |
| <code>ix</code>     |                                                                  |
| <code>loc</code>    |                                                                  |
| <code>ndim</code>   | Number of axes / array dimensions                                |
| <code>shape</code>  |                                                                  |
| <code>values</code> | Numpy representation of NDFrame                                  |

**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`

**pandas.DataFrame.empty**

`DataFrame.empty`  
True if NDFrame is entirely empty [no items]

**pandas.DataFrame.ftypes**

`DataFrame.ftypes`

**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

|         |  |
|---------|--|
| is_copy |  |
|---------|--|

**Methods**

|                                                                |                                                                                 |
|----------------------------------------------------------------|---------------------------------------------------------------------------------|
| <code>abs()</code>                                             | Return an object with absolute value taken.                                     |
| <code>add(other[, axis, level, fill_value])</code>             | Binary operator add with support to substitute a fill_value for missing data in |
| <code>add_prefix(prefix)</code>                                | Concatenate prefix string with panel items names.                               |
| <code>add_suffix(suffix)</code>                                | Concatenate suffix string with panel items names                                |
| <code>align(other[, join, axis, level, copy, ...])</code>      | Align two object on their axes with the                                         |
| <code>all([axis, bool_only, skipna, level])</code>             | Return whether all elements are True over requested axis.                       |
| <code>any([axis, bool_only, skipna, level])</code>             | Return whether any element is True over requested axis.                         |
| <code>append(other[, ignore_index, verify_integrity])</code>   | Append columns of other to end of this frame's columns and index, returning a   |
| <code>apply(func[, axis, broadcast, raw, reduce, args])</code> | Applies function along input axis of DataFrame.                                 |
| <code>applymap(func)</code>                                    | Apply a function to a DataFrame that is intended to operate                     |
| <code>as_blocks([columns])</code>                              | Convert the frame to a dict of dtype -> Constructor Types that each has         |
| <code>as_matrix([columns])</code>                              | Convert the frame to its Numpy-array matrix representation. Columns             |
| <code>asfreq(freq[, method, how, normalize])</code>            | Convert all TimeSeries inside to specified frequency using DateOffset           |
| <code>astype(dtype[, copy, raise_on_error])</code>             | Cast object to input numpy.dtype                                                |
| <code>at_time(time[, asof])</code>                             | Select values at particular time of day (e.g.                                   |
| <code>between_time(start_time, end_time[, ...])</code>         | Select values between particular times of the day (e.g., 9:00-9:30 AM)          |

Continued c

Table 27.39 – continued from previous page

|                                                             |                                                                                                              |
|-------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <code>bfill([axis, inplace, limit, downcast])</code>        | Synonym for <code>NDFrame.fillna(method='bfill')</code>                                                      |
| <code>bool()</code>                                         | Return the bool of a single element <code>PandasObject</code>                                                |
| <code>boxplot([column, by, ax, fontsize, rot, grid])</code> | Make a box plot from <code>DataFrame</code> column/columns optionally grouped                                |
| <code>clip([lower, upper, out])</code>                      | Trim values at input threshold(s)                                                                            |
| <code>clip_lower(threshold)</code>                          | Return copy of the input with values below given value truncated                                             |
| <code>clip_upper(threshold)</code>                          | Return copy of input with values above given value truncated                                                 |
| <code>combine(other, func[, fill_value, overwrite])</code>  | Add two <code>DataFrame</code> objects and do not propagate NaN values, so if for a                          |
| <code>combineAdd(other)</code>                              | Add two <code>DataFrame</code> objects and do not propagate                                                  |
| <code>combineMult(other)</code>                             | Multiply two <code>DataFrame</code> objects and do not propagate NaN values, so if                           |
| <code>combine_first(other)</code>                           | Combine two <code>DataFrame</code> objects and default to non-null values in frame                           |
| <code>compound([axis, skipna, level])</code>                | Return the compound percentage of the values for the requested axis                                          |
| <code>consolidate([inplace])</code>                         | Compute <code>NDFrame</code> with “consolidated” internals (data of each dtype                               |
| <code>convert_objects([convert_dates, ...])</code>          | Attempt to infer better dtype for object columns                                                             |
| <code>copy([deep])</code>                                   | Make a copy of this object                                                                                   |
| <code>corr([method, min_periods])</code>                    | Compute pairwise correlation of columns, excluding NA/null values                                            |
| <code>corrwith(other[, axis, drop])</code>                  | Compute pairwise correlation between rows or columns of two <code>DataFrame</code>                           |
| <code>count([axis, level, numeric_only])</code>             | Return Series with number of non-NA/null observations over requested                                         |
| <code>cov([min_periods])</code>                             | Compute pairwise covariance of columns, excluding NA/null values                                             |
| <code>cummax([axis, dtype, out, skipna])</code>             | Return cumulative max over requested axis.                                                                   |
| <code>cummin([axis, dtype, out, skipna])</code>             | Return cumulative min over requested axis.                                                                   |
| <code>cumprod([axis, dtype, out, skipna])</code>            | Return cumulative prod over requested axis.                                                                  |
| <code>cumsum([axis, dtype, out, skipna])</code>             | Return cumulative sum over requested axis.                                                                   |
| <code>delevel(*args, **kwargs)</code>                       |                                                                                                              |
| <code>describe([percentile_width])</code>                   | Generate various summary statistics of each column, excluding                                                |
| <code>diff([periods])</code>                                | 1st discrete difference of object                                                                            |
| <code>div(other[, axis, level, fill_value])</code>          | Binary operator <code>truediv</code> with support to substitute a <code>fill_value</code> for missing data i |
| <code>divide(other[, axis, level, fill_value])</code>       | Binary operator <code>truediv</code> with support to substitute a <code>fill_value</code> for missing data i |
| <code>dot(other)</code>                                     | Matrix multiplication with <code>DataFrame</code> or <code>Series</code> objects                             |
| <code>drop(labels[, axis, level, inplace])</code>           | Return new object with labels in requested axis removed                                                      |
| <code>drop_duplicates([cols, take_last, inplace])</code>    | Return <code>DataFrame</code> with duplicate rows removed, optionally only                                   |
| <code>dropna([axis, how, thresh, subset, inplace])</code>   | Return object with labels on given axis omitted where alternately any                                        |
| <code>duplicated([cols, take_last])</code>                  | Return boolean Series denoting duplicate rows, optionally only                                               |
| <code>eq(other[, axis, level])</code>                       | Wrapper for flexible comparison methods <code>eq</code>                                                      |
| <code>eval(expr, **kwargs)</code>                           | Evaluate an expression in the context of the calling <code>DataFrame</code>                                  |
| <code>ffill([axis, inplace, limit, downcast])</code>        | Synonym for <code>NDFrame.fillna(method='ffill')</code>                                                      |
| <code>fillna([value, method, axis, inplace, ...])</code>    | Fill NA/NaN values using the specified method                                                                |
| <code>filter([items, like, regex, axis])</code>             | Restrict the info axis to set of items or wildcard                                                           |
| <code>first(offset)</code>                                  | Convenience method for subsetting initial periods of time series data                                        |
| <code>first_valid_index()</code>                            | Return label for first non-NA/null value                                                                     |
| <code>floordiv(other[, axis, level, fill_value])</code>     | Binary operator <code>floordiv</code> with support to substitute a <code>fill_value</code> for missing data  |
| <code>from_csv(path[, header, sep, index_col, ...])</code>  | Read delimited file into <code>DataFrame</code>                                                              |
| <code>from_dict(data[, orient, dtype])</code>               | Construct <code>DataFrame</code> from dict of array-like or dicts                                            |
| <code>from_items(items[, columns, orient])</code>           | Convert (key, value) pairs to <code>DataFrame</code> . The keys will be the axis                             |
| <code>from_records(data[, index, exclude, ...])</code>      | Convert structured or record ndarray to <code>DataFrame</code>                                               |
| <code>ge(other[, axis, level])</code>                       | Wrapper for flexible comparison methods <code>ge</code>                                                      |
| <code>get(key[, default])</code>                            | Get item from object for given key ( <code>DataFrame</code> column, Panel slice,                             |
| <code>get_dtype_counts()</code>                             | return the counts of dtypes in this frame                                                                    |
| <code>get_ftype_counts()</code>                             | return the counts of ftypes in this frame                                                                    |
| <code>get_value(index, col)</code>                          | Quickly retrieve single value at passed column and index                                                     |
| <code>get_values()</code>                                   | same as values (but handles sparseness conversions)                                                          |

Continued c

Table 27.39 – continued from previous page

|                                                               |                                                                                   |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>groupby([by, axis, level, as_index, sort, ...])</code>  | Group series using mapper (dict or key function, apply given function)            |
| <code>gt(other[, axis, level])</code>                         | Wrapper for flexible comparison methods gt                                        |
| <code>head([n])</code>                                        | Returns first n rows                                                              |
| <code>hist(data[, column, by, grid, xlabelsize, ...])</code>  | Draw Histogram the DataFrame's series using matplotlib / pylab.                   |
| <code>icol(i)</code>                                          |                                                                                   |
| <code>idxmax([axis, skipna])</code>                           | Return index of first occurrence of maximum over requested axis.                  |
| <code>idxmin([axis, skipna])</code>                           | Return index of first occurrence of minimum over requested axis.                  |
| <code>iget_value(i, j)</code>                                 |                                                                                   |
| <code>info([verbose, buf, max_cols])</code>                   | Concise summary of a DataFrame.                                                   |
| <code>insert(loc, column, value[, allow_duplicates])</code>   | Insert column into DataFrame at specified location.                               |
| <code>interpolate([method, axis, limit, inplace, ...])</code> | Interpolate values according to different methods.                                |
| <code>irow(i[, copy])</code>                                  |                                                                                   |
| <code>isin(values)</code>                                     | Return boolean DataFrame showing whether each element in the                      |
| <code>isnull()</code>                                         | Return a boolean same-sized object indicating if the values are null              |
| <code>iteritems()</code>                                      | Iterator over (column, series) pairs                                              |
| <code>iterkv(*args, **kwargs)</code>                          | iteritems alias used to get around 2to3. Deprecated                               |
| <code>iterrows()</code>                                       | Iterate over rows of DataFrame as (index, Series) pairs.                          |
| <code>itertuples([index])</code>                              | Iterate over rows of DataFrame as tuples, with index value                        |
| <code>join(other[, on, how, lsuffix, rsuffix, sort])</code>   | Join columns with other DataFrame either on index or on a key                     |
| <code>keys()</code>                                           | Get the 'info axis' (see Indexing for more)                                       |
| <code>kurt([axis, skipna, level, numeric_only])</code>        | Return unbiased kurtosis over requested axis                                      |
| <code>kurtosis([axis, skipna, level, numeric_only])</code>    | Return unbiased kurtosis over requested axis                                      |
| <code>last(offset)</code>                                     | Convenience method for subsetting final periods of time series data               |
| <code>last_valid_index()</code>                               | Return label for last non-NA/null value                                           |
| <code>le(other[, axis, level])</code>                         | Wrapper for flexible comparison methods le                                        |
| <code>load(path)</code>                                       | Deprecated.                                                                       |
| <code>lookup(row_labels, col_labels)</code>                   | Label-based "fancy indexing" function for DataFrame.                              |
| <code>lt(other[, axis, level])</code>                         | Wrapper for flexible comparison methods lt                                        |
| <code>mad([axis, skipna, level])</code>                       | Return the mean absolute deviation of the values for the requested axis           |
| <code>mask(cond)</code>                                       | Returns copy whose values are replaced with nan if the                            |
| <code>max([axis, skipna, level, numeric_only])</code>         | This method returns the maximum of the values in the object.                      |
| <code>mean([axis, skipna, level, numeric_only])</code>        | Return the mean of the values for the requested axis                              |
| <code>median([axis, skipna, level, numeric_only])</code>      | Return the median of the values for the requested axis                            |
| <code>merge(right[, how, on, left_on, right_on, ...])</code>  | Merge DataFrame objects by performing a database-style join operation by          |
| <code>min([axis, skipna, level, numeric_only])</code>         | This method returns the minimum of the values in the object.                      |
| <code>mod(other[, axis, level, fill_value])</code>            | Binary operator mod with support to substitute a fill_value for missing data in   |
| <code>mode([axis, numeric_only])</code>                       | Gets the mode of each element along the axis selected.                            |
| <code>mul(other[, axis, level, fill_value])</code>            | Binary operator mul with support to substitute a fill_value for missing data in   |
| <code>multiply(other[, axis, level, fill_value])</code>       | Binary operator mul with support to substitute a fill_value for missing data in   |
| <code>ne(other[, axis, level])</code>                         | Wrapper for flexible comparison methods ne                                        |
| <code>notnull()</code>                                        | Return a boolean same-sized object indicating if the values are                   |
| <code>pct_change([periods, fill_method, limit, freq])</code>  | Percent change over given number of periods                                       |
| <code>pivot([index, columns, values])</code>                  | Reshape data (produce a "pivot" table) based on column values.                    |
| <code>pivot_table(data[, values, rows, cols, ...])</code>     | Create a spreadsheet-style pivot table as a DataFrame. The levels in the          |
| <code>plot([frame, x, y, subplots, sharex, ...])</code>       | Make line, bar, or scatter plots of DataFrame series with the index on the x-axis |
| <code>pop(item)</code>                                        | Return item and drop from frame.                                                  |
| <code>pow(other[, axis, level, fill_value])</code>            | Binary operator pow with support to substitute a fill_value for missing data in   |
| <code>prod([axis, skipna, level, numeric_only])</code>        | Return the product of the values for the requested axis                           |
| <code>product([axis, skipna, level, numeric_only])</code>     | Return the product of the values for the requested axis                           |
| <code>quantile([q, axis, numeric_only])</code>                | Return values at the given quantile over requested axis, a la                     |

Continued c

Table 27.39 – continued from previous page

|                                                                |                                                                                    |
|----------------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>query(expr, **kwargs)</code>                             | Query the columns of a frame with a boolean expression.                            |
| <code>radd(other[, axis, level, fill_value])</code>            | Binary operator radd with support to substitute a fill_value for missing data in   |
| <code>rank([axis, numeric_only, method, ...])</code>           | Compute numerical data ranks (1 through n) along axis.                             |
| <code>rdiv(other[, axis, level, fill_value])</code>            | Binary operator rtruediv with support to substitute a fill_value for missing data  |
| <code>reindex([index, columns])</code>                         | Conform DataFrame to new index with optional filling logic, placing                |
| <code>reindex_axis(labels[, axis, method, level, ...])</code>  | Conform input object to new index with optional filling logic,                     |
| <code>reindex_like(other[, method, copy, limit])</code>        | return an object with matching indicies to myself                                  |
| <code>rename([index, columns])</code>                          | Alter axes input function or functions.                                            |
| <code>rename_axis(mapper[, axis, copy, inplace])</code>        | Alter index and / or columns using input function or functions.                    |
| <code>reorder_levels(order[, axis])</code>                     | Rearrange index levels using input order.                                          |
| <code>replace([to_replace, value, inplace, limit, ...])</code> | Replace values given in 'to_replace' with 'value'.                                 |
| <code>resample(rule[, how, axis, fill_method, ...])</code>     | Convenience method for frequency conversion and resampling of regular time-        |
| <code>reset_index([level, drop, inplace, ...])</code>          | For DataFrame with multi-level index, return new DataFrame with                    |
| <code>rfloordiv(other[, axis, level, fill_value])</code>       | Binary operator rfloordiv with support to substitute a fill_value for missing data |
| <code>rmod(other[, axis, level, fill_value])</code>            | Binary operator rmod with support to substitute a fill_value for missing data in   |
| <code>rmul(other[, axis, level, fill_value])</code>            | Binary operator rmul with support to substitute a fill_value for missing data in   |
| <code>rpow(other[, axis, level, fill_value])</code>            | Binary operator rpow with support to substitute a fill_value for missing data in   |
| <code>rsub(other[, axis, level, fill_value])</code>            | Binary operator rsub with support to substitute a fill_value for missing data in   |
| <code>rtruediv(other[, axis, level, fill_value])</code>        | Binary operator rtruediv with support to substitute a fill_value for missing data  |
| <code>save(path)</code>                                        | Deprecated.                                                                        |
| <code>select(crit[, axis])</code>                              | Return data corresponding to axis labels matching criteria                         |
| <code>set_index(keys[, drop, append, inplace, ...])</code>     | Set the DataFrame index (row labels) using one or more existing                    |
| <code>set_value(index, col, value)</code>                      | Put single value at passed column and index                                        |
| <code>shift([periods, freq, axis])</code>                      | Shift index by desired number of periods with an optional time freq                |
| <code>skew([axis, skipna, level, numeric_only])</code>         | Return unbiased skew over requested axis                                           |
| <code>sort([columns, column, axis, ascending, inplace])</code> | Sort DataFrame either by labels (along either axis) or by the values in            |
| <code>sort_index([axis, by, ascending, inplace, kind])</code>  | Sort DataFrame either by labels (along either axis) or by the values in            |
| <code>sortlevel([level, axis, ascending, inplace])</code>      | Sort multilevel index by chosen axis and primary level.                            |
| <code>squeeze()</code>                                         | squeeze length 1 dimensions                                                        |
| <code>stack([level, dropna])</code>                            | Pivot a level of the (possibly hierarchical) column labels, returning a            |
| <code>std([axis, skipna, level, ddof])</code>                  | Return unbiased standard deviation over requested axis                             |
| <code>sub(other[, axis, level, fill_value])</code>             | Binary operator sub with support to substitute a fill_value for missing data in    |
| <code>subtract(other[, axis, level, fill_value])</code>        | Binary operator sub with support to substitute a fill_value for missing data in    |
| <code>sum([axis, skipna, level, numeric_only])</code>          | Return the sum of the values for the requested axis                                |
| <code>swapaxes(axis1, axis2[, copy])</code>                    | Interchange axes and swap values axes appropriately                                |
| <code>swaplevel(i, j[, axis])</code>                           | Swap levels i and j in a MultiIndex on a particular axis                           |
| <code>tail([n])</code>                                         | Returns last n rows                                                                |
| <code>take(indices[, axis, convert, is_copy])</code>           | Analogous to ndarray.take                                                          |
| <code>to_clipboard([excel, sep])</code>                        | Attempt to write text representation of object to the system clipboard             |
| <code>to_csv(path_or_buf[, sep, na_rep, ...])</code>           | Write DataFrame to a comma-separated values (csv) file                             |
| <code>to_dense()</code>                                        | Return dense representation of NDFrame (as opposed to sparse)                      |
| <code>to_dict([outtype])</code>                                | Convert DataFrame to dictionary.                                                   |
| <code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code> | Write DataFrame to a excel sheet                                                   |
| <code>to_gbq(destination_table[, schema, ...])</code>          | Write a DataFrame to a Google BigQuery table.                                      |
| <code>to_hdf(path_or_buf, key, **kwargs)</code>                | activate the HDFStore                                                              |
| <code>to_html([buf, columns, col_space, colSpace, ...])</code> | Render a DataFrame as an HTML table.                                               |
| <code>to_json([path_or_buf, orient, date_format, ...])</code>  | Convert the object to a JSON string.                                               |
| <code>to_latex([buf, columns, col_space, ...])</code>          | Render a DataFrame to a tabular environment table.                                 |
| <code>to_msgpack([path_or_buf])</code>                         | msgpack (serialize) object to input file path                                      |
| <code>to_panel()</code>                                        | Transform long (stacked) format (DataFrame) into wide (3D, Panel)                  |

Continued c

Table 27.39 – continued from previous page

|                                                              |                                                                                  |
|--------------------------------------------------------------|----------------------------------------------------------------------------------|
| <code>to_period([freq, axis, copy])</code>                   | Convert DataFrame from DatetimeIndex to PeriodIndex with desired                 |
| <code>to_pickle(path)</code>                                 | Pickle (serialize) object to input file path                                     |
| <code>to_records([index, convert_datetime64])</code>         | Convert DataFrame to record array. Index will be put in the                      |
| <code>to_sparse([fill_value, kind])</code>                   | Convert to SparseDataFrame                                                       |
| <code>to_sql(name, con[, flavor, if_exists])</code>          | Write records stored in a DataFrame to a SQL database.                           |
| <code>to_stata(fname[, convert_dates, ...])</code>           | A class for writing Stata binary dta files from array-like objects               |
| <code>to_string([buf, columns, col_space, ...])</code>       | Render a DataFrame to a console-friendly tabular output.                         |
| <code>to_timestamp([freq, how, axis, copy])</code>           | Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period               |
| <code>to_wide(*args, **kwargs)</code>                        |                                                                                  |
| <code>transpose()</code>                                     | Transpose index and columns                                                      |
| <code>truediv(other[, axis, level, fill_value])</code>       | Binary operator truediv with support to substitute a fill_value for missing data |
| <code>truncate([before, after, axis, copy])</code>           | Truncates a sorted NDFrame before and/or after some particular                   |
| <code>tshift([periods, freq, axis])</code>                   | Shift the time index, using the index's frequency if available                   |
| <code>tz_convert(tz[, axis, copy])</code>                    | Convert TimeSeries to target time zone. If it is time zone naive, it             |
| <code>tz_localize(tz[, axis, copy, infer_dst])</code>        | Localize tz-naive TimeSeries to target time zone                                 |
| <code>unstack([level])</code>                                | Pivot a level of the (necessarily hierarchical) index labels, returning          |
| <code>update(other[, join, overwrite, ...])</code>           | Modify DataFrame in place using non-NA values from passed                        |
| <code>var([axis, skipna, level, ddof])</code>                | Return unbiased variance over requested axis                                     |
| <code>where(cond[, other, inplace, axis, level, ...])</code> | Return an object of same shape as self and whose corresponding                   |
| <code>xs(key[, axis, level, copy, drop_level])</code>        | Returns a cross-section (row(s) or column(s)) from the 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 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.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=True*, *args=()*, *\*\*kws*)

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

**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, default True

Try to apply reduction procedures

**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

#### Examples

```
>>> 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 methdo

**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, **kws)`

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

**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

### **pandas.DataFrame.combineAdd**

`DataFrame.combineAdd` (*other*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

**Parameters** `other` : DataFrame

**Returns** DataFrame

### **pandas.DataFrame.combineMult**

`DataFrame.combineMult` (*other*)

Multiply two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

**Parameters** `other` : DataFrame

**Returns** DataFrame

### **pandas.DataFrame.combine\_first**

`DataFrame.combine_first` (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

**Parameters** `other` : DataFrame

**Returns** `combined` : DataFrame

## Examples

a's values prioritized, use values from b to fill holes:

```
>>> a.combine_first(b)
```

## pandas.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, 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

`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** **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)}

`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** `differed` : 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

**pandas.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

**pandas.DataFrame.dot**`DataFrame.dot` (*other*)

Matrix multiplication with DataFrame or Series objects

**Parameters** *other* : DataFrame or Series

**Returns** `dot_product` : DataFrame or Series

### **pandas.DataFrame.drop**

`DataFrame.drop` (*labels*, *axis=0*, *level=None*, *inplace=False*, *\*\*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.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

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = DataFrame(randn(10, 2), columns=list('ab'))
>>> df.eval('a + b')
>>> df.eval('c=a + b')
```

## pandas.DataFrame.ffill

`DataFrame.fffll` (*axis=0, inplace=False, limit=None, downcast=None*)  
Synonym for `NDFrame.fillna(method='ffill')`

## 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.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.first\_valid\_index

`DataFrame.first_valid_index()`

Return label for first non-NA/null value

### pandas.DataFrame.floordiv

`DataFrame.floordiv(other, axis='columns', level=None, fill_value=None)`

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*)

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)

**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 named associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

**coerce\_float** : boolean, default False

Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

**Returns** **df** : DataFrame

**pandas.DataFrame.ge**

`DataFrame.ge` (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods `ge`

### **pandas.DataFrame.get**

`DataFrame.get (key, default=None)`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found

**Parameters** `key` : object

**Returns** `value` : type of items contained in object

### **pandas.DataFrame.get\_dtype\_counts**

`DataFrame.get_dtype_counts ()`

return the counts of dtypes in this frame

### **pandas.DataFrame.get\_ftype\_counts**

`DataFrame.get_ftype_counts ()`

return the counts of ftypes in this frame

### **pandas.DataFrame.get\_value**

`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**

`DataFrame.get_values ()`

same as `values` (but handles sparseness conversions)

### **pandas.DataFrame.groupby**

`DataFrame.groupby (by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False)`

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns

**Parameters** `by` : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

`axis` : int, default 0

`level` : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels



**as\_index** : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as\_index=False is effectively “SQL-style” grouped output

**sort** : boolean, default True

Sort group keys. Get better performance by turning this off

**group\_keys** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**Returns** GroupBy object

### Examples

```
DataFrame 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, \*\*kws)  
Draw Histogram 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.icol**

`DataFrame.icol(i)`

### **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.iget\_value**

`DataFrame.iget_value` (*i, j*)

**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.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  $0 \leq \text{loc} \leq \text{len}(\text{columns})$

`column` : object

`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.interpld` with the order given both 'polynomial' and 'spline' require that you also specify an 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 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:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
 A B
0 True True
1 False False
2 True False
```

When `values` is a dict:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
 A B
0 True False # Note that B didn't match the 1 here.
1 False True
2 True True
```

When `values` is a Series or DataFrame:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
 A B
0 True False
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,

```
>>> 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

**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.keys

DataFrame.**keys**()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major\_axis for Panel.

### pandas.DataFrame.kurt

DataFrame.**kurt** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return unbiased kurtosis over requested axis 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.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

```
>>> A >>> B
 lkey value rkey value
0 foo 1 0 foo 5
1 bar 2 1 bar 6
2 baz 3 2 qux 7
3 foo 4 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**

```
>>> 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.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**

```
>>> 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)
>>> table
 small large
foo one 1 4
 two 6 NaN
bar one 5 4
 two 6 7

```

**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, \*\*kwargs*)

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

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)

$0 \leq q \leq 1$

**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

```
>>> 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

```
>>> 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

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

### `pandas.DataFrame.reindex_like`

`DataFrame.reindex_like` (*other*, *method=None*, *copy=True*, *limit=None*)  
return an object with matching indicies to myself

**Parameters** **other** : Object

**method** : string or None

**copy** : boolean, default True

**limit** : int, default None

Maximum size gap to forward or backward fill

**Returns** **reindexed** : same as input

### Notes

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

### `pandas.DataFrame.rename`

`DataFrame.rename` (*index=None*, *columns=None*, *\*\*kwargs*)

Alter axes input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

**Parameters** **index**, **columns** : dict-like or function, optional

Transformation to apply to that axis values

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new DataFrame. If True then value of copy is ignored.

**Returns** **renamed** : DataFrame (new object)

### `pandas.DataFrame.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 interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.

- **None:**
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If `value` is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill

**regex** : bool or same types as `to_replace`, default False

Whether to interpret `to_replace` and/or `value` as regular expressions. If this is True then `to_replace` must be a string. Otherwise, `to_replace` must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method** : string, optional, { 'pad', 'ffill', 'bfill' }

The method to use when for replacement, when `to_replace` is a list.

**Returns** **filled** : NDFrame

**Raises** **AssertionError**

- If `regex` is not a bool and `to_replace` is not None.

**TypeError**

- If `to_replace` is a dict and `value` is not a list, dict, ndarray, or Series
- If `to_replace` is None and `regex` is not compilable into a regular expression or is a list, dict, ndarray, or Series.

**ValueError**

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

**See Also:**

`NDFrame.reindex`, `NDFrame.asfreq`, `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** : 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, fill\_method for upsampling, default None

**closed** : {'right', 'left'}

Which side of bin interval is closed

**label** : {'right', 'left'}

Which bin edge label to label bucket with

**convention** : {'start', 'end', 's', 'e'}

**kind** : "period"/"timestamp"

**loffset**: timedelta

Adjust the resampled time labels

**limit**: int, default None

Maximum size gap to when reindexing with fill\_method

**base** : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

### pandas.DataFrame.reset\_index

`DataFrame.reset_index` (*level=None*, *drop=False*, *inplace=False*, *col\_level=0*, *col\_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level\_0', 'level\_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level\_0' (if 'index' is already taken) will be used.

**Parameters** **level** : int, str, tuple, or list, default None

Only remove the given levels from the index. Removes all levels by default

**drop** : boolean, default False

Do not try to insert index into dataframe columns. This resets the index to the default integer index.

**inplace** : boolean, default False

Modify the DataFrame in place (do not create a new object)

**col\_level** : int or str, default 0

If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

**col\_fill** : object, default ''

If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

**Returns** **resetted** : DataFrame

### **pandas.DataFrame.rfloordiv**

DataFrame.**rfloordiv**(*other*, *axis*='columns', *level*=None, *fill\_value*=None)

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.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.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.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 FalseCheck 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**

```

>>> 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** `index` : row label`col` : column label`value` : scalar value**Returns** `frame` : DataFrame

If label pair is contained, will be reference to calling DataFrame, otherwise a new object

### pandas.DataFrame.shift

DataFrame.**shift** (*periods=1, freq=None, axis=0, \*\*kws*)

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 is not realigned

### 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.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

```
>>> 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**

```
>>> s
 a b
one 1. 2.
two 3. 4.

>>> s.stack()
one a 1
 b 2
two a 3
 b 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

**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.swapaxes**

`DataFrame.swapaxes` (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

**Returns** **y** : same as input

### **pandas.DataFrame.swaplevel**

`DataFrame.swaplevel` (*i*, *j*, *axis=0*)

Swap levels *i* and *j* in a MultiIndex on a particular axis

**Parameters** **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

**Returns** **swapped** : type of caller (new object)

### **pandas.DataFrame.tail**

`DataFrame.tail` (*n=5*)

Returns last *n* rows

### **pandas.DataFrame.take**

`DataFrame.take` (*indices*, *axis=0*, *convert=True*, *is\_copy=True*)

Analogous to `ndarray.take`

**Parameters** **indices** : list / array of ints

**axis** : int, default 0

**convert** : translate neg to pos indices (default)

**is\_copy** : mark the returned frame as a copy

**Returns** **taken** : type of caller

**pandas.DataFrame.to\_clipboard**

`DataFrame.to_clipboard` (*excel=None, sep=None, \*\*kwargs*)

Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

**Parameters** `excel` : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard

`sep` : optional, defaults to tab

**other keywords are passed to `to_csv`**

**Notes****Requirements for your platform**

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

**pandas.DataFrame.to\_csv**

`DataFrame.to_csv` (*path\_or\_buf, sep=',', na\_rep='', float\_format=None, cols=None, header=True, index=True, index\_label=None, mode='w', nanRep=None, encoding=None, quoting=None, line\_terminator='n', chunksize=None, tupleize\_cols=False, date\_format=None, \*\*kws*)

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 *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:

```
>>> 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, an 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 compilib 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\_html

`DataFrame.to_html` (*buf=None, columns=None, col\_space=None, colSpace=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, justify=None, 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** : type of date conversion, epoch or iso

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

`conn` : 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', byte-order=None*)

A class for writing Stata binary dta files from array-like objects

**Parameters** `fname` : file path or buffer

Where to save the dta file.

`convert_dates` : dict

Dictionary mapping column of datetime types to the stata internal format that you want to use for the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either a number or a name.

`encoding` : str

Default is latin-1. Note that Stata does not support unicode.

`byteorder` : str

Can be ">", "<", "little", or "big". The default is None which uses `sys.byteorder`

### **Examples**

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

Or with dates

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

**pandas.DataFrame.to\_string**

```
DataFrame.to_string(buf=None, columns=None, col_space=None, 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_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, \*\*kws*)

Shift the time index, using the index's frequency if available

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, default None

Increment to use from datetools module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns** **shifted** : NDFrame

**Notes**

If freq is not specified then tries to use the freq or inferred\_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

**pandas.DataFrame.tz\_convert**

`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

**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**

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
... ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one a 1
 b 2
two a 3
 b 4
dtype: float64

>>> s.unstack(level=-1)
 a b
one 1 2
two 3 4

>>> s.unstack(level=0)
 one two
a 1 3
b 2 4
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one 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

Whether to perform the operation in place on the data

**axis** : alignment axis if needed, default None

**level** : alignment level if needed, default None

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

**Returns** **wh** : same type as caller

### **pandas.DataFrame.xs**

`DataFrame.xs` (*key*, *axis=0*, *level=None*, *copy=True*, *drop\_level=True*)

Returns a cross-section (row(s) or column(s)) from the 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 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
>>> df
 A B C
a 100 5 2
b 4 0 9
c 9 7 3

>>> df
 first second third A B C D
bar one 1 4 1 8 9
 two 1 7 5 5 0
baz one 1 6 6 8 0
 three 2 5 3 5 3
>>> df.xs(('baz', 'three'))
 A B C D
third
2 5 3 5 3
>>> df.xs('one', level=1)
 A B C D
first 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

```

## 27.4.2 Attributes and underlying data

### Axes

- **index:** row labels
- **columns:** column labels

|                                             |                                                                     |
|---------------------------------------------|---------------------------------------------------------------------|
| <code>DataFrame.as_matrix([columns])</code> | Convert the frame to its Numpy-array matrix representation. Columns |
| <code>DataFrame.dtypes</code>               |                                                                     |
| <code>DataFrame.get_dtype_counts()</code>   | return the counts of dtypes in this frame                           |
| <code>DataFrame.values</code>               | Numpy representation of NDFrame                                     |
| <code>DataFrame.axes</code>                 |                                                                     |
| <code>DataFrame.ndim</code>                 | Number of axes / array dimensions                                   |
| <code>DataFrame.shape</code>                |                                                                     |

### 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

## **pandas.DataFrame.get\_dtype\_counts**

DataFrame.get\_dtype\_counts()  
return the counts of dtypes in this frame

## **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

## **27.4.3 Conversion**

|                                                 |                                                                      |
|-------------------------------------------------|----------------------------------------------------------------------|
| DataFrame.astype(dtype[, copy, raise_on_error]) | Cast object to input numpy.dtype                                     |
| DataFrame.convert_objects([convert_dates, ...]) | Attempt to infer better dtype for object columns                     |
| DataFrame.copy([deep])                          | Make a copy of this object                                           |
| DataFrame.isnull()                              | Return a boolean same-sized object indicating if the values are null |
| DataFrame.notnull()                             | Return a boolean same-sized object indicating if the values are      |

**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, 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`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.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

**27.4.4 Indexing, iteration**`DataFrame.head([n])`

Returns first n rows

`DataFrame.at``DataFrame.iat`

Continued on next page

Table 27.42 – continued from previous page

|                                                          |                                                                   |
|----------------------------------------------------------|-------------------------------------------------------------------|
| <code>DataFrame.ix</code>                                |                                                                   |
| <code>DataFrame.loc</code>                               |                                                                   |
| <code>DataFrame.iloc</code>                              |                                                                   |
| <code>DataFrame.insert(loc, column, value[, ...])</code> | Insert column into DataFrame at specified location.               |
| <code>DataFrame.__iter__()</code>                        | Iterate over infor axis                                           |
| <code>DataFrame.iteritems()</code>                       | Iterator over (column, series) pairs                              |
| <code>DataFrame.iterrows()</code>                        | Iterate over rows of DataFrame as (index, Series) pairs.          |
| <code>DataFrame.itertuples([index])</code>               | Iterate over rows of DataFrame as tuples, with index value        |
| <code>DataFrame.lookup(row_labels, col_labels)</code>    | Label-based “fancy indexing” function for DataFrame.              |
| <code>DataFrame.pop(item)</code>                         | Return item and drop from frame.                                  |
| <code>DataFrame.tail([n])</code>                         | Returns last n rows                                               |
| <code>DataFrame.xs(key[, axis, level, copy, ...])</code> | Returns a cross-section (row(s) or column(s)) from the DataFrame. |
| <code>DataFrame.isin(values)</code>                      | Return boolean DataFrame showing whether each element in the      |
| <code>DataFrame.query(expr, **kwargs)</code>             | Query the columns of a frame with a boolean expression.           |

### **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  $0 \leq \text{loc} \leq \text{len}(\text{columns})$

**column** : object

**value** : int, Series, or array-like

### **pandas.DataFrame.\_\_iter\_\_**

`DataFrame.__iter__()`

Iterate over infor 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,

```
>>> df = DataFrame([[1, 1.0]], columns=['x', 'y'])
>>> row = next(df.iterrows())[1]
>>> print(row['x'].dtype)
float64
>>> print(df['x'].dtype)
int64
```

### **pandas.DataFrame.itertuples**

`DataFrame.itertuples(index=True)`

Iterate over rows of DataFrame as tuples, with index value as first element of the tuple

### **pandas.DataFrame.lookup**

`DataFrame.lookup(row_labels, col_labels)`

Label-based “fancy indexing” function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

**Parameters** **row\_labels** : sequence

The row labels to use for lookup

**col\_labels** : sequence

The column labels to use for lookup

### Notes

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
 result.append(df.get_value(row, col))
```

### Examples

**values** [ndarray] The found values

### pandas.DataFrame.pop

`DataFrame.pop(item)`  
Return item and drop from frame. Raise `KeyError` if not found.

### pandas.DataFrame.tail

`DataFrame.tail(n=5)`  
Returns last n rows

### pandas.DataFrame.xs

`DataFrame.xs(key, axis=0, level=None, copy=True, drop_level=True)`  
Returns a cross-section (row(s) or column(s)) from the `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 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
>>> 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
 three 2 5 3 5 3
>>> df.xs(('baz', 'three'))
 A B C D
third
2 5 3 5 3
>>> df.xs('one', level=1)
 A B C D
first third
bar 1 4 1 8 9
baz 1 6 6 8 0
>>> df.xs(('baz', 2), level=[0, 'third'])
 A B C D
second
three 5 3 5 3

```

## pandas.DataFrame.isin

DataFrame.isin(values)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

**Parameters** values : iterable, Series, DataFrame or dictionary

The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

**Returns** DataFrame of booleans

### Examples

When values is a list:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
 A B
0 True True
1 False False
2 True False
```

When values is a dict:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
 A B
0 True False # Note that B didn't match the 1 here.
1 False True
2 True True
```

When values is a Series or DataFrame:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
 A B
0 True False
1 False False # Column A in 'other' has a 3, but not at index 1.
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__()`.

**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

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b] # same result as the previous expression
```

For more information on `.at`, `.iat`, `.ix`, `.loc`, and `.iloc`, see the [indexing documentation](#).

## 27.4.5 Binary operator functions

|                                                                |                                                                                                    |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>DataFrame.add(other[, axis, level, fill_value])</code>   | Binary operator add with support to substitute a <code>fill_value</code> for missing data in       |
| <code>DataFrame.sub(other[, axis, level, fill_value])</code>   | Binary operator sub with support to substitute a <code>fill_value</code> for missing data in       |
| <code>DataFrame.mul(other[, axis, level, fill_value])</code>   | Binary operator mul with support to substitute a <code>fill_value</code> for missing data in       |
| <code>DataFrame.div(other[, axis, level, fill_value])</code>   | Binary operator truediv with support to substitute a <code>fill_value</code> for missing data in   |
| <code>DataFrame.truediv(other[, axis, level, ...])</code>      | Binary operator truediv with support to substitute a <code>fill_value</code> for missing data in   |
| <code>DataFrame.floordiv(other[, axis, level, ...])</code>     | Binary operator floordiv with support to substitute a <code>fill_value</code> for missing data in  |
| <code>DataFrame.mod(other[, axis, level, fill_value])</code>   | Binary operator mod with support to substitute a <code>fill_value</code> for missing data in       |
| <code>DataFrame.pow(other[, axis, level, fill_value])</code>   | Binary operator pow with support to substitute a <code>fill_value</code> for missing data in       |
| <code>DataFrame.radd(other[, axis, level, fill_value])</code>  | Binary operator radd with support to substitute a <code>fill_value</code> for missing data in      |
| <code>DataFrame.rsub(other[, axis, level, fill_value])</code>  | Binary operator rsub with support to substitute a <code>fill_value</code> for missing data in      |
| <code>DataFrame.rmul(other[, axis, level, fill_value])</code>  | Binary operator rmul with support to substitute a <code>fill_value</code> for missing data in      |
| <code>DataFrame.rdiv(other[, axis, level, fill_value])</code>  | Binary operator rtruediv with support to substitute a <code>fill_value</code> for missing data in  |
| <code>DataFrame.rtruediv(other[, axis, level, ...])</code>     | Binary operator rtruediv with support to substitute a <code>fill_value</code> for missing data in  |
| <code>DataFrame.rfloordiv(other[, axis, level, ...])</code>    | Binary operator rfloordiv with support to substitute a <code>fill_value</code> for missing data in |
| <code>DataFrame.rmod(other[, axis, level, fill_value])</code>  | Binary operator rmod with support to substitute a <code>fill_value</code> for missing data in      |
| <code>DataFrame.rpow(other[, axis, level, fill_value])</code>  | Binary operator rpow with support to substitute a <code>fill_value</code> for missing data in      |
| <code>DataFrame.lt(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>lt</code>                                            |
| <code>DataFrame.gt(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>gt</code>                                            |
| <code>DataFrame.le(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>le</code>                                            |
| <code>DataFrame.ge(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>ge</code>                                            |
| <code>DataFrame.ne(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>ne</code>                                            |
| <code>DataFrame.eq(other[, axis, level])</code>                | Wrapper for flexible comparison methods <code>eq</code>                                            |
| <code>DataFrame.combine(other, func[, fill_value, ...])</code> | Add two <code>DataFrame</code> objects and do not propagate NaN values, so if for a                |
| <code>DataFrame.combineAdd(other)</code>                       | Add two <code>DataFrame</code> objects and do not propagate                                        |

Continued on next page

Table 27.43 – continued from previous page

|                                             |                                                                       |
|---------------------------------------------|-----------------------------------------------------------------------|
| <code>DataFrame.combine_first(other)</code> | Combine two DataFrame objects and default to non-null values in frame |
| <code>DataFrame.combineMult(other)</code>   | Multiply two DataFrame objects and do not propagate NaN values, so if |

## 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(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.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.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.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.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.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:

```
>>> a.combine_first(b)
```

### **pandas.DataFrame.combineMult**

DataFrame.**combineMult** (*other*)

Multiply two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

**Parameters** `other` : DataFrame

**Returns** DataFrame

## **27.4.6 Function application, GroupBy**

|                                                        |                                                                       |
|--------------------------------------------------------|-----------------------------------------------------------------------|
| DataFrame. <b>apply</b> (func[, axis, broadcast, ...]) | Applies function along input axis of DataFrame.                       |
| DataFrame. <b>applymap</b> (func)                      | Apply a function to a DataFrame that is intended to operate           |
| DataFrame. <b>groupby</b> ([by, axis, level, ...])     | Group series using mapper (dict or key function, apply given function |

### **pandas.DataFrame.apply**

DataFrame.**apply** (*func, axis=0, broadcast=False, raw=False, reduce=True, args=(), \*\*kws*)

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

**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, default True

Try to apply reduction procedures

**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

### Examples

```
>>> df.apply(numpy.sqrt) # returns DataFrame
>>> df.apply(numpy.sum, axis=0) # equiv to df.sum(0)
>>> df.apply(numpy.sum, axis=1) # equiv to df.sum(1)
```

## pandas.DataFrame.applymap

`DataFrame.applymap` (*func*)

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing `map(func, series)` for each series in the DataFrame

**Parameters** **func** : function

Python function, returns a single value from a single value

**Returns** **applied** : DataFrame

**See Also:**

`DataFrame.apply` For operations on rows/columns

**pandas.DataFrame.groupby**

`DataFrame.groupby` (*by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns

**Parameters** **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

**axis** : int, default 0

**level** : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

**as\_index** : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. `as_index=False` is effectively “SQL-style” grouped output

**sort** : boolean, default True

Sort group keys. Get better performance by turning this off

**group\_keys** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**Returns** GroupBy object

**Examples**

```
DataFrame result >>> data.groupby(func, axis=0).mean()
```

```
DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()
```

```
DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()
```

**27.4.7 Computations / Descriptive Stats**

|                                                              |                                                                       |
|--------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>DataFrame.abs()</code>                                 | Return an object with absolute value taken.                           |
| <code>DataFrame.any([axis, bool_only, skipna, level])</code> | Return whether any element is True over requested axis.               |
| <code>DataFrame.clip([lower, upper, out])</code>             | Trim values at input threshold(s)                                     |
| <code>DataFrame.clip_lower(threshold)</code>                 | Return copy of the input with values below given value truncated      |
| <code>DataFrame.clip_upper(threshold)</code>                 | Return copy of input with values above given value truncated          |
| <code>DataFrame.corr([method, min_periods])</code>           | Compute pairwise correlation of columns, excluding NA/null values     |
| <code>DataFrame.corrwith(other[, axis, drop])</code>         | Compute pairwise correlation between rows or columns of two DataFrame |
| <code>DataFrame.count([axis, level, numeric_only])</code>    | Return Series with number of non-NA/null observations over requested  |
| <code>DataFrame.cov([min_periods])</code>                    | Compute pairwise covariance of columns, excluding NA/null values      |

Continued on next page



Table 27.45 – continued from previous page

|                                                                |                                                                         |
|----------------------------------------------------------------|-------------------------------------------------------------------------|
| <code>DataFrame.cummax([axis, dtype, out, skipna])</code>      | Return cumulative max over requested axis.                              |
| <code>DataFrame.cummin([axis, dtype, out, skipna])</code>      | Return cumulative min over requested axis.                              |
| <code>DataFrame.cumprod([axis, dtype, out, skipna])</code>     | Return cumulative prod over requested axis.                             |
| <code>DataFrame.cumsum([axis, dtype, out, skipna])</code>      | Return cumulative sum over requested axis.                              |
| <code>DataFrame.describe([percentile_width])</code>            | Generate various summary statistics of each column, excluding           |
| <code>DataFrame.diff([periods])</code>                         | 1st discrete difference of object                                       |
| <code>DataFrame.eval(expr, **kwargs)</code>                    | Evaluate an expression in the context of the calling DataFrame          |
| <code>DataFrame.kurt([axis, skipna, level, ...])</code>        | Return unbiased kurtosis over requested axis                            |
| <code>DataFrame.mad([axis, skipna, level])</code>              | Return the mean absolute deviation of the values for the requested axis |
| <code>DataFrame.max([axis, skipna, level, ...])</code>         | This method returns the maximum of the values in the object.            |
| <code>DataFrame.mean([axis, skipna, level, ...])</code>        | Return the mean of the values for the requested axis                    |
| <code>DataFrame.median([axis, skipna, level, ...])</code>      | Return the median of the values for the requested axis                  |
| <code>DataFrame.min([axis, skipna, level, ...])</code>         | This method returns the minimum of the values in the object.            |
| <code>DataFrame.mode([axis, numeric_only])</code>              | Gets the mode of each element along the axis selected.                  |
| <code>DataFrame.pct_change([periods, fill_method, ...])</code> | Percent change over given number of periods                             |
| <code>DataFrame.prod([axis, skipna, level, ...])</code>        | Return the product of the values for the requested axis                 |
| <code>DataFrame.quantile([q, axis, numeric_only])</code>       | Return values at the given quantile over requested axis, a la           |
| <code>DataFrame.rank([axis, numeric_only, method, ...])</code> | Compute numerical data ranks (1 through n) along axis.                  |
| <code>DataFrame.skew([axis, skipna, level, ...])</code>        | Return unbiased skew over requested axis                                |
| <code>DataFrame.sum([axis, skipna, level, ...])</code>         | Return the sum of the values for the requested axis                     |
| <code>DataFrame.std([axis, skipna, level, ddof])</code>        | Return unbiased standard deviation over requested axis                  |
| <code>DataFrame.var([axis, skipna, level, ddof])</code>        | Return unbiased variance over requested axis                            |

**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.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)}

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**Returns** **sum** : Series

### pandas.DataFrame.describe

DataFrame.**describe** (*percentile\_width=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

```
>>> 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)

## 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, \*\*kws*)

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)



$0 \leq q \leq 1$

**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, 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.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** **vastance** : Series or DataFrame (if level specified)

## 27.4.8 Reindexing / Selection / Label manipulation

|                                                                |                                                                       |
|----------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>DataFrame.add_prefix(prefix)</code>                      | Concatenate prefix string with panel items names.                     |
| <code>DataFrame.add_suffix(suffix)</code>                      | Concatenate suffix string with panel items names                      |
| <code>DataFrame.align(other[, join, axis, level, ...])</code>  | Align two object on their axes with the                               |
| <code>DataFrame.drop(labels[, axis, level, inplace])</code>    | Return new object with labels in requested axis removed               |
| <code>DataFrame.drop_duplicates([cols, take_last, ...])</code> | Return DataFrame with duplicate rows removed, optionally only         |
| <code>DataFrame.duplicated([cols, take_last])</code>           | Return boolean Series denoting duplicate rows, optionally only        |
| <code>DataFrame.filter([items, like, regex, axis])</code>      | Restrict the info axis to set of items or wildcard                    |
| <code>DataFrame.first(offset)</code>                           | Convenience method for subsetting initial periods of time series data |
| <code>DataFrame.head([n])</code>                               | Returns first n rows                                                  |
| <code>DataFrame.idxmax([axis, skipna])</code>                  | Return index of first occurrence of maximum over requested axis.      |
| <code>DataFrame.idxmin([axis, skipna])</code>                  | Return index of first occurrence of minimum over requested axis.      |
| <code>DataFrame.last(offset)</code>                            | Convenience method for subsetting final periods of time series data   |
| <code>DataFrame.reindex([index, columns])</code>               | Conform DataFrame to new index with optional filling logic, placing   |
| <code>DataFrame.reindex_axis(labels[, axis, ...])</code>       | Conform input object to new index with optional filling logic,        |
| <code>DataFrame.reindex_like(other[, method, ...])</code>      | return an object with matching indicies to myself                     |
| <code>DataFrame.rename([index, columns])</code>                | Alter axes input function or functions.                               |
| <code>DataFrame.reset_index([level, drop, ...])</code>         | For DataFrame with multi-level index, return new DataFrame with       |
| <code>DataFrame.select(crit[, axis])</code>                    | Return data corresponding to axis labels matching criteria            |
| <code>DataFrame.set_index(keys[, drop, append, ...])</code>    | Set the DataFrame index (row labels) using one or more existing       |
| <code>DataFrame.tail([n])</code>                               | Returns last n rows                                                   |
| <code>DataFrame.take(indices[, axis, convert, is_copy])</code> | Analogous to ndarray.take                                             |
| <code>DataFrame.truncate([before, after, axis, copy])</code>   | Truncates a sorted NDFrame before and/or after some particular        |

### 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

**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

```
>>> 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

```
>>> 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.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**

```
>>> 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

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

**27.4.9 Missing data handling**

|                                                           |                                                                       |
|-----------------------------------------------------------|-----------------------------------------------------------------------|
| <code>DataFrame.dropna([axis, how, thresh, ...])</code>   | Return object with labels on given axis omitted where alternately any |
| <code>DataFrame.fillna([value, method, axis, ...])</code> | Fill NA/NaN values using the specified method                         |
| <code>DataFrame.replace([to_replace, value, ...])</code>  | Replace values given in 'to_replace' with 'value'.                    |

**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: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if *regex=True* then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:

- Nested dictionaries, e.g., `{‘a’: {‘b’: nan}}`, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- None:
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also None then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill

**regex** : bool or same types as *to\_replace*, default False

Whether to interpret *to\_replace* and/or *value* as regular expressions. If this is True then *to\_replace* must be a string. Otherwise, *to\_replace* must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method** : string, optional, {‘pad’, ‘ffill’, ‘bfill’}

The method to use when for replacement, when *to\_replace* is a list.

**Returns** **filled** : NDFrame

**Raises** **AssertionError**

- If *regex* is not a bool and *to\_replace* is not None.

**TypeError**

- If *to\_replace* is a dict and *value* is not a list, dict, ndarray, or Series
- If *to\_replace* is None and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.

**ValueError**

- If *to\_replace* and *value* are lists or ndarrays, but they are not the same length.

**See Also:**

`NDFrame.reindex`, `NDFrame.asfreq`, `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.

## 27.4.10 Reshaping, sorting, transposing

|                                                               |                                                                         |
|---------------------------------------------------------------|-------------------------------------------------------------------------|
| <code>DataFrame.delevel(*args, **kwargs)</code>               |                                                                         |
| <code>DataFrame.pivot([index, columns, values])</code>        | Reshape data (produce a “pivot” table) based on column values.          |
| <code>DataFrame.reorder_levels(order[, axis])</code>          | Rearrange index levels using input order.                               |
| <code>DataFrame.sort([columns, column, axis, ...])</code>     | Sort DataFrame either by labels (along either axis) or by the values in |
| <code>DataFrame.sort_index([axis, by, ascending, ...])</code> | Sort DataFrame either by labels (along either axis) or by the values in |
| <code>DataFrame.sortlevel([level, axis, ...])</code>          | Sort multilevel index by chosen axis and primary level.                 |
| <code>DataFrame.swaplevel(i, j[, axis])</code>                | Swap levels i and j in a MultiIndex on a particular axis                |
| <code>DataFrame.stack([level, dropna])</code>                 | Pivot a level of the (possibly hierarchical) column labels, returning a |
| <code>DataFrame.unstack([level])</code>                       | Pivot a level of the (necessarily hierarchical) index labels, returning |
| <code>DataFrame.T</code>                                      | Transpose index and columns                                             |
| <code>DataFrame.to_panel()</code>                             | Transform long (stacked) format (DataFrame) into wide (3D, Panel)       |
| <code>DataFrame.transpose()</code>                            | Transpose index and columns                                             |

### 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

```
>>> 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

```
>>> 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.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

**Returns** `stacked` : DataFrame or Series**Examples**

```
>>> s
 a b
one 1. 2.
two 3. 4.

>>> s.stack()
one a 1
 b 2
two a 3
 b 4
```

**pandas.DataFrame.unstack**`DataFrame.unstack (level=-1)`

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex)

**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

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
... ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one a 1
 b 2
two a 3
 b 4
dtype: float64

>>> s.unstack(level=-1)
 a b
one 1 2
two 3 4

>>> s.unstack(level=0)
 one two
a 1 3
b 2 4

>>> df = s.unstack(level=0)
>>> df.unstack()
one 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

## 27.4.11 Combining / joining / merging

|                                                             |                                                                               |
|-------------------------------------------------------------|-------------------------------------------------------------------------------|
| <code>DataFrame.append(other[, ignore_index, ...])</code>   | Append columns of other to end of this frame's columns and index, returning a |
| <code>DataFrame.join(other[, on, how, lsuffix, ...])</code> | Join columns with other DataFrame either on index or on a key                 |
| Continued on                                                |                                                                               |

Table 27.49 – continued from previous page

|                                                              |                                                                          |
|--------------------------------------------------------------|--------------------------------------------------------------------------|
| <code>DataFrame.merge(right[, how, on, left_on, ...])</code> | Merge DataFrame objects by performing a database-style join operation by |
| <code>DataFrame.update(other[, join, overwrite, ...])</code> | Modify DataFrame in place using non-NA values from passed                |

**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.join**

`DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)`

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

**Parameters** **other** : DataFrame, Series with name field set, or list of DataFrame

Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

**on** : column name, tuple/list of column names, or array-like

Column(s) to use for joining, otherwise join on index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

**how** : { 'left', 'right', 'outer', 'inner' }

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise

- left: use calling frame's index
- right: use input frame's index
- outer: form union of indexes
- inner: use intersection of indexes

**lsuffix** : string

Suffix to use from left frame's overlapping columns

**rsuffix** : string

Suffix to use from right frame's overlapping columns

**sort** : boolean, default False

Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

**Returns** **joined** : DataFrame

#### Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

### pandas.DataFrame.merge

DataFrame.**merge**(right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=False, suffixes=('\_x', '\_y'), copy=True)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

**Parameters** **right** : DataFrame

**how** : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

**on** : label or list

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

**left\_on** : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

**right\_on** : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left\_on docs

**left\_index** : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

**right\_index** : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left\_index

**sort** : boolean, default False

Sort the join keys lexicographically in the result DataFrame

**suffixes** : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

**copy** : boolean, default True

If False, do not copy data unnecessarily

**Returns** **merged** : DataFrame

### Examples

```
>>> A >>> B
 lkey value rkey value
0 foo 1 0 foo 5
1 bar 2 1 bar 6
2 baz 3 2 qux 7
3 foo 4 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.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.

### 27.4.12 Time series-related

|                                                  |                                                                       |
|--------------------------------------------------|-----------------------------------------------------------------------|
| DataFrame.asfreq(freq[, method, how, normalize]) | Convert all TimeSeries inside to specified frequency using DateOffset |
| 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                              |

Continu

Table 27.50 – continued from previous page

|                                                              |                                                                                               |
|--------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <code>DataFrame.last_valid_index()</code>                    | Return label for last non-NA/null value                                                       |
| <code>DataFrame.resample(rule[, how, axis, ...])</code>      | Convenience method for frequency conversion and resampling of regular time series             |
| <code>DataFrame.to_period([freq, axis, copy])</code>         | Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency                    |
| <code>DataFrame.to_timestamp([freq, how, axis, copy])</code> | Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period                            |
| <code>DataFrame.tz_convert(tz[, axis, copy])</code>          | Convert TimeSeries to target time zone. If it is time zone naive, it will be converted to UTC |
| <code>DataFrame.tz_localize(tz[, axis, copy, ...])</code>    | Localize tz-naive TimeSeries to target time zone                                              |

## 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.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 is 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, loffset=None, limit=None, base=0)`

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** **rule** : 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, fill\_method for upsampling, default None**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

**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

## 27.4.13 Plotting

---

|                                                                      |                                                                                   |
|----------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>DataFrame.boxplot</code> ( <i>[column, by, ax, ...]</i> )      | Make a box plot from DataFrame column/columns optionally grouped                  |
| <code>DataFrame.hist</code> ( <i>data[, column, by, grid, ...]</i> ) | Draw Histogram the DataFrame's series using matplotlib / pylab.                   |
| <code>DataFrame.plot</code> ( <i>[frame, x, y, subplots, ...]</i> )  | Make line, bar, or scatter plots of DataFrame series with the index on the x-axis |

---

### pandas.DataFrame.boxplot

`DataFrame.boxplot` (*column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, \*\*kws*)

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

**Returns** **ax** : matplotlib.axes.AxesSubplot

## 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*, *\*\*kws*)

Draw Histogram 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 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

## 27.4.14 Serialization / IO / Conversion

|                                                                |                                                                    |
|----------------------------------------------------------------|--------------------------------------------------------------------|
| <code>DataFrame.from_csv(path[, header, sep, ...])</code>      | Read delimited file into DataFrame                                 |
| <code>DataFrame.from_dict(data[, orient, dtype])</code>        | Construct DataFrame from dict of array-like or dicts               |
| <code>DataFrame.from_items(items[, columns, orient])</code>    | Convert (key, value) pairs to DataFrame. The keys will be the axis |
| <code>DataFrame.from_records(data[, index, ...])</code>        | Convert structured or record ndarray to DataFrame                  |
| <code>DataFrame.info([verbose, buf, max_cols])</code>          | Concise summary of a DataFrame.                                    |
| <code>DataFrame.to_pickle(path)</code>                         | Pickle (serialize) object to input file path                       |
| <code>DataFrame.to_csv(path_or_buf[, sep, na_rep, ...])</code> | Write DataFrame to a comma-separated values (csv) file             |
| <code>DataFrame.to_hdf(path_or_buf, key, **kwargs)</code>      | activate the HDFStore                                              |
| <code>DataFrame.to_dict([outtype])</code>                      | Convert DataFrame to dictionary.                                   |
| <code>DataFrame.to_excel(excel_writer[, ...])</code>           | Write DataFrame to a excel sheet                                   |
| <code>DataFrame.to_json([path_or_buf, orient, ...])</code>     | Convert the object to a JSON string.                               |

Continued on next page

Table 27.52 – continued from previous page

|                                                                |                                                                        |
|----------------------------------------------------------------|------------------------------------------------------------------------|
| <code>DataFrame.to_html([buf, columns, col_space, ...])</code> | Render a DataFrame as an HTML table.                                   |
| <code>DataFrame.to_latex([buf, columns, ...])</code>           | Render a DataFrame to a tabular environment table.                     |
| <code>DataFrame.to_stata(fname[, convert_dates, ...])</code>   | A class for writing Stata binary dta files from array-like objects     |
| <code>DataFrame.to_records([index, convert_datetime64])</code> | Convert DataFrame to record array. Index will be put in the            |
| <code>DataFrame.to_sparse([fill_value, kind])</code>           | Convert to SparseDataFrame                                             |
| <code>DataFrame.to_string([buf, columns, ...])</code>          | Render a DataFrame to a console-friendly tabular output.               |
| <code>DataFrame.to_clipboard([excel, sep])</code>              | Attempt to write text representation of object to the system clipboard |

## pandas.DataFrame.from\_csv

**classmethod** `DataFrame.from_csv` (*path, header=0, sep=',', index\_col=0, parse\_dates=True, encoding=None, tupleize\_cols=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)

**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 named 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**

`DataFrame.to_csv(path_or_buf, sep=',', na_rep='', float_format=None, cols=None, header=True, index=True, index_label=None, mode='w', nanRep=None, encoding=None, quoting=None, line_terminator='n', chunksize=None, tupleize_cols=False, date_format=None, **kwargs)`

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, an 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:

```
>>> 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** : type of date conversion, epoch or iso

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, sparsify=None, 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\_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\_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

```
>>> 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

## 27.5 Panel

### 27.5.1 Constructor

---

`Panel([data, items, major_axis, minor_axis, ...])` Represents wide format panel data, stored as 3-dimensional array

---

#### pandas.Panel

**class** `pandas.Panel` (*data=None, items=None, major\_axis=None, minor\_axis=None, copy=False, dtype=None*)

Represents wide format panel data, stored as 3-dimensional array

**Parameters** **data** : ndarray (items x major x minor), or dict of DataFrames

**items** : Index or array-like

axis=0

**major\_axis** : Index or array-like

axis=1

**minor\_axis** : Index or array-like

axis=2

**dtype** : dtype, default None

Data type to force, otherwise infer

**copy** : boolean, default False

Copy data from inputs. Only affects DataFrame / 2d ndarray input

#### Attributes

|                     |                                                                  |
|---------------------|------------------------------------------------------------------|
| <code>at</code>     |                                                                  |
| <code>axes</code>   | index(es) of the NDFrame                                         |
| <code>blocks</code> | Internal property, property synonym for <code>as_blocks()</code> |
| <code>empty</code>  | True if NDFrame is entirely empty [no items]                     |
| <code>iat</code>    |                                                                  |
| <code>iloc</code>   |                                                                  |
| <code>ix</code>     |                                                                  |
| <code>loc</code>    |                                                                  |
| <code>ndim</code>   | Number of axes / array dimensions                                |
| <code>shape</code>  | tuple of axis dimensions                                         |
| <code>values</code> | Numpy representation of NDFrame                                  |

#### pandas.Panel.at

`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.empty**

`Panel.empty`  
True if NDFrame is entirely empty [no items]

### **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

|         |  |
|---------|--|
| is_copy |  |
|---------|--|

**Methods**

|                                                           |                                                                         |
|-----------------------------------------------------------|-------------------------------------------------------------------------|
| <code>abs()</code>                                        | Return an object with absolute value taken.                             |
| <code>add(other[, axis])</code>                           | Wrapper method for add                                                  |
| <code>add_prefix(prefix)</code>                           | Concatenate prefix string with panel items names.                       |
| <code>add_suffix(suffix)</code>                           | Concatenate suffix string with panel items names                        |
| <code>align(other[, join, axis, level, copy, ...])</code> | Align two object on their axes with the                                 |
| <code>apply(func[, axis])</code>                          | Apply                                                                   |
| <code>as_blocks([columns])</code>                         | Convert the frame to a dict of dtype -> Constructor Types that each has |
| <code>as_matrix()</code>                                  |                                                                         |
| <code>asfreq(freq[, method, how, normalize])</code>       | Convert all TimeSeries inside to specified frequency using DateOffset   |
| <code>astype(dtype[, copy, raise_on_error])</code>        | Cast object to input numpy.dtype                                        |
| <code>at_time(time[, asof])</code>                        | Select values at particular time of day (e.g.                           |
| <code>between_time(start_time, end_time[, ...])</code>    | Select values between particular times of the day (e.g., 9:00-9:30 AM)  |
| <code>bfill([axis, inplace, limit, downcast])</code>      | Synonym for NDFrame.fillna(method='bfill')                              |
| <code>bool()</code>                                       | Return the bool of a single element PandasObject                        |
| <code>clip([lower, upper, out])</code>                    | Trim values at input threshold(s)                                       |
| <code>clip_lower(threshold)</code>                        | Return copy of the input with values below given value truncated        |
| <code>clip_upper(threshold)</code>                        | Return copy of input with values above given value truncated            |
| <code>compound([axis, skipna, level])</code>              | Return the compound percentage of the values for the requested axis     |
| <code>conform(frame[, axis])</code>                       | Conform input DataFrame to align with chosen axis pair.                 |
| <code>consolidate([inplace])</code>                       | Compute NDFrame with "consolidated" internals (data of each dtype       |
| <code>convert_objects([convert_dates, ...])</code>        | Attempt to infer better dtype for object columns                        |
| <code>copy([deep])</code>                                 | Make a copy of this object                                              |
| <code>count([axis])</code>                                | Return number of observations over requested axis.                      |
| <code>cummax([axis, dtype, out, skipna])</code>           | Return cumulative max over requested axis.                              |
| <code>cummin([axis, dtype, out, skipna])</code>           | Return cumulative min over requested axis.                              |
| <code>cumprod([axis, dtype, out, skipna])</code>          | Return cumulative prod over requested axis.                             |
| <code>cumsum([axis, dtype, out, skipna])</code>           | Return cumulative sum over requested axis.                              |
| <code>div(other[, axis])</code>                           | Wrapper method for truediv                                              |
| <code>divide(other[, axis])</code>                        | Wrapper method for truediv                                              |
| <code>drop(labels[, axis, level, inplace])</code>         | Return new object with labels in requested axis removed                 |
| <code>dropna([axis, how, inplace])</code>                 | Drop 2D from panel, holding passed axis constant                        |
| <code>eq(other)</code>                                    | Wrapper for comparison method eq                                        |
| <code>ffill([axis, inplace, limit, downcast])</code>      | Synonym for NDFrame.fillna(method='ffill')                              |
| <code>fillna([value, method, axis, inplace, ...])</code>  | Fill NA/NaN values using the specified method                           |
| <code>filter([items, like, regex, axis])</code>           | Restrict the info axis to set of items or wildcard                      |
| <code>first(offset)</code>                                | Convenience method for subsetting initial periods of time series data   |
| <code>floordiv(other[, axis])</code>                      | Wrapper method for floordiv                                             |
| <code>fromDict(data[, intersect, orient, dtype])</code>   | Construct Panel from dict of DataFrame objects                          |
| <code>from_dict(data[, intersect, orient, dtype])</code>  | Construct Panel from dict of DataFrame objects                          |
| <code>ge(other)</code>                                    | Wrapper for comparison method ge                                        |
| <code>get(key[, default])</code>                          | Get item from object for given key (DataFrame column, Panel slice,      |

Continued on 1

Table 27.55 – continued from previous page

|                                                                |                                                                               |
|----------------------------------------------------------------|-------------------------------------------------------------------------------|
| <code>get_dtype_counts()</code>                                | return the counts of dtypes in this frame                                     |
| <code>get_ftype_counts()</code>                                | return the counts of ftypes in this frame                                     |
| <code>get_value(*args)</code>                                  | Quickly retrieve single value at (item, major, minor) location                |
| <code>get_values()</code>                                      | same as values (but handles sparseness conversions)                           |
| <code>groupby(function[, axis])</code>                         | Group data on given axis, returning GroupBy object                            |
| <code>gt(other)</code>                                         | Wrapper for comparison method gt                                              |
| <code>head([n])</code>                                         |                                                                               |
| <code>interpolate([method, axis, limit, inplace, ...])</code>  | Interpolate values according to different methods.                            |
| <code>isnull()</code>                                          | Return a boolean same-sized object indicating if the values are null          |
| <code>iteritems()</code>                                       | Iterate over (label, values) on info axis                                     |
| <code>iterkv(*args, **kwargs)</code>                           | iteritems alias used to get around 2to3. Deprecated                           |
| <code>join(other[, how, lsuffix, rsuffix])</code>              | Join items with other Panel either on major and minor axes column             |
| <code>keys()</code>                                            | Get the ‘info axis’ (see Indexing for more)                                   |
| <code>kurt([axis, skipna, level, numeric_only])</code>         | Return unbiased kurtosis over requested axis                                  |
| <code>kurtosis([axis, skipna, level, numeric_only])</code>     | Return unbiased kurtosis over requested axis                                  |
| <code>last(offset)</code>                                      | Convenience method for subsetting final periods of time series data           |
| <code>le(other)</code>                                         | Wrapper for comparison method le                                              |
| <code>load(path)</code>                                        | Deprecated.                                                                   |
| <code>lt(other)</code>                                         | Wrapper for comparison method lt                                              |
| <code>mad([axis, skipna, level])</code>                        | Return the mean absolute deviation of the values for the requested axis       |
| <code>major_xs(key[, copy])</code>                             | Return slice of panel along major axis                                        |
| <code>mask(cond)</code>                                        | Returns copy whose values are replaced with nan if the                        |
| <code>max([axis, skipna, level, numeric_only])</code>          | This method returns the maximum of the values in the object.                  |
| <code>mean([axis, skipna, level, numeric_only])</code>         | Return the mean of the values for the requested axis                          |
| <code>median([axis, skipna, level, numeric_only])</code>       | Return the median of the values for the requested axis                        |
| <code>min([axis, skipna, level, numeric_only])</code>          | This method returns the minimum of the values in the object.                  |
| <code>minor_xs(key[, copy])</code>                             | Return slice of panel along minor axis                                        |
| <code>mod(other[, axis])</code>                                | Wrapper method for mod                                                        |
| <code>mul(other[, axis])</code>                                | Wrapper method for mul                                                        |
| <code>multiply(other[, axis])</code>                           | Wrapper method for mul                                                        |
| <code>ne(other)</code>                                         | Wrapper for comparison method ne                                              |
| <code>notnull()</code>                                         | Return a boolean same-sized object indicating if the values are               |
| <code>pct_change([periods, fill_method, limit, freq])</code>   | Percent change over given number of periods                                   |
| <code>pop(item)</code>                                         | Return item and drop from frame.                                              |
| <code>pow(other[, axis])</code>                                | Wrapper method for pow                                                        |
| <code>prod([axis, skipna, level, numeric_only])</code>         | Return the product of the values for the requested axis                       |
| <code>product([axis, skipna, level, numeric_only])</code>      | Return the product of the values for the requested axis                       |
| <code>radd(other[, axis])</code>                               | Wrapper method for radd                                                       |
| <code>rdiv(other[, axis])</code>                               | Wrapper method for rtruediv                                                   |
| <code>reindex([items, major_axis, minor_axis])</code>          | Conform Panel to new index with optional filling logic, placing               |
| <code>reindex_axis(labels[, axis, method, level, ...])</code>  | Conform input object to new index with optional filling logic,                |
| <code>reindex_like(other[, method, copy, limit])</code>        | return an object with matching indicies to myself                             |
| <code>rename([items, major_axis, minor_axis])</code>           | Alter axes input function or functions.                                       |
| <code>rename_axis(mapper[, axis, copy, inplace])</code>        | Alter index and / or columns using input function or functions.               |
| <code>replace([to_replace, value, inplace, limit, ...])</code> | Replace values given in ‘to_replace’ with ‘value’.                            |
| <code>resample(rule[, how, axis, fill_method, ...])</code>     | Convenience method for frequency conversion and resampling of regular time-se |
| <code>rfloordiv(other[, axis])</code>                          | Wrapper method for rfloordiv                                                  |
| <code>rmod(other[, axis])</code>                               | Wrapper method for rmod                                                       |
| <code>rmul(other[, axis])</code>                               | Wrapper method for rmul                                                       |
| <code>rpow(other[, axis])</code>                               | Wrapper method for rpow                                                       |

Continued on next page

Table 27.55 – continued from previous page

|                                                               |                                                                        |
|---------------------------------------------------------------|------------------------------------------------------------------------|
| <code>rsub(other[, axis])</code>                              | Wrapper method for <code>rsub</code>                                   |
| <code>rtruediv(other[, axis])</code>                          | Wrapper method for <code>rtruediv</code>                               |
| <code>save(path)</code>                                       | Deprecated.                                                            |
| <code>select(crit[, axis])</code>                             | Return data corresponding to axis labels matching criteria             |
| <code>set_value(*args)</code>                                 | Quickly set single value at (item, major, minor) location              |
| <code>shift(lags[, freq, axis])</code>                        | Shift major or minor axis by specified number of leads/lags.           |
| <code>skew([axis, skipna, level, numeric_only])</code>        | Return unbiased skew over requested axis                               |
| <code>sort_index([axis, ascending])</code>                    | Sort object by labels (along an axis)                                  |
| <code>squeeze()</code>                                        | squeeze length 1 dimensions                                            |
| <code>std([axis, skipna, level, ddof])</code>                 | Return unbiased standard deviation over requested axis                 |
| <code>sub(other[, axis])</code>                               | Wrapper method for <code>sub</code>                                    |
| <code>subtract(other[, axis])</code>                          | Wrapper method for <code>sub</code>                                    |
| <code>sum([axis, skipna, level, numeric_only])</code>         | Return the sum of the values for the requested axis                    |
| <code>swapaxes(axis1, axis2[, copy])</code>                   | Interchange axes and swap values axes appropriately                    |
| <code>swaplevel(i, j[, axis])</code>                          | Swap levels i and j in a MultiIndex on a particular axis               |
| <code>tail([n])</code>                                        |                                                                        |
| <code>take(indices[, axis, convert, is_copy])</code>          | Analogous to <code>ndarray.take</code>                                 |
| <code>toLong(*args, **kwargs)</code>                          |                                                                        |
| <code>to_clipboard([excel, sep])</code>                       | Attempt to write text representation of object to the system clipboard |
| <code>to_dense()</code>                                       | Return dense representation of NDFrame (as opposed to sparse)          |
| <code>to_excel(path[, na_rep, engine])</code>                 | Write each DataFrame in Panel to a separate excel sheet                |
| <code>to_frame([filter_observations])</code>                  | Transform wide format into long (stacked) format as DataFrame          |
| <code>to_hdf(path_or_buf, key, **kwargs)</code>               | activate the HDFStore                                                  |
| <code>to_json([path_or_buf, orient, date_format, ...])</code> | Convert the object to a JSON string.                                   |
| <code>to_long(*args, **kwargs)</code>                         |                                                                        |
| <code>to_msgpack([path_or_buf])</code>                        | msgpack (serialize) object to input file path                          |
| <code>to_pickle(path)</code>                                  | Pickle (serialize) object to input file path                           |
| <code>to_sparse([fill_value, kind])</code>                    | Convert to SparsePanel                                                 |
| <code>transpose(*args, **kwargs)</code>                       | Permute the dimensions of the Panel                                    |
| <code>truediv(other[, axis])</code>                           | Wrapper method for <code>truediv</code>                                |
| <code>truncate([before, after, axis, copy])</code>            | Truncates a sorted NDFrame before and/or after some particular         |
| <code>tshift([periods, freq, axis])</code>                    |                                                                        |
| <code>tz_convert(tz[, axis, copy])</code>                     | Convert TimeSeries to target time zone. If it is time zone naive, it   |
| <code>tz_localize(tz[, axis, copy, infer_dst])</code>         | Localize tz-naive TimeSeries to target time zone                       |
| <code>update(other[, join, overwrite, ...])</code>            | Modify Panel in place using non-NA values from passed                  |
| <code>var([axis, skipna, level, ddof])</code>                 | Return unbiased variance over requested axis                           |
| <code>where(cond[, other, inplace, axis, level, ...])</code>  | Return an object of same shape as self and whose corresponding         |
| <code>xs(key[, axis, copy])</code>                            | 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')`

Apply

**Parameters** **func** : numpy function

Signature should match `numpy.{sum, mean, var, std}` etc.

**axis** : {'major', 'minor', 'items'}

**fill\_value** : boolean, default True

Replace NaN values with specified first

**Returns** **result** : DataFrame or Panel

### 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 methdo

**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, *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

**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.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.ffill

Panel.**ffill** (*axis=0, inplace=False, limit=None, downcast=None*)

Synonym for NDFrame.fillna(method='ffill')

#### 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`

### 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 frame

### **pandas.Panel.get\_ftype\_counts**

**Panel.get\_ftype\_counts()**  
return the counts of ftypes in this frame

### **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
- **‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘polynomial’** is passed to *scipy.interpolate.interpld* with the order given both ‘polynomial’ and ‘spline’ require that you also specify an 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 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**

`ts.last('5M')` -> Last 5 months

#### **pandas.Panel.le**

`Panel.le` (*other*)

Wrapper for comparison method `le`

#### **pandas.Panel.load**

`Panel.load` (*path*)

Deprecated. Use `read_pickle` instead.

#### **pandas.Panel.lt**

`Panel.lt` (*other*)

Wrapper for comparison method `lt`

### pandas.Panel.mad

`Panel.mad` (*axis=None, skipna=None, level=None, \*\*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

`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}

**Axis to broadcast over**

**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}

**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.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

```
>>> 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

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```



**pandas.Panel.reindex\_like****Panel.reindex\_like** (*other, method=None, copy=True, limit=None*)

return an object with matching indicies to myself

**Parameters** **other** : Object**method** : string or None**copy** : boolean, default True**limit** : int, default None

Maximum size gap to forward or backward fill

**Returns** **reindexed** : same as input**Notes**Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`**pandas.Panel.rename****Panel.rename** (*items=None, major\_axis=None, minor\_axis=None, \*\*kwargs*)

Alter axes input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

**Parameters** **items, major\_axis, minor\_axis** : dict-like or function, optional

Transformation to apply to that axis values

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new Panel. If True then value of copy is ignored.

**Returns** **renamed** : Panel (new object)**pandas.Panel.rename\_axis****Panel.rename\_axis** (*mapper, axis=0, copy=True, inplace=False*)

Alter index and / or columns using input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

**Parameters** **mapper** : dict-like or function, optional**axis** : int or string, default 0**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False**Returns** **renamed** : type of caller

**pandas.Panel.replace**

`Panel.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)`

Replace values given in 'to\_replace' with 'value'.

**Parameters** `to_replace` : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - str: string exactly matching *to\_replace* will be replaced with *value*
  - regex: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- None:
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also None then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill

**regex** : bool or same types as *to\_replace*, default False

Whether to interpret *to\_replace* and/or *value* as regular expressions. If this is True then *to\_replace* must be a string. Otherwise, *to\_replace* must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

**method** : string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a list.

**Returns** `filled` : NDFrame

**Raises** `AssertionError`

- If `regex` is not a bool and `to_replace` is not None.

**TypeError**

- If `to_replace` is a dict and `value` is not a list, dict, ndarray, or Series
- If `to_replace` is None and `regex` is not compilable into a regular expression or is a list, dict, ndarray, or Series.

**ValueError**

- If `to_replace` and `value` are lists or ndarrays, but they are not the same length.

**See Also:**

`NDFrame.reindex`, `NDFrame.asfreq`, `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` : 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, fill\_method for upsampling, default None

`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}

**Axis to broadcast over**

**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.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.rsub**

`Panel.rsub (other, axis=0)`

Wrapper method for rsub

**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.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

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** **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
- OS X: none

### pandas.Panel.to\_dense

`Panel.to_dense()`

Return dense representation of NDFrame (as opposed to sparse)

### pandas.Panel.to\_excel

`Panel.to_excel(path, na_rep='', engine=None, **kwargs)`

Write each DataFrame in Panel to a separate excel sheet

**Parameters path** : string or ExcelWriter object

File path or existing ExcelWriter

**na\_rep** : string, default ''

Missing data representation

**engine** : string, default None

write engine to use - you can also set this via the options  
`io.excel.xlsx.writer,` `io.excel.xls.writer,` and  
`io.excel.xlsm.writer.`

**Other Parameters** **float\_format** : string, default None

Format string for floating point numbers

**cols** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** : upper left cell row to dump data frame

**startcol** : upper left cell column to dump data frame

## Notes

Keyword arguments (and *na\_rep*) are passed to the `to_excel` method for each DataFrame written.

## pandas.Panel.to\_frame

`Panel.to_frame (filter_observations=True)`

Transform wide format into long (stacked) format as DataFrame

**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, an 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 compilib 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** : type of date conversion, epoch or iso

epoch = epoch milliseconds, iso = ISO8601, default is epoch

**double\_precision** : The number of decimal places to use when encoding floating point values, default 10.

**force\_ascii** : force encoded string to be ASCII, default True.

**date\_unit** : string, default 'ms' (milliseconds)

The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

**default\_handler** : callable, default None

Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

**Returns** same type as input object with filtered info axis

### **pandas.Panel.to\_long**

`Panel.to_long(*args, **kwargs)`

### **pandas.Panel.to\_msgpack**

`Panel.to_msgpack(path_or_buf=None, **kwargs)`  
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

**Parameters** **path** : string File path, buffer-like, or None  
if None, return generated string

**append** : boolean whether to append to an existing msgpack  
(default is False)

**compress** : type of compressor (zlib or blosc), default to None (no compression)

### **pandas.Panel.to\_pickle**

`Panel.to_pickle(path)`  
Pickle (serialize) object to input file path

**Parameters** **path** : string  
File path

### **pandas.Panel.to\_sparse**

`Panel.to_sparse(fill_value=None, kind='block')`  
Convert to SparsePanel

**Parameters** `fill_value` : float, default NaN

`kind` : { 'block', 'integer' }

**Returns** `y` : SparseDataFrame

### pandas.Panel.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

```
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

### pandas.Panel.truediv

`Panel.truediv` (*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.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

### **pandas.Panel.tshift**

`Panel.tshift` (*periods=1, freq=None, axis='major', \*\*kws*)

### **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

### **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`

## 27.5.2 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

|                           |                                   |
|---------------------------|-----------------------------------|
| <code>Panel.values</code> | Numpy representation of NDFrame   |
| <code>Panel.axes</code>   | index(es) of the NDFrame          |
| <code>Panel.ndim</code>   | Number of axes / array dimensions |
| <code>Panel.shape</code>  | tuple of axis dimensions          |

### `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

## 27.5.3 Conversion

|                                                          |                                                                      |
|----------------------------------------------------------|----------------------------------------------------------------------|
| <code>Panel.astype(dtype[, copy, raise_on_error])</code> | Cast object to input numpy.dtype                                     |
| <code>Panel.copy([deep])</code>                          | Make a copy of this object                                           |
| <code>Panel.isnull()</code>                              | Return a boolean same-sized object indicating if the values are null |
| <code>Panel.notnull()</code>                             | Return a boolean same-sized object indicating if the values are      |

### `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

## 27.5.4 Getting and setting

|                                     |                                                                |
|-------------------------------------|----------------------------------------------------------------|
| <code>Panel.get_value(*args)</code> | Quickly retrieve single value at (item, major, minor) location |
| <code>Panel.set_value(*args)</code> | Quickly set single value at (item, major, minor) location      |

### 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

## 27.5.5 Indexing, iteration, slicing

|                                          |                                           |
|------------------------------------------|-------------------------------------------|
| <code>Panel.at</code>                    |                                           |
| <code>Panel.iat</code>                   |                                           |
| <code>Panel.ix</code>                    |                                           |
| <code>Panel.loc</code>                   |                                           |
| <code>Panel.iloc</code>                  |                                           |
| <code>Panel.__iter__()</code>            | Iterate over infor axis                   |
| <code>Panel.iteritems()</code>           | Iterate over (label, values) on info axis |
| <code>Panel.pop(item)</code>             | Return item and drop from frame.          |
| <code>Panel.xs(key[, axis, copy])</code> | Return slice of panel along selected axis |
| <code>Panel.major_xs(key[, copy])</code> | Return slice of panel along major axis    |
| <code>Panel.minor_xs(key[, copy])</code> | Return slice of panel along minor axis    |

### 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 infor 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](#).

## 27.5.6 Binary operator functions

|                                             |                                  |
|---------------------------------------------|----------------------------------|
| <code>Panel.add(other[, axis])</code>       | Wrapper method for add           |
| <code>Panel.sub(other[, axis])</code>       | Wrapper method for sub           |
| <code>Panel.mul(other[, axis])</code>       | Wrapper method for mul           |
| <code>Panel.div(other[, axis])</code>       | Wrapper method for truediv       |
| <code>Panel.truediv(other[, axis])</code>   | Wrapper method for truediv       |
| <code>Panel.floordiv(other[, axis])</code>  | Wrapper method for floordiv      |
| <code>Panel.mod(other[, axis])</code>       | Wrapper method for mod           |
| <code>Panel.pow(other[, axis])</code>       | Wrapper method for pow           |
| <code>Panel.radd(other[, axis])</code>      | Wrapper method for radd          |
| <code>Panel.rsub(other[, axis])</code>      | Wrapper method for rsub          |
| <code>Panel.rmul(other[, axis])</code>      | Wrapper method for rmul          |
| <code>Panel.rdiv(other[, axis])</code>      | Wrapper method for rtruediv      |
| <code>Panel.rtruediv(other[, axis])</code>  | Wrapper method for rtruediv      |
| <code>Panel.rfloordiv(other[, axis])</code> | Wrapper method for rfloordiv     |
| <code>Panel.rmod(other[, axis])</code>      | Wrapper method for rmod          |
| <code>Panel.rpow(other[, axis])</code>      | Wrapper method for rpow          |
| <code>Panel.lt(other)</code>                | Wrapper for comparison method lt |
| <code>Panel.gt(other)</code>                | Wrapper for comparison method gt |
| <code>Panel.le(other)</code>                | Wrapper for comparison method le |
| <code>Panel.ge(other)</code>                | Wrapper for comparison method ge |
| <code>Panel.ne(other)</code>                | Wrapper for comparison method ne |
| <code>Panel.eq(other)</code>                | Wrapper for comparison method eq |

### 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}

**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.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.truediv

`Panel.truediv (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.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.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.pow**

Panel.**pow**(*other*, *axis=0*)  
Wrapper method for pow

**Parameters** **other** : DataFrame or Panel  
**axis** : {items, major\_axis, minor\_axis}  
**Axis to broadcast over**  
**Returns** Panel

### **pandas.Panel.radd**

Panel.**radd**(*other*, *axis=0*)  
Wrapper method for radd

**Parameters** **other** : DataFrame or Panel  
**axis** : {items, major\_axis, minor\_axis}  
**Axis to broadcast over**  
**Returns** Panel

### **pandas.Panel.rsub**

Panel.**rsub**(*other*, *axis=0*)  
Wrapper method for rsub

**Parameters** **other** : DataFrame or Panel  
**axis** : {items, major\_axis, minor\_axis}  
**Axis to broadcast over**  
**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`

**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

## 27.5.7 Function application, GroupBy

|                                          |                                                    |
|------------------------------------------|----------------------------------------------------|
| Panel. <b>apply</b> (func[, axis])       | Apply                                              |
| Panel. <b>groupby</b> (function[, axis]) | Group data on given axis, returning GroupBy object |

### pandas.Panel.apply

Panel.**apply** (*func*, *axis*=*'major'*)  
 Apply  
**Parameters** **func** : numpy function



Signature should match numpy.{sum, mean, var, std} etc.

**axis** : { 'major', 'minor', 'items' }

**fill\_value** : boolean, default True

Replace NaN values with specified first

**Returns** **result** : DataFrame or Panel

## 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

## 27.5.8 Computations / Descriptive Stats

|                                                                |                                                                  |
|----------------------------------------------------------------|------------------------------------------------------------------|
| <code>Panel.abs()</code>                                       | Return an object with absolute value taken.                      |
| <code>Panel.clip([lower, upper, out])</code>                   | Trim values at input threshold(s)                                |
| <code>Panel.clip_lower(threshold)</code>                       | Return copy of the input with values below given value truncated |
| <code>Panel.clip_upper(threshold)</code>                       | Return copy of input with values above given value truncated     |
| <code>Panel.count([axis])</code>                               | Return number of observations over requested axis.               |
| <code>Panel.cummax([axis, dtype, out, skipna])</code>          | Return cumulative max over requested axis.                       |
| <code>Panel.cummin([axis, dtype, out, skipna])</code>          | Return cumulative min over requested axis.                       |
| <code>Panel.cumprod([axis, dtype, out, skipna])</code>         | Return cumulative prod over requested axis.                      |
| <code>Panel.cumsum([axis, dtype, out, skipna])</code>          | Return cumulative sum over requested axis.                       |
| <code>Panel.max([axis, skipna, level, numeric_only])</code>    | This method returns the maximum of the values in the object.     |
| <code>Panel.mean([axis, skipna, level, numeric_only])</code>   | Return the mean of the values for the requested axis             |
| <code>Panel.median([axis, skipna, level, numeric_only])</code> | Return the median of the values for the requested axis           |
| <code>Panel.min([axis, skipna, level, numeric_only])</code>    | This method returns the minimum of the values in the object.     |
| <code>Panel.pct_change([periods, fill_method, ...])</code>     | Percent change over given number of periods                      |
| <code>Panel.prod([axis, skipna, level, numeric_only])</code>   | Return the product of the values for the requested axis          |
| <code>Panel.skew([axis, skipna, level, numeric_only])</code>   | Return unbiased skew over requested axis                         |
| <code>Panel.sum([axis, skipna, level, numeric_only])</code>    | Return the sum of the values for the requested axis              |
| <code>Panel.std([axis, skipna, level, ddof])</code>            | Return unbiased standard deviation over requested axis           |
| <code>Panel.var([axis, skipna, level, ddof])</code>            | Return unbiased variance over requested 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.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, \*\*kws*)  
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

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.sum

`Panel.sum(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return the sum of the values for the requested axis

**Parameters** **axis** : {items (0), major\_axis (1), minor\_axis (2)}

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

**Returns** **sum** : DataFrame or Panel (if level specified)

## pandas.Panel.std

`Panel.std(axis=None, skipna=None, level=None, 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)

**27.5.9 Reindexing / Selection / Label manipulation**

|                                                               |                                                                       |
|---------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>Panel.add_prefix(prefix)</code>                         | Concatenate prefix string with panel items names.                     |
| <code>Panel.add_suffix(suffix)</code>                         | Concatenate suffix string with panel items names                      |
| <code>Panel.drop(labels[, axis, level, inplace])</code>       | Return new object with labels in requested axis removed               |
| <code>Panel.filter([items, like, regex, axis])</code>         | Restrict the info axis to set of items or wildcard                    |
| <code>Panel.first(offset)</code>                              | Convenience method for subsetting initial periods of time series data |
| <code>Panel.last(offset)</code>                               | Convenience method for subsetting final periods of time series data   |
| <code>Panel.reindex([items, major_axis, minor_axis])</code>   | Conform Panel to new index with optional filling logic, placing       |
| <code>Panel.reindex_axis(labels[, axis, method, ...])</code>  | Conform input object to new index with optional filling logic,        |
| <code>Panel.reindex_like(other[, method, copy, limit])</code> | return an object with matching indicies to myself                     |
| <code>Panel.rename([items, major_axis, minor_axis])</code>    | Alter axes input function or functions.                               |
| <code>Panel.select(crit[, axis])</code>                       | Return data corresponding to axis labels matching criteria            |
| <code>Panel.take(indices[, axis, convert, is_copy])</code>    | Analogous to ndarray.take                                             |
| <code>Panel.truncate([before, after, axis, copy])</code>      | Truncates a sorted NDFrame before and/or after some particular        |

**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 NoneMethod 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**

```
>>> 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

```
>>> 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

## 27.5.10 Missing data handling

---

|                                                                |                                                  |
|----------------------------------------------------------------|--------------------------------------------------|
| <code>Panel.dropna([axis, how, inplace])</code>                | Drop 2D from panel, holding passed axis constant |
| <code>Panel.fillna([value, method, axis, inplace, ...])</code> | 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`

## 27.5.11 Reshaping, sorting, transposing

|                                                   |                                                          |
|---------------------------------------------------|----------------------------------------------------------|
| <code>Panel.sort_index([axis, ascending])</code>  | Sort object by labels (along an axis)                    |
| <code>Panel.swaplevel(i, j[, axis])</code>        | Swap levels i and j in a MultiIndex on a particular axis |
| <code>Panel.transpose(*args, **kwargs)</code>     | Permute the dimensions of the Panel                      |
| <code>Panel.swapaxes(axis1, axis2[, copy])</code> | Interchange axes and swap values axes appropriately      |
| <code>Panel.conform(frame[, axis])</code>         | Conform input DataFrame to align with chosen axis pair.  |

### pandas.Panel.sort\_index

`Panel.sort_index` (*axis=0, ascending=True*)

Sort object by labels (along an axis)

**Parameters** **axis** : {0, 1}

Sort index/rows versus columns

**ascending** : boolean, default True

Sort ascending vs. descending

**Returns** **sorted\_obj** : type of caller

### pandas.Panel.swaplevel

`Panel.swaplevel` (*i, j, axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

**Parameters** **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

**Returns** **swapped** : type of caller (new object)

### pandas.Panel.transpose

`Panel.transpose` (*\*args, \*\*kwargs*)

Permute the dimensions of the Panel

**Parameters** **args** : three positional arguments: each one of

**{0,1,2,'items','major\_axis','minor\_axis'}**

**copy** : boolean, default False

Make a copy of the underlying data. Mixed-dtype data will always result in a copy

**Returns** **y** : same as input

### Examples

```
>>> 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

## 27.5.12 Combining / joining / merging

---

|                                                          |                                                                   |
|----------------------------------------------------------|-------------------------------------------------------------------|
| <code>Panel.join(other[, how, lsuffix, rsuffix])</code>  | Join items with other Panel either on major and minor axes column |
| <code>Panel.update(other[, join, overwrite, ...])</code> | Modify Panel in place using non-NA values from passed             |

---

## pandas.Panel.join

**Panel.join** (*other, how='left', lsuffix='', rsuffix=''*)

Join items with other Panel either on major and minor axes column

**Parameters** **other** : Panel or list of Panels

Index should be similar to one of the columns in this one

**how** : {'left', 'right', 'outer', 'inner'}

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise \* left: use calling frame's index \* right: use input frame's index \* outer: form union of indexes \* inner: use intersection of indexes

**lsuffix** : string

Suffix to use from left frame's overlapping columns

**rsuffix** : string

Suffix to use from right frame's overlapping columns

**Returns** **joined** : Panel

## pandas.Panel.update

**Panel.update** (*other*, *join*='left', *overwrite*=True, *filter\_func*=None, *raise\_conflict*=False)

Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

**Parameters** **other** : Panel, or object coercible to Panel**join** : How to join individual DataFrames

{ 'left', 'right', 'outer', 'inner' }, default 'left'

**overwrite** : boolean, default True

If True then overwrite values for common keys in the calling panel

**filter\_func** : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

**raise\_conflict** : bool

If True, will raise an error if a DataFrame and other both contain data in the same place.

## 27.5.13 Time series-related

|                                                             |                                                                             |
|-------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>Panel.asfreq(freq[, method, how, normalize])</code>   | Convert all TimeSeries inside to specified frequency using DateOffset       |
| <code>Panel.shift(lags[, freq, axis])</code>                | Shift major or minor axis by specified number of leads/lags.                |
| <code>Panel.resample(rule[, how, axis, ...])</code>         | Convenience method for frequency conversion and resampling of regular time- |
| <code>Panel.tz_convert(tz[, axis, copy])</code>             | Convert TimeSeries to target time zone. If it is time zone naive, it        |
| <code>Panel.tz_localize(tz[, axis, copy, infer_dst])</code> | Localize tz-naive TimeSeries to target time zone                            |

## 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 methdo

**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*, *loffset=None*, *limit=None*, *base=0*)

Convenience method for frequency conversion and resampling of regular time-series data.

**Parameters** **rule** : 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, fill\_method for upsampling, default None

**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

**27.5.14 Serialization / IO / Conversion**

|                                                                |                                                                        |
|----------------------------------------------------------------|------------------------------------------------------------------------|
| <code>Panel.from_dict(data[, intersect, orient, dtype])</code> | Construct Panel from dict of DataFrame objects                         |
| <code>Panel.to_pickle(path)</code>                             | Pickle (serialize) object to input file path                           |
| <code>Panel.to_excel(path[, na_rep, engine])</code>            | Write each DataFrame in Panel to a separate excel sheet                |
| <code>Panel.to_hdf(path_or_buf, key, **kwargs)</code>          | activate the HDFStore                                                  |
| <code>Panel.to_json([path_or_buf, orient, ...])</code>         | Convert the object to a JSON string.                                   |
| <code>Panel.to_sparse([fill_value, kind])</code>               | Convert to SparsePanel                                                 |
| <code>Panel.to_frame([filter_observations])</code>             | Transform wide format into long (stacked) format as DataFrame          |
| <code>Panel.to_clipboard([excel, sep])</code>                  | Attempt to write text representation of object to the system clipboard |

**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`, and  
`io.excel.xlsm.writer`.

**Other Parameters** `float_format` : string, default None

Format string for floating point numbers

`cols` : sequence, optional

Columns to write

`header` : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

`index` : boolean, default True

Write row names (index)

`index_label` : string or sequence, default None

Column label for index column(s) if desired. If None is given, and `header` and `index` are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

`startrow` : upper left cell row to dump data frame

`startcol` : upper left cell column to dump data frame

## Notes

Keyword arguments (and `na_rep`) are passed to the `to_excel` method for each DataFrame written.

**pandas.Panel.to\_hdf**

`Panel.to_hdf` (*path\_or\_buf*, *key*, *\*\*kwargs*)  
activate the HDFStore

**Parameters** *path\_or\_buf* : the path (string) or buffer to put the store

**key** : string, an 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** : type of date conversion, epoch or iso

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

**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

## 27.6 Index

Many of these methods or variants thereof are available on the objects that contain an index (Series/Dataframe) and those should most likely be used before calling these methods directly.

---

`Index` Immutable ndarray implementing an ordered, sliceable set.

---

### 27.6.1 pandas.Index

**class** `pandas.Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects

**Parameters** `data` : array-like (1-dimensional)

`dtype` : NumPy dtype (default: object)

`copy` : bool

Make a copy of input ndarray

`name` : object

Name to be stored in the index

### Notes

An Index instance can **only** contain hashable objects

### Attributes

|                           |                                                                                                                      |
|---------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>T</code>            | Same as <code>self.transpose()</code> , except that <code>self</code> is returned if <code>self.ndim &lt; 2</code> . |
| <code>base</code>         | Base object if memory is from some other object.                                                                     |
| <code>ctypes</code>       | An object to simplify the interaction of the array with the <code>ctypes</code> module.                              |
| <code>data</code>         | Python buffer object pointing to the start of the array's data.                                                      |
| <code>flags</code>        |                                                                                                                      |
| <code>flat</code>         | A 1-D iterator over the array.                                                                                       |
| <code>imag</code>         | The imaginary part of the array.                                                                                     |
| <code>is_monotonic</code> |                                                                                                                      |
| <code>itemsize</code>     | Length of one array element in bytes.                                                                                |
| <code>names</code>        |                                                                                                                      |
| <code>nbytes</code>       | Total bytes consumed by the elements of the array.                                                                   |
| <code>ndim</code>         | Number of array dimensions.                                                                                          |
| <code>nlevels</code>      |                                                                                                                      |
| <code>real</code>         | The real part of the array.                                                                                          |
| <code>shape</code>        | Tuple of array dimensions.                                                                                           |
| <code>size</code>         | Number of elements in the array.                                                                                     |
| <code>strides</code>      | Tuple of bytes to step in each dimension when traversing an array.                                                   |
| <code>values</code>       |                                                                                                                      |

## pandas.Index.T

### Index.T

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

### Examples

```
>>> 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`:

```
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> 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

```
>>> 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.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.

`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.Index.imag

`Index.imag`

The imaginary part of the array.

### Examples

```

>>> 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

```
>>> 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

```
>>> 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.]])
>>> 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

```
>>> 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  $(i[0], i[1], \dots, i[n])$  in an array  $a$  is:

```
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):

```
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

```
>>> 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=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

```

## pandas.Index.values

Index.**values**

|      |  |
|------|--|
| asi8 |  |
| name |  |

## Methods

|                                                   |                                                                                       |
|---------------------------------------------------|---------------------------------------------------------------------------------------|
| <code>all([axis, out])</code>                     | Returns True if all elements evaluate to True.                                        |
| <code>any([axis, out])</code>                     | Returns True if any of the elements of <i>a</i> evaluate to True.                     |
| <code>append(other)</code>                        | Append a collection of Index options together                                         |
| <code>argmax([axis, out])</code>                  | Return indices of the maximum values along the given axis.                            |
| <code>argmin([axis, out])</code>                  | Return indices of the minimum values along the given axis of <i>a</i> .               |
| <code>argsort(*args, **kwargs)</code>             | See docstring for ndarray.argsort                                                     |
| <code>asof(label)</code>                          | For a sorted index, return the most recent label up to and including the passed label |
| <code>asof_locs(where, mask)</code>               | where : array of timestamps                                                           |
| <code>astype(dtype)</code>                        |                                                                                       |
| <code>byteswap(inplace)</code>                    | Swap the bytes of the array elements                                                  |
| <code>choose(choices[, out, mode])</code>         | Use an index array to construct a new array from a set of choices.                    |
| <code>clip(a_min, a_max[, out])</code>            | Return an array whose values are limited to [a_min, a_max].                           |
| <code>compress(condition[, axis, out])</code>     | Return selected slices of this array along given axis.                                |
| <code>conj()</code>                               | Complex-conjugate all elements.                                                       |
| <code>conjugate()</code>                          | Return the complex conjugate, element-wise.                                           |
| <code>copy([names, name, dtype, deep])</code>     | Make a copy of this object.                                                           |
| <code>cumprod([axis, dtype, out])</code>          | Return the cumulative product of the elements along the given axis.                   |
| <code>cumsum([axis, dtype, out])</code>           | Return the cumulative sum of the elements along the given axis.                       |
| <code>delete(loc)</code>                          | Make new Index with passed location deleted                                           |
| <code>diagonal([offset, axis1, axis2])</code>     | Return specified diagonals.                                                           |
| <code>diff(other)</code>                          | Compute sorted set difference of two Index objects                                    |
| <code>dot(b[, out])</code>                        | Dot product of two arrays.                                                            |
| <code>drop(labels)</code>                         | Make new Index with passed list of labels deleted                                     |
| <code>dump(file)</code>                           | Dump a pickle of the array to the specified file.                                     |
| <code>dumps()</code>                              | Returns the pickle of the array as a string.                                          |
| <code>equals(other)</code>                        | Determines if two Index objects contain the same elements.                            |
| <code>fill(*args, **kwargs)</code>                | This method will not function because object is immutable.                            |
| <code>flatten([order])</code>                     | Return a copy of the array collapsed into one dimension.                              |
| <code>format([name, formatter])</code>            | Render a string representation of the Index                                           |
| <code>get_duplicates()</code>                     |                                                                                       |
| <code>get_indexer(target[, method, limit])</code> | Compute indexer and mask for new index given the current index.                       |

Continued on next page

Table 27.71 – continued from previous page

|                                                           |                                                                                                |
|-----------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>get_indexer_non_unique(target, **kwargs)</code>     | return an indexer suitable for taking from a non unique index                                  |
| <code>get_level_values(level)</code>                      | Return vector of label values for requested level, equal to the length                         |
| <code>get_loc(key)</code>                                 | Get integer location for requested label                                                       |
| <code>get_value(series, key)</code>                       | Fast lookup of value from 1-dimensional ndarray.                                               |
| <code>get_values()</code>                                 |                                                                                                |
| <code>getfield(dtype[, offset])</code>                    | Returns a field of the given array as a certain type.                                          |
| <code>groupby(to_groupby)</code>                          |                                                                                                |
| <code>holds_integer()</code>                              |                                                                                                |
| <code>identical(other)</code>                             | Similar to equals, but check that other comparable attributes are                              |
| <code>insert(loc, item)</code>                            | Make new Index inserting new item at location                                                  |
| <code>intersection(other)</code>                          | Form the intersection of two Index objects. Sortedness of the result is                        |
| <code>is_(other)</code>                                   | More flexible, faster check like <code>is</code> but that works through views                  |
| <code>is_floating()</code>                                |                                                                                                |
| <code>is_integer()</code>                                 |                                                                                                |
| <code>is_lexsorted_for_tuple(tup)</code>                  |                                                                                                |
| <code>is_mixed()</code>                                   |                                                                                                |
| <code>is_numeric()</code>                                 |                                                                                                |
| <code>is_type_compatible(typ)</code>                      |                                                                                                |
| <code>isin(values)</code>                                 | Compute boolean array of whether each index value is found in the                              |
| <code>item(*args)</code>                                  | Copy an element of an array to a standard Python scalar and return it.                         |
| <code>itemset(*args, **kwargs)</code>                     | This method will not function because object is immutable.                                     |
| <code>join(other[, how, level, return_indexers])</code>   | Internal API method. Compute <code>join_index</code> and <code>indexers</code> to conform data |
| <code>map mapper)</code>                                  |                                                                                                |
| <code>max([axis, out])</code>                             | Return the maximum along a given axis.                                                         |
| <code>mean([axis, dtype, out])</code>                     | Returns the average of the array elements along given axis.                                    |
| <code>min([axis, out])</code>                             | Return the minimum along a given axis.                                                         |
| <code>newbyteorder([new_order])</code>                    | Return the array with the same data viewed with a different byte order.                        |
| <code>nonzero()</code>                                    | Return the indices of the elements that are non-zero.                                          |
| <code>order([return_indexer, ascending])</code>           | Return sorted copy of Index                                                                    |
| <code>prod([axis, dtype, out])</code>                     | Return the product of the array elements over the given axis                                   |
| <code>ptp([axis, out])</code>                             | Peak to peak (maximum - minimum) value along a given axis.                                     |
| <code>put(*args, **kwargs)</code>                         | This method will not function because object is immutable.                                     |
| <code>ravel([order])</code>                               | Return a flattened array.                                                                      |
| <code>reindex(target[, method, level, limit, ...])</code> | For Index, simply returns the new index and the results of                                     |
| <code>rename(name[, inplace])</code>                      | Set new names on index.                                                                        |
| <code>repeat(repeats[, axis])</code>                      | Repeat elements of an array.                                                                   |
| <code>reshape(shape[, order])</code>                      | Returns an array containing the same data with a new shape.                                    |
| <code>resize(new_shape[, refcheck])</code>                | Change shape and size of array in-place.                                                       |
| <code>round([decimals, out])</code>                       | Return <i>a</i> with each element rounded to the given number of decimals.                     |
| <code>searchsorted(v[, side, sorter])</code>              | Find indices where elements of <i>v</i> should be inserted in <i>a</i> to maintain order.      |
| <code>set_names(names[, inplace])</code>                  | Set new names on index.                                                                        |
| <code>set_value(arr, key, value)</code>                   | Fast lookup of value from 1-dimensional ndarray.                                               |
| <code>setfield(val, dtype[, offset])</code>               | Put a value into a specified place in a field defined by a data-type.                          |
| <code>setflags([write, align, uic])</code>                | Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.                            |
| <code>shift([periods, freq])</code>                       | Shift Index containing datetime objects by input number of periods and                         |
| <code>slice_indexer([start, end, step])</code>            | For an ordered Index, compute the slice indexer for input labels and                           |
| <code>slice_locs([start, end])</code>                     | For an ordered Index, compute the slice locations for input labels                             |
| <code>sort(*args, **kwargs)</code>                        |                                                                                                |
| <code>squeeze([axis])</code>                              | Remove single-dimensional entries from the shape of <i>a</i> .                                 |
| <code>std([axis, dtype, out, ddof])</code>                | Returns the standard deviation of the array elements along given axis.                         |

Continued on next p

Table 27.71 – continued from previous page

|                                                        |                                                                                    |
|--------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>sum([axis, dtype, out])</code>                   | Return the sum of the array elements over the given axis.                          |
| <code>summary([name])</code>                           |                                                                                    |
| <code>swapaxes(axis1, axis2)</code>                    | Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.        |
| <code>take(indexer[, axis])</code>                     | Analogous to <code>ndarray.take</code>                                             |
| <code>to_datetime([dayfirst])</code>                   | For an Index containing strings or <code>datetime.datetime</code> objects, attempt |
| <code>to_native_types([slicer])</code>                 | slice and dice then format                                                         |
| <code>to_series()</code>                               | return a series with both index and values equal to the index keys                 |
| <code>tofile(fid[, sep, format])</code>                | Write array to a file as text or binary (default).                                 |
| <code>tolist()</code>                                  | Overridden version of <code>ndarray.tolist</code>                                  |
| <code>tostring([order])</code>                         | Construct a Python string containing the raw data bytes in the array.              |
| <code>trace([offset, axis1, axis2, dtype, out])</code> | Return the sum along diagonals of the array.                                       |
| <code>transpose(*axes)</code>                          | Returns a view of the array with axes transposed.                                  |
| <code>union(other)</code>                              | Form the union of two Index objects and sorts if possible                          |
| <code>unique()</code>                                  | Return array of unique values in the Index. Significantly faster than              |
| <code>var([axis, dtype, out, ddof])</code>             | Returns the variance of the array elements, along given axis.                      |
| <code>view(*args, **kwargs)</code>                     |                                                                                    |

**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.argmax**

**Index.argmax** (*axis=None, out=None*)

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmax* for detailed documentation.

**See Also:**

**numpy.argmax** 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

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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:

```
>>> 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'}, optionalWhether 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

```
>>> 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

```
>>> 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:

```
>>> 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 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[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 'bigish' 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=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.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=None, level=None, limit=None, copy\_if\_needed=False, take-able=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.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:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
 [1]])

>>> 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:

```
>>> 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...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> 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:**

`numpy.searchsorted` 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.set\_value**

`Index.set_value(arr, key, value)`

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

**pandas.Index.setfield**

`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**

```
>>> 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]])
>>> 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)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
```

### 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

```
>>> 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.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.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.

## 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

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'\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

```
>>> 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]])
>>> a.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.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)`

## 27.6.2 Modifying and Computations

|                                                          |                                                                       |
|----------------------------------------------------------|-----------------------------------------------------------------------|
| <code>Index.copy([names, name, dtype, deep])</code>      | Make a copy of this object.                                           |
| <code>Index.delete(loc)</code>                           | Make new Index with passed location deleted                           |
| <code>Index.diff(other)</code>                           | Compute sorted set difference of two Index objects                    |
| <code>Index.drop(labels)</code>                          | Make new Index with passed list of labels deleted                     |
| <code>Index.equals(other)</code>                         | Determines if two Index objects contain the same elements.            |
| <code>Index.identical(other)</code>                      | Similar to equals, but check that other comparable attributes are     |
| <code>Index.insert(loc, item)</code>                     | Make new Index inserting new item at location                         |
| <code>Index.order([return_indexer, ascending])</code>    | Return sorted copy of Index                                           |
| <code>Index.reindex(target[, method, level, ...])</code> | For Index, simply returns the new index and the results of            |
| <code>Index.repeat(repeats[, axis])</code>               | Repeat elements of an array.                                          |
| <code>Index.set_names(names[, inplace])</code>           | Set new names on index.                                               |
| <code>Index.unique()</code>                              | Return array of unique values in the Index. Significantly faster than |

### pandas.Index.copy

`Index.copy(names=None, name=None, dtype=None, deep=False)`

Make a copy of this object. Name and dtype sets those attributes on the new object.

**Parameters** `name` : string, optional

`dtype` : numpy dtype or pandas type

**Returns** `copy` : Index

#### Notes

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy.

### pandas.Index.delete

`Index.delete` (*loc*)

Make new Index with passed location 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

```
>>> 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

`item` : object

**Returns** `new_index` : Index

## 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

## 27.6.3 Conversion

|                                            |                                                                                    |
|--------------------------------------------|------------------------------------------------------------------------------------|
| <code>Index.astype(dtype)</code>           |                                                                                    |
| <code>Index.tolist()</code>                | Overridden version of <code>ndarray.tolist</code>                                  |
| <code>Index.to_datetime([dayfirst])</code> | For an Index containing strings or <code>datetime.datetime</code> objects, attempt |
| <code>Index.to_series()</code>             | return a series with both index and values equal to the index keys                 |



**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

**27.6.4 Sorting**

|                                                       |                                   |
|-------------------------------------------------------|-----------------------------------|
| <code>Index.argsort(*args, **kwargs)</code>           | See docstring for ndarray.argsort |
| <code>Index.order([return_indexer, ascending])</code> | Return sorted copy of Index       |
| <code>Index.sort(*args, **kwargs)</code>              |                                   |

**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)`**27.6.5 Time-specific operations**

|                                           |                                                                        |
|-------------------------------------------|------------------------------------------------------------------------|
| <code>Index.shift([periods, freq])</code> | Shift Index containing datetime objects by input number of periods and |
|-------------------------------------------|------------------------------------------------------------------------|

## pandas.Index.shift

`Index.shift` (*periods=1, freq=None*)

Shift Index containing datetime objects by input number of periods and DateOffset

**Returns** `shifted` : Index

## 27.6.6 Combining / joining / merging

|                                                                         |                                                                                                |
|-------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>Index.append</code> ( <i>other</i> )                              | Append a collection of Index options together                                                  |
| <code>Index.intersection</code> ( <i>other</i> )                        | Form the intersection of two Index objects. Sortedness of the result is                        |
| <code>Index.join</code> ( <i>other[, how, level, return_indexers]</i> ) | Internal API method. Compute <code>join_index</code> and <code>indexers</code> to conform data |
| <code>Index.union</code> ( <i>other</i> )                               | Form the union of two Index objects and sorts if possible                                      |

## 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

## 27.6.7 Selecting

|                                                             |                                                                        |
|-------------------------------------------------------------|------------------------------------------------------------------------|
| <code>Index.get_indexer(target[, method, limit])</code>     | Compute indexer and mask for new index given the current index.        |
| <code>Index.get_indexer_non_unique(target, **kwargs)</code> | return an indexer suitable for taking from a non unique index          |
| <code>Index.get_level_values(level)</code>                  | Return vector of label values for requested level, equal to the length |
| <code>Index.get_loc(key)</code>                             | Get integer location for requested label                               |
| <code>Index.get_value(series, key)</code>                   | Fast lookup of value from 1-dimensional ndarray.                       |
| <code>Index.isin(values)</code>                             | Compute boolean array of whether each index value is found in the      |
| <code>Index.slice_indexer([start, end, step])</code>        | For an ordered Index, compute the slice indexer for input labels and   |
| <code>Index.slice_locs([start, end])</code>                 | For an ordered Index, compute the slice locations for input labels     |

### 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

```
>>> 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.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

### 27.6.8 Properties

|                                 |
|---------------------------------|
| <code>Index.is_monotonic</code> |
| <code>Index.is_numeric()</code> |

#### **pandas.Index.is\_monotonic**

`Index.is_monotonic`

#### **pandas.Index.is\_numeric**

`Index.is_numeric()`

## 27.7 DatetimeIndex

|                            |                                                                            |
|----------------------------|----------------------------------------------------------------------------|
| <code>DatetimeIndex</code> | Immutable ndarray of datetime64 data, represented internally as int64, and |
|----------------------------|----------------------------------------------------------------------------|

### 27.7.1 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

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| T             | Same as self.transpose(), except that self is returned if self.ndim < 2.   |
| asi8          |                                                                            |
| asobject      | Convert to Index of datetime objects                                       |
| base          | Base object if memory is from some other object.                           |
| ctypes        | An object to simplify the interaction of the array with the ctypes module. |
| data          | Python buffer object pointing to the start of the array's data.            |
| date          | Returns numpy array of datetime.date.                                      |
| day           |                                                                            |
| dayofweek     |                                                                            |
| dayofyear     |                                                                            |
| dtype         |                                                                            |
| flags         |                                                                            |
| flat          | A 1-D iterator over the array.                                             |
| freq          |                                                                            |
| freqstr       |                                                                            |
| hour          |                                                                            |
| imag          | The imaginary part of the array.                                           |
| inferred_type |                                                                            |
| is_all_dates  |                                                                            |
| is_monotonic  |                                                                            |
| itemsize      | Length of one array element in bytes.                                      |
| microsecond   |                                                                            |
| minute        |                                                                            |
| month         |                                                                            |
| names         |                                                                            |
| nanosecond    |                                                                            |
| nbytes        | Total bytes consumed by the elements of the array.                         |
| ndim          | Number of array dimensions.                                                |
| nlevels       |                                                                            |
| quarter       |                                                                            |
| real          | The real part of the array.                                                |
| second        |                                                                            |
| shape         | Tuple of array dimensions.                                                 |
| size          | Number of elements in the array.                                           |
| strides       | Tuple of bytes to step in each dimension when traversing an array.         |
| time          | Returns numpy array of datetime.time.                                      |
| tzinfo        | Alias for tz attribute                                                     |
| values        |                                                                            |
| week          |                                                                            |
| weekday       |                                                                            |
| weekofyear    |                                                                            |
| year          |                                                                            |

**pandas.DatetimeIndex.T**`DatetimeIndex.T`Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.**Examples**

```
>>> 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.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`:

```
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

## pandas.DatetimeIndex.ctype

### DatetimeIndex.ctype

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).ctype.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).ctype`. 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

```

>>> 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.data

DatetimeIndex.**data**

Python buffer object pointing to the start of the array's data.

## pandas.DatetimeIndex.date

DatetimeIndex.**date**

Returns numpy array of datetime.date. The date part of the Timestamps.

## pandas.DatetimeIndex.day

DatetimeIndex.**day**

## pandas.DatetimeIndex.dayofweek

DatetimeIndex.**dayofweek**

## pandas.DatetimeIndex.dayofyear

DatetimeIndex.**dayofyear**

## pandas.DatetimeIndex.dtype

DatetimeIndex.**dtype**

### 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**

```
>>> 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**

```
>>> 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.DatetimeIndex.minute**

DatetimeIndex.**minute**

### **pandas.DatetimeIndex.month**

DatetimeIndex.**month**

### **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**

```
>>> 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**

```
>>> 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**

```
>>> 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**

```
>>> 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

```
>>> 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:

```
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):

```
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

```

>>> 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=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

```

## pandas.DatetimeIndex.time

DatetimeIndex.**time**

Returns numpy array of datetime.time. The time part of the Timestamps.

## pandas.DatetimeIndex.tzinfo

DatetimeIndex.**tzinfo**

Alias for tz attribute

## pandas.DatetimeIndex.values

DatetimeIndex.**values**

## pandas.DatetimeIndex.week

DatetimeIndex.**week**

## pandas.DatetimeIndex.weekday

DatetimeIndex.**weekday**

**pandas.DatetimeIndex.weekofyear**DatetimeIndex.**weekofyear****pandas.DatetimeIndex.year**DatetimeIndex.**year**

|        |  |
|--------|--|
| name   |  |
| offset |  |

**Methods**

|                                                       |                                                                                       |
|-------------------------------------------------------|---------------------------------------------------------------------------------------|
| <code>all([axis, out])</code>                         | Returns True if all elements evaluate to True.                                        |
| <code>any([axis, out])</code>                         | Returns True if any of the elements of <i>a</i> evaluate to True.                     |
| <code>append(other)</code>                            | Append a collection of Index options together                                         |
| <code>argmax([axis, out])</code>                      | Return indices of the maximum values along the given axis.                            |
| <code>argmin()</code>                                 |                                                                                       |
| <code>argsort(*args, **kwargs)</code>                 | See docstring for ndarray.argsort                                                     |
| <code>asof(label)</code>                              | For a sorted index, return the most recent label up to and including the passed label |
| <code>asof_locs(where, mask)</code>                   | where : array of timestamps                                                           |
| <code>astype(dtype)</code>                            |                                                                                       |
| <code>byteswap(inplace)</code>                        | Swap the bytes of the array elements                                                  |
| <code>choose(choices[, out, mode])</code>             | Use an index array to construct a new array from a set of choices.                    |
| <code>clip(a_min, a_max[, out])</code>                | Return an array whose values are limited to [a_min, a_max].                           |
| <code>compress(condition[, axis, out])</code>         | Return selected slices of this array along given axis.                                |
| <code>conj()</code>                                   | Complex-conjugate all elements.                                                       |
| <code>conjugate()</code>                              | Return the complex conjugate, element-wise.                                           |
| <code>copy([names, name, dtype, deep])</code>         | Make a copy of this object.                                                           |
| <code>cumprod([axis, dtype, out])</code>              | Return the cumulative product of the elements along the given axis.                   |
| <code>cumsum([axis, dtype, out])</code>               | Return the cumulative sum of the elements along the given axis.                       |
| <code>delete(loc)</code>                              | Make new DatetimeIndex with passed location deleted                                   |
| <code>diagonal([offset, axis1, axis2])</code>         | Return specified diagonals.                                                           |
| <code>diff(other)</code>                              | Compute sorted set difference of two Index objects                                    |
| <code>dot(b[, out])</code>                            | Dot product of two arrays.                                                            |
| <code>drop(labels)</code>                             | Make new Index with passed list of labels deleted                                     |
| <code>dump(file)</code>                               | Dump a pickle of the array to the specified file.                                     |
| <code>dumps()</code>                                  | Returns the pickle of the array as a string.                                          |
| <code>equals(other)</code>                            | Determines if two Index objects contain the same elements.                            |
| <code>fill(*args, **kwargs)</code>                    | This method will not function because object is immutable.                            |
| <code>flatten([order])</code>                         | Return a copy of the array collapsed into one dimension.                              |
| <code>format([name, formatter])</code>                | Render a string representation of the Index                                           |
| <code>get_duplicates()</code>                         |                                                                                       |
| <code>get_indexer(target[, method, limit])</code>     | Compute indexer and mask for new index given the current index.                       |
| <code>get_indexer_non_unique(target, **kwargs)</code> | return an indexer suitable for taking from a non unique index                         |
| <code>get_level_values(level)</code>                  | Return vector of label values for requested level, equal to the length of the index   |
| <code>get_loc(key)</code>                             | Get integer location for requested label                                              |
| <code>get_value(series, key)</code>                   | Fast lookup of value from 1-dimensional ndarray.                                      |
| <code>get_value_maybe_box(series, key)</code>         |                                                                                       |
| <code>get_values()</code>                             |                                                                                       |

Continued on



Table 27.81 – continued from previous page

|                                                                |                                                                                      |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <code>getfield(dtype[, offset])</code>                         | Returns a field of the given array as a certain type.                                |
| <code>groupby(f)</code>                                        |                                                                                      |
| <code>holds_integer()</code>                                   |                                                                                      |
| <code>identical(other)</code>                                  | Similar to equals, but check that other comparable attributes are                    |
| <code>indexer_at_time(time[, asof])</code>                     | Select values at particular time of day (e.g.                                        |
| <code>indexer_between_time(start_time, end_time[, ...])</code> | Select values between particular times of day (e.g., 9:00-9:30AM)                    |
| <code>insert(loc, item)</code>                                 | Make new Index inserting new item at location                                        |
| <code>intersection(other)</code>                               | Specialized intersection for DatetimeIndex objects. May be much faster               |
| <code>is_(other)</code>                                        | More flexible, faster check like <code>is</code> but that works through views        |
| <code>is_floating()</code>                                     |                                                                                      |
| <code>is_integer()</code>                                      |                                                                                      |
| <code>is_lexsorted_for_tuple(tup)</code>                       |                                                                                      |
| <code>is_mixed()</code>                                        |                                                                                      |
| <code>is_numeric()</code>                                      |                                                                                      |
| <code>is_type_compatible(typ)</code>                           |                                                                                      |
| <code>isin(values)</code>                                      | Compute boolean array of whether each index value is found in the                    |
| <code>item(*args)</code>                                       | Copy an element of an array to a standard Python scalar and return it.               |
| <code>itemset(*args, **kwargs)</code>                          | This method will not function because object is immutable.                           |
| <code>join(other[, how, level, return_indexers])</code>        | See <code>Index.join</code>                                                          |
| <code>map(f)</code>                                            |                                                                                      |
| <code>max([axis])</code>                                       | Overridden <code>ndarray.max</code> to return a Timestamp                            |
| <code>mean([axis, dtype, out])</code>                          | Returns the average of the array elements along given axis.                          |
| <code>min([axis])</code>                                       | Overridden <code>ndarray.min</code> to return a Timestamp                            |
| <code>newbyteorder([new_order])</code>                         | Return the array with the same data viewed with a different byte order.              |
| <code>nonzero()</code>                                         | Return the indices of the elements that are non-zero.                                |
| <code>normalize()</code>                                       | Return DatetimeIndex with times to midnight. Length is unaltered                     |
| <code>order([return_indexer, ascending])</code>                | Return sorted copy of Index                                                          |
| <code>prod([axis, dtype, out])</code>                          | Return the product of the array elements over the given axis                         |
| <code>ptp([axis, out])</code>                                  | Peak to peak (maximum - minimum) value along a given axis.                           |
| <code>put(*args, **kwargs)</code>                              | This method will not function because object is immutable.                           |
| <code>ravel([order])</code>                                    | Return a flattened array.                                                            |
| <code>reindex(target[, method, level, limit, ...])</code>      | For Index, simply returns the new index and the results of                           |
| <code>rename(name[, inplace])</code>                           | Set new names on index.                                                              |
| <code>repeat(repeats[, axis])</code>                           | Analogous to <code>ndarray.repeat</code>                                             |
| <code>reshape(shape[, order])</code>                           | Returns an array containing the same data with a new shape.                          |
| <code>resize(new_shape[, refcheck])</code>                     | Change shape and size of array in-place.                                             |
| <code>round([decimals, out])</code>                            | Return <i>a</i> with each element rounded to the given number of decimals.           |
| <code>searchsorted(key[, side])</code>                         |                                                                                      |
| <code>set_names(names[, inplace])</code>                       | Set new names on index.                                                              |
| <code>set_value(arr, key, value)</code>                        | Fast lookup of value from 1-dimensional ndarray.                                     |
| <code>setfield(val, dtype[, offset])</code>                    | Put a value into a specified place in a field defined by a data-type.                |
| <code>setflags([write, align, uic])</code>                     | Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively                   |
| <code>shift(n[, freq])</code>                                  | Specialized shift which produces a DatetimeIndex                                     |
| <code>slice_indexer([start, end, step])</code>                 | <code>Index.slice_indexer</code> , customized to handle time slicing                 |
| <code>slice_locs([start, end])</code>                          | <code>Index.slice_locs</code> , customized to handle partial ISO-8601 string slicing |
| <code>snap([freq])</code>                                      | Snap time stamps to nearest occurring frequency                                      |
| <code>sort(*args, **kwargs)</code>                             |                                                                                      |
| <code>squeeze([axis])</code>                                   | Remove single-dimensional entries from the shape of <i>a</i> .                       |
| <code>std([axis, dtype, out, ddof])</code>                     | Returns the standard deviation of the array elements along given axis.               |
| <code>sum([axis, dtype, out])</code>                           | Return the sum of the array elements over the given axis.                            |

Continued on

Table 27.81 – continued from previous page

|                                                        |                                                                                                            |
|--------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>summary([name])</code>                           |                                                                                                            |
| <code>swapaxes(axis1, axis2)</code>                    | Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.                                |
| <code>take(indices[, axis])</code>                     | Analogous to <code>ndarray.take</code>                                                                     |
| <code>to_datetime([dayfirst])</code>                   |                                                                                                            |
| <code>to_native_types([slicer])</code>                 | slice and dice then format                                                                                 |
| <code>to_period([freq])</code>                         | Cast to <code>PeriodIndex</code> at a particular frequency                                                 |
| <code>to_pydatetime()</code>                           | Return <code>DatetimeIndex</code> as object <code>ndarray</code> of <code>datetime.datetime</code> objects |
| <code>to_series()</code>                               | return a series with both index and values equal to the index keys                                         |
| <code>tofile(fid[, sep, format])</code>                | Write array to a file as text or binary (default).                                                         |
| <code>tolist()</code>                                  | See <code>ndarray.tolist</code>                                                                            |
| <code>tostring([order])</code>                         | Construct a Python string containing the raw data bytes in the array.                                      |
| <code>trace([offset, axis1, axis2, dtype, out])</code> | Return the sum along diagonals of the array.                                                               |
| <code>transpose(*axes)</code>                          | Returns a view of the array with axes transposed.                                                          |
| <code>tz_convert(tz)</code>                            | Convert <code>DatetimeIndex</code> from one time zone to another (using <code>pytz</code> )                |
| <code>tz_localize(tz[, infer_dst])</code>              | Localize tz-naive <code>DatetimeIndex</code> to given time zone (using <code>pytz</code> )                 |
| <code>union(other)</code>                              | Specialized union for <code>DatetimeIndex</code> objects. If combine                                       |
| <code>union_many(others)</code>                        | A bit of a hack to accelerate unioning a collection of indexes                                             |
| <code>unique()</code>                                  | <code>Index.unique</code> with handling for <code>DatetimeIndex</code> metadata                            |
| <code>var([axis, dtype, out, ddof])</code>             | Returns the variance of the array elements, along given axis.                                              |
| <code>view(*args, **kwargs)</code>                     |                                                                                                            |

**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

```
>>> 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

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
 dtype='<S3')
```

### pandas.DatetimeIndex.choose

`DatetimeIndex.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

`DatetimeIndex.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

`DatetimeIndex.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

```
>>> 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

```
>>> 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:

```
>>> 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.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.DatetimeIndex.format`

`DatetimeIndex.format` (*name*=False, *formatter*=None, *\*\*kwargs*)

Render a string representation of the Index

### `pandas.DatetimeIndex.get_duplicates`

`DatetimeIndex.get_duplicates` ()

### `pandas.DatetimeIndex.get_indexer`

`DatetimeIndex.get_indexer` (*target*, *method*=None, *limit*=None)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to `ndarray.take` to align the current data to the new index. 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

```
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

### **pandas.DatetimeIndex.get\_indexer\_non\_unique**

`DatetimeIndex.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**

`DatetimeIndex.get_level_values` (*level*)

Return vector of label values for requested level, equal to the length of the index

**Parameters** *level* : int

**Returns** *values* : ndarray

### **pandas.DatetimeIndex.get\_loc**

`DatetimeIndex.get_loc` (*key*)

Get integer location for requested label

**Returns** *loc* : int

### **pandas.DatetimeIndex.get\_value**

`DatetimeIndex.get_value` (*series*, *key*)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

### **pandas.DatetimeIndex.get\_value\_maybe\_box**

`DatetimeIndex.get_value_maybe_box` (*series*, *key*)

### **pandas.DatetimeIndex.get\_values**

`DatetimeIndex.get_values` ()

### 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

```
>>> 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:

```
>>> 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

**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[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.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 `ndarray.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.newbytorder(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 `'bigish'` are valid to specify big-endian.

**Returns** `new_arr` : array

New array object with the dtype 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**

`DatetimeIndex.normalize` ()

Return `DatetimeIndex` with times to midnight. Length is unaltered

**Returns** `normalized` : `DatetimeIndex`

**pandas.DatetimeIndex.order**

`DatetimeIndex.order` (*return\_indexer=False, ascending=True*)  
Return sorted copy of Index

**pandas.DatetimeIndex.prod**

`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**

`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**

`DatetimeIndex.put` (*\*args, \*\*kwargs*)  
This method will not function because object is immutable.

**pandas.DatetimeIndex.ravel**

`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:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
 [1]])

>>> 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:

```
>>> 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...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

## pandas.DatetimeIndex.round

`DatetimeIndex.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.DatetimeIndex.searchsorted**

`DatetimeIndex.searchsorted` (*key*, *side*='left')

### **pandas.DatetimeIndex.set\_names**

`DatetimeIndex.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.DatetimeIndex.set\_value**

`DatetimeIndex.set_value` (*arr*, *key*, *value*)

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

### **pandas.DatetimeIndex.setfield**

`DatetimeIndex.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

```

>>> 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]])
>>> 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)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])

```

## 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 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

```
>>> 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

DatetimeIndex.**shift** (*n*, *freq=None*)

Specialized shift which produces a DatetimeIndex

**Parameters** *n* : int

Periods to shift by

**freq** : DateOffset or timedelta-like, optional

**Returns** **shifted** : DatetimeIndex

### pandas.DatetimeIndex.slice\_indexer

DatetimeIndex.**slice\_indexer** (*start=None*, *end=None*, *step=None*)

Index.slice\_indexer, customized to handle time slicing

### pandas.DatetimeIndex.slice\_locs

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:**

`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` (*licer=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

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\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.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.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

```
>>> 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]])
>>> 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

### 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*)

## **27.7.2 Time/Date Components**

---

`DatetimeIndex.year`

`DatetimeIndex.month`

Continued on next page

**Table 27.82 – continued from previous page**

|                                        |                                       |
|----------------------------------------|---------------------------------------|
| <code>DatetimeIndex.day</code>         |                                       |
| <code>DatetimeIndex.hour</code>        |                                       |
| <code>DatetimeIndex.minute</code>      |                                       |
| <code>DatetimeIndex.second</code>      |                                       |
| <code>DatetimeIndex.microsecond</code> |                                       |
| <code>DatetimeIndex.nanosecond</code>  |                                       |
| <code>DatetimeIndex.date</code>        | Returns numpy array of datetime.date. |
| <code>DatetimeIndex.time</code>        | Returns numpy array of datetime.time. |
| <code>DatetimeIndex.dayofyear</code>   |                                       |
| <code>DatetimeIndex.weekofyear</code>  |                                       |
| <code>DatetimeIndex.week</code>        |                                       |
| <code>DatetimeIndex.dayofweek</code>   |                                       |
| <code>DatetimeIndex.weekday</code>     |                                       |
| <code>DatetimeIndex.quarter</code>     |                                       |

**pandas.DatetimeIndex.year**`DatetimeIndex.year`**pandas.DatetimeIndex.month**`DatetimeIndex.month`**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`**pandas.DatetimeIndex.weekday**`DatetimeIndex.weekday`**pandas.DatetimeIndex.quarter**`DatetimeIndex.quarter`**27.7.3 Selecting**

|                                                             |                                                                   |
|-------------------------------------------------------------|-------------------------------------------------------------------|
| <code>DatetimeIndex.indexer_at_time(time[, asof])</code>    | Select values at particular time of day (e.g.                     |
| <code>DatetimeIndex.indexer_between_time(...[, ...])</code> | Select values between particular times of day (e.g., 9:00-9:30AM) |

**pandas.DatetimeIndex.indexer\_at\_time**`DatetimeIndex.indexer_at_time` (*time*, *asof*=False)

Select values at particular time of day (e.g. 9:30AM)

**Parameters** `time` : `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

## **27.7.4 Time-specific operations**

|                                                         |                                                                  |
|---------------------------------------------------------|------------------------------------------------------------------|
| <code>DatetimeIndex.normalize()</code>                  | Return DatetimeIndex with times to midnight. Length is unaltered |
| <code>DatetimeIndex.snap([freq])</code>                 | Snap time stamps to nearest occurring frequency                  |
| <code>DatetimeIndex.tz_convert(tz)</code>               | Convert DatetimeIndex from one time zone to another (using pytz) |
| <code>DatetimeIndex.tz_localize(tz[, infer_dst])</code> | Localize tz-naive DatetimeIndex to given time zone (using pytz)  |

### **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`

## 27.7.5 Conversion

|                                                    |                                                                                               |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <code>DatetimeIndex.to_datetime([dayfirst])</code> |                                                                                               |
| <code>DatetimeIndex.to_period([freq])</code>       | Cast to <code>PeriodIndex</code> at a particular frequency                                    |
| <code>DatetimeIndex.to_pydatetime()</code>         | Return <code>DatetimeIndex</code> as object ndarray of <code>datetime.datetime</code> objects |

### `pandas.DatetimeIndex.to_datetime`

`DatetimeIndex.to_datetime(dayfirst=False)`

### `pandas.DatetimeIndex.to_period`

`DatetimeIndex.to_period(freq=None)`  
Cast to `PeriodIndex` at a particular frequency

### `pandas.DatetimeIndex.to_pydatetime`

`DatetimeIndex.to_pydatetime()`  
Return `DatetimeIndex` as object ndarray of `datetime.datetime` objects

**Returns** `datetimes` : ndarray

## 27.8 GroupBy

`GroupBy` objects are returned by `groupby` calls: `pandas.DataFrame.groupby()`, `pandas.Series.groupby()`, etc.

### 27.8.1 Indexing, iteration

|                                             |                                                               |
|---------------------------------------------|---------------------------------------------------------------|
| <code>GroupBy.__iter__()</code>             | Groupby iterator                                              |
| <code>GroupBy.groups</code>                 | dict {group name -> group labels}                             |
| <code>GroupBy.indices</code>                | dict {group name -> group indices}                            |
| <code>GroupBy.get_group(name[, obj])</code> | Constructs <code>NDFrame</code> from group with provided name |

### `pandas.core.groupby.GroupBy.__iter__`

`GroupBy.__iter__()`  
Groupby iterator

**Returns** Generator yielding sequence of (name, subsetted object)  
for each group

### **pandas.core.groupby.GroupBy.groups**

GroupBy.**groups**  
dict {group name -> group labels}

### **pandas.core.groupby.GroupBy.indices**

GroupBy.**indices**  
dict {group name -> group indices}

### **pandas.core.groupby.GroupBy.get\_group**

GroupBy.**get\_group** (*name*, *obj=None*)  
Constructs NDFrame from group with provided name

**Parameters** **name** : object  
the name of the group to get as a DataFrame  
**obj** : NDFrame, default None  
the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

**Returns** **group** : type of obj

## **27.8.2 Function application**

---

|                                                   |                                                                    |
|---------------------------------------------------|--------------------------------------------------------------------|
| GroupBy. <b>apply</b> (func, *args, **kwargs)     | Apply function and combine results together in an intelligent way. |
| GroupBy. <b>aggregate</b> (func, *args, **kwargs) |                                                                    |
| GroupBy. <b>transform</b> (func, *args, **kwargs) |                                                                    |

---

### **pandas.core.groupby.GroupBy.apply**

GroupBy.**apply** (*func*, \*args, \*\*kwargs)  
Apply function and combine results together in an intelligent way. The split-apply-combine combination rules attempt to be as common sense based as possible. For example:

case 1: group DataFrame apply aggregation function (f(chunk) -> Series) yield DataFrame, with group axis having group labels

case 2: group DataFrame apply transform function ((f(chunk) -> DataFrame with same indexes) yield DataFrame with resulting chunks glued together

case 3: group Series apply function with f(chunk) -> DataFrame yield DataFrame with result of chunks glued together

**Parameters** **func** : function

**Returns** **applied** : type depending on grouped object and function

**See Also:**`aggregate, transform`**Notes**

See online documentation for full exposition on how to use `apply`

**pandas.core.groupby.GroupBy.aggregate**

`GroupBy.aggregate(func, *args, **kwargs)`

**pandas.core.groupby.GroupBy.transform**

`GroupBy.transform(func, *args, **kwargs)`

**27.8.3 Computations / Descriptive Stats**

|                                  |                                                                |
|----------------------------------|----------------------------------------------------------------|
| <code>GroupBy.mean()</code>      | Compute mean of groups, excluding missing values               |
| <code>GroupBy.median()</code>    | Compute median of groups, excluding missing values             |
| <code>GroupBy.std([ddof])</code> | Compute standard deviation of groups, excluding missing values |
| <code>GroupBy.var([ddof])</code> | Compute variance of groups, excluding missing values           |
| <code>GroupBy.ohlc()</code>      | Compute sum of values, excluding missing values                |

**pandas.core.groupby.GroupBy.mean**

`GroupBy.mean()`

Compute mean of groups, excluding missing values

For multiple groupings, the result index will be a `MultiIndex`

**pandas.core.groupby.GroupBy.median**

`GroupBy.median()`

Compute median of groups, excluding missing values

For multiple groupings, the result index will be a `MultiIndex`

**pandas.core.groupby.GroupBy.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

**prefix** : string, default None

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

```
>>> 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
```

## 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

**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, **kwargs)`

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 → 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, **kwargs)`

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)

**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

**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.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 BeautifulSoup 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, and 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

```
pandas.io.json.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

**pandas.io.parsers.read\_csv**

```
pandas.io.parsers.read_csv(filepath_or_buffer, sep=',', dialect=None, compression=None,
doublequote=True, escapechar=None, quotechar='"', quoting=0,
skipinitialspace=False, lineterminator=None, header='infer',
index_col=None, names=None, prefix=None, skiprows=None, skip-
footer=None, skip_footer=0, na_values=None, na_fvalues=None,
true_values=None, false_values=None, delimiter=None, con-
verters=None, dtype=None, usecols=None, engine='c', de-
lim_whitespace=False, as_reccarray=False, na_filter=True,
compact_ints=False, use_unsigned=False, low_memory=True,
buffer_lines=None, warn_bad_lines=True, error_bad_lines=True,
keep_default_na=True, thousands=None, comment=None, deci-
mal='.', parse_dates=False, keep_date_col=False, dayfirst=False,
date_parser=None, memory_map=False, nrows=None, itera-
tor=False, chunksize=None, verbose=False, encoding=None,
squeeze=False, mangle_dupe_cols=True, tupleize_cols=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

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

**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'

**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)

**Returns** **result** : DataFrame or TextParser

## pandas.io.parsers.read\_fwf

`pandas.io.parsers.read_fwf` (*filepath\_or\_buffer*, *colspecs*='infer', *widths*=None, *\*\*kwargs*)

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

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

**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'

**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)

**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**

```
pandas.io.parsers.read_table(filepath_or_buffer, sep='\t', dialect=None, compression=None,
 doublequote=True, escapechar=None, quotechar='"', quoting=0,
 skipinitialspace=False, lineterminator=None, header='infer',
 index_col=None, names=None, prefix=None, skiprows=None, skip
 footer=None, skip_footer=0, na_values=None, na_fvalues=None,
 true_values=None, false_values=None, delimiter=None, con
 verters=None, dtype=None, usecols=None, engine='c', de
 lim_whitespace=False, as_reccarray=False, na_filter=True,
 compact_ints=False, use_unsigned=False, low_memory=True,
 buffer_lines=None, warn_bad_lines=True, error_bad_lines=True,
 keep_default_na=True, thousands=None, comment=None, deci
 mal='.', parse_dates=False, keep_date_col=False, dayfirst=False,
 date_parser=None, memory_map=False, nrows=None, itera
 tor=False, chunksize=None, verbose=False, encoding=None,
 squeeze=False, mangle_dupe_cols=True, tupleize_cols=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 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

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

**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'

**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)

**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 representation

**chunksize** : size to chunk the writing

**expectedrows** : expected TOTAL row size of this table

**encoding** : default None, provide an encoding for strings

**dropna** : boolean, default True, do not write an ALL nan row to the store settable by the option 'io.hdf.dropna\_table'

**Notes**

—

**Does \*not\* check if data being appended overlaps with existing data in the table, so be careful**

## pandas.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.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:  $\alpha = 1/(1 + com)$ ,

**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.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:  $\alpha = 1/(1 + com)$ ,

**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.stats.moments.ewmcov

pandas.stats.moments.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:  $\alpha = 1/(1 + com)$ ,

**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.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 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

`pandas.stats.moments.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, :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 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 -> 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* : type of input argument

### 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*

**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, center=False, time\_rule=None, \*\*kwargs*)

$O(N \log(\text{window}))$  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, center=False, time\_rule=None*)

Moving quantile

**Parameters** `arg` : Series, DataFrame

**window** : Number of observations used for calculating statistic

**quantile** :  $0 \leq \text{quantile} \leq 1$

**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

## 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

```
>>> A >>> B
 lkey value rkey value
0 foo 1 0 foo 5
1 bar 2 1 bar 6
2 baz 3 2 qux 7
3 foo 4 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.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

```
>>> 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)
>>> 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*, *unit*='ns')

Convert argument to datetime

**Parameters** **arg** : string, datetime, array of strings (with possible NAs)

**errors** : {'ignore', 'raise'}, default 'ignore'

Errors are ignored by default (values left untouched)

**dayfirst** : boolean, default False

If True parses dates with the day first, eg 20/01/2005 Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug).

**utc** : boolean, default None

Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well)

**box** : boolean, default True

If True returns a DatetimeIndex, if False returns ndarray of values

**format** : string, default None

strftime to parse time, eg “%d/%m/%Y”

**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

**Returns** **ret** : datetime if parsing succeeded



# RELEASE 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>

## 28.1 pandas 0.13.0

**Release date:** not-yet-released

### 28.1.1 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](#))

### 28.1.2 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 `qtandas DataFrameModel` and `DataFrameWidget`.
- Added `pandas.io.gbq` for reading from (and writing to) Google BigQuery into a `DataFrame`. ([GH4140](#))

### 28.1.3 Improvements to existing features

- `read_html` now raises a `URLError` instead of catching and raising a `ValueError` ([GH4303](#), [GH4305](#))
- `read_excel` now supports an integer in its `sheetname` argument giving the index of the sheet to read in ([GH4301](#)).
- `get_dummies` works with `NaN` ([GH4446](#))
- Added a test for `read_clipboard()` and `to_clipboard()` ([GH4282](#))
- Added `bins` argument to `value_counts` ([GH3945](#)), also `sort` and `ascending`, now available in `Series` method as well as top-level function.
- Text parser now treats anything that reads like `inf` ("`inf`", "`Inf`", "`-Inf`", "`iNf`", etc.) to infinity. ([GH4220](#), [GH4219](#)), affecting `read_table`, `read_csv`, etc.
- Added a more informative error message when plot arguments contain overlapping color and style arguments ([GH4402](#))
- Significant table writing performance improvements in `HDFStore`
- JSON date serialisation 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` dtype `Series`. This is frequency conversion; astyping is also supported.
- `Timedelta64` support `fillna/ffill/bfill` with an integer interpreted as seconds, or a `timedelta` (GH3371)
- Box numeric ops on `timedelta` `Series` (GH4984)
- `Datetime64` support `ffill/bfill`
- Performance improvements with `__getitem__` on `DataFrames` with when the key is a column
- Support for using a `DatetimeIndex/PeriodsIndex` directly in a datelike calculation e.g. `s.s.index` (GH4629)
- Better/cleaned up exceptions in `core/common`, `io/excel` and `core/format` (GH4721, GH3954), as well as cleaned up test cases in `tests/test_frame`, `tests/test_multilevel` (GH4732).
- Performance improvement of timeseries plotting with `PeriodIndex` and added test to `vbench` (GH4705 and GH4722)
- Add `axis` and `level` keywords to `where`, so that the other argument can now be an alignable pandas object.
- `to_datetime` with a format of `'%Y%m%d'` now parses much faster
- It's now easier to hook new Excel writers into pandas (just subclass `ExcelWriter` and register your engine). You can specify an engine in `to_excel` or in `ExcelWriter`. You can also specify which writers you want to use by default with config options `io.excel.xlsx.writer` and `io.excel.xls.writer`. (GH4745, GH4750)
- `Panel.to_excel()` now accepts keyword arguments that will be passed to its `DataFrame's to_excel()` methods. (GH4750)
- Added `XlsxWriter` as an optional `ExcelWriter` engine. This is about 5x faster than the default `openpyxl` `xlsx` writer and is equivalent in speed to the `xlwt` `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 `xlrd.Book` for the `io` (formerly `path_or_buf`) argument; this requires engine to be set. (GH4961).
- `concat` now gives a more informative error message when passed objects that cannot be concatenated (GH4608).

- Add `halflife` option to exponentially weighted moving functions (PR [GH4998](#))
- `to_dict` now takes `records` as a possible outtype. Returns an array of column-keyed dictionaries. ([GH4936](#))
- `tz_localize` can infer a fall daylight savings transition based on the structure of unlocalized data ([GH4230](#))
- `DatetimeIndex` is now in the API documentation
- Improve support for converting R datasets to pandas objects (more informative index for timeseries and numeric, support for factors, dist, and high-dimensional arrays).
- `read_html()` now supports the `parse_dates`, `tupleize_cols` and `thousands` parameters ([GH4770](#)).
- `json_normalize()` is a new method to allow you to create a flat table from semi-structured JSON data. *See the docs* ([GH1067](#))
- `DataFrame.from_records()` will now accept generators ([GH4910](#))
- `DataFrame.interpolate()` and `Series.interpolate()` have been expanded to include interpolation methods from `scipy`. ([GH4434](#), [GH1892](#))
- `Series` now supports a `to_frame` method to convert it to a single-column `DataFrame` ([GH5164](#))
- `DatetimeIndex` (and `date_range`) can now be constructed in a left- or right-open fashion using the `closed` parameter ([GH4579](#))
- Python csv parser now supports `usecols` ([GH4335](#))
- Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. ([GH5271](#))
- `NDFrame.drop()` now accepts names as well as integers for the axis argument. ([GH5354](#))
- Added short docstrings to a few methods that were missing them + fixed the docstrings for Panel flex methods. ([GH5336](#))
- `NDFrame.drop()`, `NDFrame.dropna()`, and `.drop_duplicates()` all accept `inplace` as a keyword argument; however, this only means that the wrapper is updated inplace, a copy is still made internally. ([GH1960](#), [GH5247](#), [GH5628](#), and related [GH2325](#) [still not closed])
- Fixed bug in `tools.plotting.andrews_curves` so that lines are drawn grouped by color as expected.
- `read_excel()` now tries to convert integral floats (like `1.0`) to `int` by default. ([GH5394](#))
- Excel writers now have a default option `merge_cells` in `to_excel()` to merge cells in `MultiIndex` and `Hierarchical Rows`. Note: using this option it is no longer possible to round trip Excel files with merged `MultiIndex` and `Hierarchical Rows`. Set the `merge_cells` to `False` to restore the previous behaviour. ([GH5254](#))
- The FRED `DataReader` now accepts multiple series (:issue‘3413’)
- `StataWriter` adjusts variable names to Stata’s limitations ([GH5709](#))

## 28.1.4 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 `@jtratrner`. As a result, `pandas` now uses iterators more extensively. This also led to the introduction of substantive parts of the Benjamin Peterson’s `six` library into `compat`. ([GH4384](#), [GH4375](#), [GH4372](#))
- `pandas.util.compat` and `pandas.util.py3compat` have been merged into `pandas.compat`. `pandas.compat` now includes many functions allowing 2/3 compatibility. It contains both list and iterator versions of `range`, `filter`, `map` and `zip`, plus other necessary



elements for Python 3 compatibility. `lmap`, `lzip`, `lrange` and `lfilter` all produce lists instead of iterators, for compatibility with `numpy`, subscripting and `pandas` constructors. (GH4384, GH4375, GH4372)

- deprecated `iterkv`, which will be removed in a future release (was just an alias of `iteritems` used to get around 2to3's changes). (GH4384, GH4375, GH4372)
- `Series.get` with negative indexers now returns the same as `[]` (GH4390)
- allow `ix/loc` for `Series/DataFrame/Panel` to set on any axis even when the single-key is not currently contained in the index for that axis (GH2578, GH5226, GH5632, GH5720, GH5744, GH5756)
- Default export for `to_clipboard` is now `csv` with a sep of `t` for compat (GH3368)
- `at` now will enlarge the object inplace (and return the same) (GH2578)
- `DataFrame.plot` will scatter plot `x` versus `y` by passing `kind='scatter'` (GH2215)
- `HDFStore`
  - `append_to_multiple` automatically synchronizes writing rows to multiple tables and adds a `dropna` kwarg (GH4698)
  - handle a passed `Series` in table format (GH4330)
  - added an `is_open` property to indicate if the underlying file handle is open; a closed store will now report 'CLOSED' when viewing the store (rather than raising an error) (GH4409)
  - a close of a `HDFStore` now will close that instance of the `HDFStore` but will only close the actual file if the ref count (by `PyTables`) w.r.t. all of the open handles are 0. Essentially you have a local instance of `HDFStore` referenced by a variable. Once you close it, it will report closed. Other references (to the same file) will continue to operate until they themselves are closed. Performing an action on a closed file will raise `ClosedFileError`
  - removed the `_quiet` attribute, replace by a `DuplicateWarning` if retrieving duplicate rows from a table (GH4367)
  - removed the `warn` argument from `open`. Instead a `PossibleDataLossError` exception will be raised if you try to use `mode='w'` with an OPEN file handle (GH4367)
  - allow a passed locations array or mask as a `where` condition (GH4467)
  - add the keyword `dropna=True` to `append` to change whether ALL nan rows are not written to the store (default is `True`, ALL nan rows are NOT written), also settable via the option `io.hdf.dropna_table` (GH4625)
  - the `format` keyword now replaces the `table` keyword; allowed values are `fixed(f) | table(t)` the `Storer` format has been renamed to `Fixed`
  - a column multi-index will be recreated properly (GH4710); raise on trying to use a multi-index with `data_columns` on the same axis
  - `select_as_coordinates` will now return an `Int64Index` of the resultant selection set
  - support `timedelta64[ns]` as a serialization type (GH3577)
  - store `datetime.date` objects as ordinals rather than `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 meta-data 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 list can be passed to `to_replace` (GH4743).
- provide automatic dtype conversions on `_reduce` operations (GH3371)
- exclude non-numeric if mixed types with datelike in `_reduce` operations (GH3371)

- default for `tupleize_cols` is now `False` for both `to_csv` and `read_csv`. Fair warning in 0.12 ([GH3604](#))
- moved `timedeltas` support to `pandas.tseries.timedeltas.py`; add `timedeltas` string parsing, add top-level `to_timedelta` function
- `NDFrame` now is compatible with Python's `toplevel abs()` function ([GH4821](#)).
- raise a `TypeError` on invalid comparison ops on `Series/DataFrame` (e.g. `integer/datetime`) ([GH4968](#))
- Added a new index type, `Float64Index`. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes `[], ix, loc` for scalar indexing and slicing work exactly the same. Indexing on other index types are preserved (and positional fallback for `[], ix`), with the exception, that floating point slicing on indexes on non `Float64Index` will raise a `TypeError`, e.g. `Series(range(5))[3.5:4.5]` ([GH263](#),:issue:5375)
- Make Categorical repr nicer ([GH4368](#))
- Remove deprecated `Factor` ([GH3650](#))
- Remove deprecated `set_printoptions/reset_printoptions` (:issue:3046)
- Remove deprecated `_verbose_info` ([GH3215](#))
- Begin removing methods that don't make sense on `GroupBy` objects ([GH4887](#)).
- Remove deprecated `read_clipboard/to_clipboard/ExcelFile/ExcelWriter` from `pandas.io.parsers` ([GH3717](#))
- All non-Index `NDFrames` (`Series`, `DataFrame`, `Panel`, `Panel4D`, `SparsePanel`, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (`add`, `sub`, `mul`, etc.). `SparsePanel` does not support `pow` or `mod` with non-scalars. ([GH3765](#))
- Arithmetic func factories are now passed real names (suitable for using with `super`) ([GH5240](#))
- Provide `numpy` compatibility with 1.7 for a calling convention like `np.prod(pandas_object)` as `numpy` call with additional keyword args ([GH4435](#))
- Provide `__dir__` method (and local context) for tab completion / remove `ipython` completers code ([GH4501](#))
- Support non-unique axes in a `Panel` via indexing operations ([GH4960](#))
- `.truncate` will raise a `ValueError` if invalid before and after dates are given ([GH5242](#))
- `Timestamp` now supports `now/today/utcnow` class methods ([GH5339](#))
- default for `display.max_seq_len` is now 100 rather than `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 `ipy2<=2.3.8`. an Issue has been opened against `ipy2` and a workaround is detailed in [GH5698](#). Thanks @JanSchulz.
- `Series.argmax` and `Series.argmin` are now aliased to `Series.idxmax` and `Series.idxmin`. 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](#))

### 28.1.5 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 `NotImplemented`
- `Series` now inherits from `NDFrame` rather than directly from `ndarray`. There are several minor changes that affect the API.

- numpy functions that do not support the array interface will now return `ndarrays` rather than series, e.g. `np.diff`, `np.ones_like`, `np.where`
- `Series(0.5)` would previously return the scalar `0.5`, this is no longer supported
- `TimeSeries` is now an alias for `Series`. the property `is_time_series` can be used to distinguish (if desired)
- Refactor of Sparse objects to use `BlockManager`
- Created a new block type in internals, `SparseBlock`, which can hold multi-dtypes and is non-consolidatable. `SparseSeries` and `SparseDataFrame` now inherit more methods from there hierarchy (`Series/DataFrame`), and no longer inherit from `SparseArray` (which instead is the object of the `SparseBlock`)
- Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
- Operations on sparse structures within `DataFrames` should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
- enable `setitem` on `SparseSeries` for boolean/integer/slices
- `SparsePanels` implementation is unchanged (e.g. not using `BlockManager`, needs work)
- added `ftypes` method to `Series/DataFrame`, similar to `dtypes`, but indicates if the underlying is sparse/dense (as well as the dtype)
- All `NDFrame` objects now have a `_prop_attributes`, which can be used to indicate various values to propagate to a new object from an existing (e.g. name in `Series` will follow more automatically now)
- Internal type checking is now done via a suite of generated classes, allowing `isinstance(value, klass)` without having to directly import the class, courtesy of @jtratner
- Bug in `Series` update where the parent frame is not updating its cache based on changes ([GH4080](#), [GH5216](#)) or types ([GH3217](#)), `fillna` ([GH3386](#))
- Indexing with dtype conversions fixed ([GH4463](#), [GH4204](#))
- Refactor `Series.reindex` to `core/generic.py` ([GH4604](#), [GH4618](#)), allow `method=` in reindexing on a `Series` to work
- `Series.copy` no longer accepts the `order` parameter and is now consistent with `NDFrame` `copy`
- Refactor `rename` methods to `core/generic.py`; fixes `Series.rename` for ([GH4605](#)), and adds `rename` with the same signature for `Panel`
- `Series` (for index) / `Panel` (for items) now as attribute access to its elements ([GH1903](#))
- Refactor `clip` methods to `core/generic.py` ([GH4798](#))
- Refactor of `_get_numeric_data/_get_bool_data` to `core/generic.py`, allowing `Series/Panel` functionality
- Refactor of `Series` arithmetic with time-like objects (`datetime/timedelta/time` etc.) into a separate, cleaned up wrapper class. ([GH4613](#))
- Complex compat for `Series` with `ndarray`. ([GH4819](#))
- Removed unnecessary `rwproperty` from codebase in favor of builtin property. ([GH4843](#))

- Refactor object level numeric methods (mean/sum/min/max...) from object level modules to `core/generic.py` (GH4435).
- Refactor cum objects to `core/generic.py` (GH4435), note that these have a more numpy-like function signature.
- `read_html()` now uses `TextParser` to parse HTML data from `bs4/lxml` (GH4770).
- Removed the `keep_internal` keyword parameter in `pandas/core/groupby.py` because it wasn't being used (GH5102).
- Base `DateOffsets` are no longer all instantiated on importing `pandas`, instead they are generated and cached on the fly. The internal representation and handling of `DateOffsets` has also been clarified. (GH5189, related GH5004)
- `MultiIndex` constructor now validates that passed levels and labels are compatible. (GH5213, GH5214)
- Unity `dropna` for `Series/DataFrame` signature (GH5250), tests from GH5234, courtesy of @rockg
- Rewrite `assert_almost_equal()` in cython for performance (GH4398)
- Added an internal `_update_inplace` method to facilitate updating `NDFrame` wrappers on in-place ops (only is for convenience of caller, doesn't actually prevent copies). (GH5247)

## 28.1.6 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 py3 (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 cparser with the `usecols` parameter (GH3192)
- Fix an issue in merging blocks where the resulting `DataFrame` had partially set `_ref_locs` (GH4403)
- Fixed an issue where hist subplots were being overwritten when they were called using the top level matplotlib API (GH4408)
- Fixed a bug where calling `Series.astype(str)` would truncate the string (GH4405, GH4437)
- Fixed a py3 compat issue where bytes were being repr'd as tuples (GH4455)
- Fixed Panel attribute naming conflict if item is named `'a'` (GH3440)
- Fixed an issue where duplicate indexes were raising when plotting (GH4486)
- Fixed an issue where `cumsum` and `cumprod` didn't work with bool dtypes (GH4170, GH4440)
- Fixed Panel slicing issued in `xs` that was returning an incorrect dimmed object (GH4016)
- Fix resampling bug where custom reduce function not used if only one group (GH3849, GH4494)
- Fixed Panel assignment with a transposed frame (GH3830)
- Raise on set indexing with a Panel and a Panel as a value which needs alignment (GH3777)
- `frozenset` objects now raise in the `Series` constructor (GH4482, GH4480)
- Fixed issue with sorting a duplicate multi-index that has multiple dtypes (GH4516)
- Fixed bug in `DataFrame.set_values` which was causing name attributes to be lost when expanding the index. (GH3742, GH4039)
- Fixed issue where individual names, levels and labels could be set on `MultiIndex` without validation (GH3714, GH4039)
- Fixed (GH3334) in `pivot_table`. Margins did not compute if values is the index.



- Fix bug in having a rhs of `np.timedelta64` or `np.offsets.DateOffset` when operating with date-times ([GH4532](#))
- Fix arithmetic with `series/datetimeindex` and `np.timedelta64` not working the same ([GH4134](#)) and buggy `timedelta` in `numpy 1.6` ([GH4135](#))
- Fix bug in `pd.read_clipboard` on windows with PY3 ([GH4561](#)); not decoding properly
- `tslib.get_period_field()` and `tslib.get_period_field_arr()` now raise if code argument out of range ([GH4519](#), [GH4520](#))
- Fix boolean indexing on an empty series loses index names ([GH4235](#)), `infer_dtype` works with empty arrays.
- Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue ([GH3317](#))
- Fixed issue where `DataFrame.apply` was reraising exceptions incorrectly (causing the original stack trace to be truncated).
- Fix selection with `ix/loc` and `non_unique` selectors ([GH4619](#))
- Fix assignment with `iloc/loc` involving a dtype change in an existing column ([GH4312](#), [GH5702](#)) have internal `setitem_with_indexer` in `core/indexing` to use `Block.setitem`
- Fixed bug where thousands operator was not handled correctly for floating point numbers in `csv_import` ([GH4322](#))
- Fix an issue with `CacheableOffset` not properly being used by many `DateOffset`; this prevented the `DateOffset` from being cached ([GH4609](#))
- Fix boolean comparison with a `DataFrame` on the lhs, and a list/tuple on the rhs ([GH4576](#))
- Fix error/dtype conversion with `setitem` of `None` on `Series/DataFrame` ([GH4667](#))
- Fix decoding based on a passed in non-default encoding in `pd.read_stata` ([GH4626](#))
- Fix `DataFrame.from_records` with a plain-vanilla `ndarray`. ([GH4727](#))
- Fix some inconsistencies with `Index.rename` and `MultiIndex.rename`, etc. ([GH4718](#), [GH4628](#))
- Bug in using `iloc/loc` with a cross-sectional and duplicate indicies ([GH4726](#))
- Bug with using `QUOTE_NONE` with `to_csv` causing `Exception`. ([GH4328](#))
- Bug with `Series` indexing not raising an error when the right-hand-side has an incorrect length ([GH2702](#))
- Bug in multi-indexing with a partial string selection as one part of a `MultiIndex` ([GH4758](#))
- Bug with reindexing on the index with a non-unique index will now raise `ValueError` ([GH4746](#))
- Bug in setting with `loc/ix` a single indexer with a multi-index axis and a `numpy` array, related to ([GH3777](#))
- Bug in concatenation with duplicate columns across dtypes not merging with `axis=0` ([GH4771](#), [GH4975](#))
- Bug in `iloc` with a slice index failing ([GH4771](#))
- Incorrect error message with no colspecs or width in `read_fwf`. ([GH4774](#))
- Fix bugs in indexing in a `Series` with a duplicate index ([GH4548](#), [GH4550](#))
- Fixed bug with reading compressed files with `read_fwf` in Python 3. ([GH3963](#))
- Fixed an issue with a duplicate index and assignment with a dtype change ([GH4686](#))
- Fixed bug with reading compressed files in as `bytes` rather than `str` in Python 3. Simplifies bytes-producing file-handling in Python 3 ([GH3963](#), [GH4785](#)).



- Fixed an issue related to ticklocs/ticklabels with log scale bar plots across different versions of matplotlib (GH4789)
- Suppressed DeprecationWarning associated with internal calls issued by repr() (GH4391)
- Fixed an issue with a duplicate index and duplicate selector with .loc (GH4825)
- Fixed an issue with DataFrame.sort\_index where, when sorting by a single column and passing a list for ascending, the argument for ascending was being interpreted as True (GH4839, GH4846)
- Fixed Panel.tshift not working. Added freq support to Panel.shift (GH4853)
- Fix an issue in TextFileReader w/ Python engine (i.e. PythonParser) with thousands != "," (GH4596)
- Bug in getitem with a duplicate index when using where (GH4879)
- Fix Type inference code coerces float column into datetime (GH4601)
- Fixed \_ensure\_numeric does not check for complex numbers (GH4902)
- Fixed a bug in Series.hist where two figures were being created when the by argument was passed (GH4112, GH4113).
- Fixed a bug in convert\_objects for > 2 ndims (GH4937)
- Fixed a bug in DataFrame/Panel cache insertion and subsequent indexing (GH4939, GH5424)
- Fixed string methods for FrozenNDArray and FrozenList (GH4929)
- Fixed a bug with setting invalid or out-of-range values in indexing enlargement scenarios (GH4940)
- Tests for fillna on empty Series (GH4346), thanks @immerrr
- Fixed copy() to shallow copy axes/indices as well and thereby keep separate metadata. (GH4202, GH4830)
- Fixed skiprows option in Python parser for read\_csv (GH4382)
- Fixed bug preventing cut from working with np.inf levels without explicitly passing labels (GH3415)
- Fixed wrong check for overlapping in DatetimeIndex.union (GH4564)
- Fixed conflict between thousands separator and date parser in csv\_parser (GH4678)
- Fix appending when dtypes are not the same (error showing mixing float/np.datetime64) (GH4993)
- Fix repr for DateOffset. No longer show duplicate entries in kwds. Removed unused offset fields. (GH4638)
- Fixed wrong index name during read\_csv if using usecols. Applies to c parser only. (GH4201)
- Timestamp objects can now appear in the left hand side of a comparison operation with a Series or DataFrame object (GH4982).
- Fix a bug when indexing with np.nan via iloc/loc (GH5016)
- Fixed a bug where low memory c parser could create different types in different chunks of the same file. Now coerces to numerical type or raises warning. (GH3866)
- Fix a bug where reshaping a Series to its own shape raised TypeError (GH4554) and other reshaping issues.
- Bug in setting with ix/loc and a mixed int/string index (GH4544)
- Make sure series-series boolean comparisons are label based (GH4947)
- Bug in multi-level indexing with a Timestamp partial indexer (GH4294)
- Tests/fix for multi-index construction of an all-nan frame (GH4078)
- Fixed a bug where read\_html() wasn't correctly inferring values of tables with commas (GH5029)

- Fixed a bug where `read_html()` wasn't providing a stable ordering of returned tables (GH4770, GH5029).
- Fixed a bug where `read_html()` was incorrectly parsing when passed `index_col=0` (GH5066).
- Fixed a bug where `read_html()` was incorrectly inferring the type of headers (GH5048).
- Fixed a bug where `DatetimeIndex` joins with `PeriodIndex` caused a stack overflow (GH3899).
- Fixed a bug where `groupby` objects didn't allow plots (GH5102).
- Fixed a bug where `groupby` objects weren't tab-completing column names (GH5102).
- Fixed a bug where `groupby.plot()` and `friends` were duplicating figures multiple times (GH5102).
- Provide automatic conversion of object dtypes on `fillna`, related (GH5103)
- Fixed a bug where default options were being overwritten in the option parser cleaning (GH5121).
- Treat a list/ndarray identically for `iloc` indexing with list-like (GH5006)
- Fix `MultiIndex.get_level_values()` with missing values (GH5074)
- Fix bound checking for `Timestamp()` with `datetime64` input (GH4065)
- Fix a bug where `TestReadHtml` wasn't calling the correct `read_html()` function (GH5150).
- Fix a bug with `NDFrame.replace()` which made replacement appear as though it was (incorrectly) using regular expressions (GH5143).
- Fix better error message for `to_datetime` (GH4928)
- Made sure different locales are tested on travis-ci (GH4918). Also adds a couple of utilities for getting locales and setting locales with a context manager.
- Fixed segfault on `isnull(MultiIndex)` (now raises an error instead) (GH5123, GH5125)
- Allow duplicate indices when performing operations that align (GH5185, GH5639)
- Compound dtypes in a constructor raise `NotImplementedError` (GH5191)
- Bug in comparing duplicate frames (GH4421) related
- Bug in `describe` on duplicate frames
- Bug in `to_datetime` with a format and `coerce=True` not raising (GH5195)
- Bug in `loc` setting with multiple indexers and a rhs of a Series that needs broadcasting (GH5206)
- Fixed bug where inplace setting of levels or labels on `MultiIndex` would not clear cached values property and therefore return wrong values. (GH5215)
- Fixed bug where filtering a grouped `DataFrame` or `Series` did not maintain the original ordering (GH4621).
- Fixed `Period` with a business date freq to always roll-forward if on a non-business date. (GH5203)
- Fixed bug in Excel writers where frames with duplicate column names weren't written correctly. (GH5235)
- Fixed issue with `drop` and a non-unique index on `Series` (GH5248)
- Fixed seg fault in C parser caused by passing more names than columns in the file. (GH5156)
- Fix `Series.isin` with date/time-like dtypes (GH5021)
- C and Python Parser can now handle the more common multi-index column format which doesn't have a row for index names (GH4702)
- Bug when trying to use an out-of-bounds date as an object dtype (GH5312)
- Bug when trying to display an embedded `PandasObject` (GH5324)

- Allows operating of Timestamps to return a datetime if the result is out-of-bounds related (GH5312)
- Fix return value/type signature of `initObjToJSON()` to be compatible with numpy's `import_array()` (GH5334, GH5326)
- Bug when renaming then `set_index` on a DataFrame (GH5344)
- Test suite no longer leaves around temporary files when testing graphics. (GH5347) (thanks for catching this @yarikoptic!)
- Fixed html tests on win32. (GH4580)
- Make sure that `head/tail` are `iloc` based, (GH5370)
- Fixed bug for `PeriodIndex` string representation if there are 1 or 2 elements. (GH5372)
- The `GroupBy` methods `transform` and `filter` can be used on Series and DataFrames that have repeated (non-unique) indices. (GH4620)
- Fix empty series not printing name in `repr` (GH4651)
- Make tests create temp files in temp directory by default. (GH5419)
- `pd.to_timedelta` of a scalar returns a scalar (GH5410)
- `pd.to_timedelta` accepts `NaN` and `NaT`, returning `NaT` instead of raising (GH5437)
- performance improvements in `isnull` on larger size pandas objects
- Fixed various setitem with 1d ndarray that does not have a matching length to the indexer (GH5508)
- Bug in getitem with a multi-index and `iloc` (GH5528)
- Bug in delitem on a Series (GH5542)
- Bug fix in `apply` when using custom function and objects are not mutated (GH5545)
- Bug in selecting from a non-unique index with `loc` (GH5553)
- Bug in groupby returning non-consistent types when user function returns a `None`, (GH5592)
- Work around regression in numpy 1.7.0 which erroneously raises `IndexError` from `ndarray.item` (GH5666)
- Bug in repeated indexing of object with resultant non-unique index (GH5678)
- Bug in `fillna` with Series and a passed series/dict (GH5703)
- Bug in groupby transform with a datetime-like grouper (GH5712)
- Bug in multi-index selection in PY3 when using certain keys (GH5725)
- Row-wise concat of differing dtypes failing in certain cases (GH5754)

## 28.2 pandas 0.12.0

Release date: 2013-07-24

### 28.2.1 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](#))

## 28.2.2 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)
- `DatetimeIndex`es no longer try to convert mixed-integer indexes during join operations (GH3877)
- Add `unit` keyword to `Timestamp` and `to_datetime` to enable passing of integers or floats that are in an epoch unit of `D`, `s`, `ms`, `us`, `ns`, thanks @mtkini (GH3969) (e.g. unix timestamps or epoch `s`, with fractional seconds allowed) (GH3540)
- `DataFrame` `corr` method (`spearman`) is now cythonized.
- Improved `network` test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. (GH3910, GH3914)
- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters
- Added `layout` keyword to `DataFrame.hist()` for more customizable layout (GH4050)
- `Timestamp.min` and `Timestamp.max` now represent valid `Timestamp` instances instead of the default `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)

### 28.2.3 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 than `numpy` threshold print options. (GH3426, GH3466)
- Added `mangle_dupe_cols` option to `read_table/csv`, allowing users to control legacy behaviour re dupe cols (`A`, `A.1`, `A.2` vs `A`, `A`) (GH3468) Note: The default value will change in 0.12 to the “no mangle” behaviour, If your code relies on this behaviour, explicitly specify `mangle_dupe_cols=True` in your calls.
- Do not allow `astypes` on `datetime64[ns]` except to `object`, and `timedelta64[ns]` to `object/int` (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when performed on a `Series` and return an *empty Series* when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of `slice` objects: - `sum`, `prod`, `mean`, `std`, `var`, `skew`, `kurt`, `corr`, and `cov`
- Do not allow `datetimelike/timedeltalike` creation except with valid types (e.g. cannot pass `datetime64[ms]`) (GH3423)

- Add `squeeze` keyword to `groupby` to allow reduction from `DataFrame` -> `Series` if groups are unique. Regression from 0.10.1, partial revert on (GH2893) with (GH3596)
- Raise on `iloc` when boolean indexing with a label based indexer mask e.g. a boolean `Series`, even with integer labels, will raise. Since `iloc` is purely positional based, the labels on the `Series` are not alignable (GH3631)
- The `raise_on_error` option to plotting methods is obviated by GH3572, so it is removed. Plots now always raise when data cannot be plotted or the object being plotted has a dtype of `object`.
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead (GH3582, GH3675, GH3676).
- the method and axis arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. (GH3907)
- Deprecated `display.height`, `display.width` is now only a formatting option does not control triggering of summary, similar to < 0.11.0.
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if `True`, default is `False` (same as prior to 0.12) (GH3679)
- io API changes
  - added `pandas.io.api` for i/o imports
  - removed Excel support to `pandas.io.excel`
  - added top-level `pd.read_sql` and `to_sql` `DataFrame` methods
  - removed clipboard support to `pandas.io.clipboard`
  - replace top-level and instance methods `save` and `load` with top-level `read_pickle` and `to_pickle` instance method, `save` and `load` will give deprecation warning.
- the method and axis arguments of `DataFrame.replace()` are deprecated
- set `FutureWarning` to require `data_source`, and to replace year/month with expiry date in `pandas.io` options. This is in preparation to add options 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)

## 28.2.4 Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. ([GH2301](#))

## 28.2.5 Bug Fixes

- Fixed an esoteric excel reading bug, `xlrd`  $\geq$  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 bud 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 similarly 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 were open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in `test_yahoo`, `test_google` where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where `sharex` and `sharey` were not being passed to `grouped_hist` (GH4089)
- Fix bug where HDFStore will fail to append because of a different block ordering on-disk (GH4096)
- Better error messages on inserting incompatible columns to a frame (GH4107)
- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when `regex=False` (GH4115)

- Fixed bug in `convert_objects(convert_numeric=True)` where a mixed numeric and object Series/Frame was not converting properly (GH4119)
- Fixed bugs in multi-index selection with column multi-index and duplicates (GH4145, GH4146)
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` (GH4152)
- Fixed bug in `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)

## 28.3 pandas 0.11.0

**Release date:** 2013-04-22

### 28.3.1 New features

- New documentation section, `10 Minutes to Pandas`
- New documentation section, `Cookbook`
- Allow mixed dtypes (e.g `float32/float64/int32/int16/int8`) to coexist in DataFrames and propagate in operations
- Add function to `pandas.io.data` for retrieving stock index components from Yahoo! finance (GH2795)
- Support slicing with time objects (GH2681)
- Added `.iloc` attribute, to support strict integer based indexing, analogous to `.ix` (GH2922)
- Added `.loc` attribute, to support strict label based indexing, analogous to `.ix` (GH3053)
- Added `.iat` attribute, to support fast scalar access via integers (replaces `iget_value/iset_value`)
- Added `.at` attribute, to support fast scalar access via labels (replaces `get_value/set_value`)
- Moved functionality from `irow, icol, iget_value/iset_value` to `.iloc` indexer (via `_ixs` methods in each object)
- Added support for expression evaluation using the `numexpr` library
- Added `convert=boolean` to take routines to translate negative indices to positive, defaults to `True`
- Added `to_series()` method to indices, to facilitate the creation of indexers (GH3275)

## 28.3.2 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
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D
```

```
In [3]: p.reindex(items=['ItemA']).squeeze()
```

```

 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 [4]: p.reindex(items=['ItemA'], minor=['B']).squeeze()
```

```
2001-01-02 -0.282863
```

```
2001-01-03 -0.173215
2001-01-04 -2.104569
2001-01-05 -0.706771
Freq: D, Name: B, dtype: float64
```

- Improvement to Yahoo API access in `pd.io.data.Options` (GH2758)
- added option `display.max_seq_items` to control the number of elements printed per sequence pprinting it. (GH2979)
- added option `display.chop_threshold` to control display of small numerical values. (GH2739)
- added option `display.max_info_rows` to prevent verbose\_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- `value_counts()` now accepts a “normalize” argument, for normalized histograms. (GH2710).
- `DataFrame.from_records` now accepts not only dicts but any instance of the `collections.Mapping` ABC.
- Allow selection semantics via a string with a datelike index to work in both Series and DataFrames (GH3070)

```
In [5]: idx = date_range("2001-10-1", periods=5, freq='M')
```

```
In [6]: ts = Series(np.random.rand(len(idx)), index=idx)
```

```
In [7]: ts['2001']
```

```
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']
```

```
 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/huynng/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)

### 28.3.3 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 (date-time/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.

## 28.3.4 Bug Fixes

- Fix seg fault on empty data frame when fillna with `pad` or `backfill` ([GH2778](#))
- Single element ndarrays of datetimelike objects are handled (e.g. `np.array(datetime(2001,1,1,0,0))`), w/o dtype being passed
- 0-dim ndarrays with a passed dtype are handled correctly (e.g. `np.array(0.,dtype='float32')`)
- Fix some boolean indexing inconsistencies in `Series.__getitem__/_setitem__` ([GH2776](#))
- Fix issues with DataFrame and Series constructor with integers that overflow `int64` and some mixed typed type lists ([GH2845](#))
- `HDFStore`
  - Fix weird PyTables error when using too many selectors in a where also correctly filter on any number of values in a Term expression (so not using `numexpr` filtering, but `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 indicies was producing incorrect return values (see [GH2922](#), [GH2892](#)), also check for out-of-bounds indices ([GH3029](#))
- Bug in DataFrame column insertion when the column creation fails, existing frame is left in an irrecoverable state ([GH3010](#))

- Bug in DataFrame update, combine\_first where non-specified values could cause dtype changes (GH3016, GH3041)
- Bug in groupby with first/last where dtypes could change (GH3041, GH2763)
- Formatting of an index that has nan was inconsistent or wrong (would fill from other values), (GH2850)
- Unstack of a frame with no nans would always cause dtype upcasting (GH2929)
- Fix scalar datetime.datetime parsing bug in read\_csv (GH3071)
- Fixed slow printing of large Dataframes, due to inefficient dtype reporting (GH2807)
- Fixed a segfault when using a function as grouper in groupby (GH3035)
- Fix pretty-printing of infinite data structures (closes GH2978)
- Fixed exception when plotting timeseries bearing a timezone (closes GH2877)
- str.contains ignored na argument (GH2806)
- Substitute warning for segfault when grouping with categorical grouper of mismatched length (GH3011)
- Fix exception in SparseSeries.density (GH2083)
- Fix upsampling bug with closed='left' and daily to daily data (GH3020)
- Fixed missing tick bars on scatter\_matrix plot (GH3063)
- Fixed bug in Timestamp(d,tz=foo) when d is date() rather than datetime() (GH2993)
- series.plot(kind='bar') now respects pylab color schem (GH3115)
- Fixed bug in reshape if not passed correct input, now raises TypeError (GH2719)
- Fixed a bug where Series ctor did not respect ordering if OrderedDict passed in (GH3282)
- Fix NameError issue on RESO\_US (GH2787)
- Allow selection in an *unordered* timeseries to work similiary to an *ordered* timeseries (GH2437).
- Fix implemented .xs when called with axes=1 and a level parameter (GH2903)
- Timestamp now supports the class method fromordinal similar to datetimes (GH3042)
- Fix issue with indexing a series with a boolean key and specifying a 1-len list on the rhs (GH2745) or a list on the rhs (GH3235)
- Fixed bug in groupby apply when kernel generate list of arrays having unequal len (GH1738)
- fixed handling of rolling\_corr with center=True which could produce corr>1 (GH3155)
- Fixed issues where indices can be passed as 'index/column' in addition to 0/1 for the axis parameter
- PeriodIndex.tolist now boxes to Period (GH3178)
- PeriodIndex.get\_loc KeyError now reports Period instead of ordinal (GH3179)
- df.to\_records bug when handling MultiIndex (GH3189)
- Fix Series.\_\_getitem\_\_ segfault when index less than -length (GH3168)
- Fix bug when using Timestamp as a date parser (GH2932)
- Fix bug creating date range from Timestamp with time zone and passing same time zone (GH2926)
- Add comparison operators to Period object (GH2781)
- Fix bug when concatenating two Series into a DataFrame when they have the same name (GH2797)
- Fix automatic color cycling when plotting consecutive timeseries without color arguments (GH2816)

- fixed bug in the pickling of PeriodIndex (GH2891)
- Upcast/split blocks when needed in a mixed DataFrame when setitem with an indexer (GH3216)
- Invoking df.applymap on a dataframe with dupe cols now raises a ValueError (GH2786)
- Apply with invalid returned indices raise correct Exception (GH2808)
- Fixed a bug in plotting log-scale bar plots (GH3247)
- df.plot() grid on/off now obeys the mpl default style, just like series.plot(). (GH3233)
- Fixed a bug in the legend of plotting.andrews\_curves() (GH3278)
- Produce a series on apply if we only generate a singular series and have a simple index (GH2893)
- Fix Python ascii file parsing when integer falls outside of floating point spacing (GH3258)
- fixed pretty printing of sets (GH3294)
- Panel() and Panel.from\_dict() now respects ordering when give OrderedDict (GH3303)
- DataFrame where with a datetimelike incorrectly selecting (GH3311)
- Ensure index casts work even in Int64Index
- Fix set\_index segfault when passing MultiIndex (GH3308)
- Ensure pickles created in py2 can be read in py3
- Insert ellipsis in MultiIndex summary repr (GH3348)
- Groupby will handle mutation among an input groups columns (and fallback to non-fast apply) (GH3380)
- Eliminated unicode errors on FreeBSD when using MPL GTK backend (GH3360)
- Period.strftime should return unicode strings always (GH3363)
- Respect passed read\_\* chunksize in get\_chunk function (GH3406)

## 28.4 pandas 0.10.1

**Release date:** 2013-01-22

### 28.4.1 New features

- Add data interface to World Bank WDI pandas.io.wb (GH2592)

### 28.4.2 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



### 28.4.3 Improvements to existing features

- `HDFStore`
  - enables storing of multi-index dataframes (closes [GH1277](#))
  - support data column indexing and selection, via `data_columns` keyword in `append`
  - support write chunking to reduce memory footprint, via `chunksize` keyword to `append`
  - support automagic indexing via `index` keyword to `append`
  - support `expectedrows` keyword in `append` to inform `PyTables` about the expected tablesize
  - support `start` and `stop` keywords in `select` to limit the row selection space
  - added `get_store` context manager to automatically import with `pandas`
  - added column filtering via `columns` keyword in `select`
  - added methods `append_to_multiple/select_as_multiple/select_as_coordinates` to do multiple-table `append`/`selection`
  - added support for `datetime64` in `columns`
  - added method `unique` to select the unique values in an indexable or data column
  - added method `copy` to copy an existing store (and possibly upgrade)
  - show the shape of the data on disk for non-table stores when printing the store
  - added ability to read `PyTables` flavor tables (allows compatibility to other `HDF5` systems)
- Add `logx` option to `DataFrame/Series.plot` ([GH2327](#), [GH2565](#))
- Support reading gzipped data from file-like object
- `pivot_table` `aggfunc` can be anything used in `GroupBy.aggregate` ([GH2643](#))
- Implement `DataFrame` merges in case where set cardinalities might overflow 64-bit integer ([GH2690](#))
- Raise exception in C file parser if integer dtype specified and have NA values. ([GH2631](#))
- Attempt to parse ISO8601 format dates when `parse_dates=True` in `read_csv` for major performance boost in such cases ([GH2698](#))
- Add methods `neg` and `inv` to `Series`
- Implement `kind` option in `ExcelFile` to indicate whether it's an XLS or XLSX file ([GH2613](#))

### 28.4.4 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)

## 28.5 pandas 0.10.0

**Release date:** 2012-12-17

### 28.5.1 New features

- Brand new high-performance delimited file parsing engine written in C and Cython. 50% or better performance in many standard use cases with a fraction as much memory usage. (GH407, GH821)
- Many new file parser (`read_csv`, `read_table`) features:
  - Support for on-the-fly gzip or bz2 decompression (*compression* option)
  - Ability to get back `numpy.recarray` instead of DataFrame (*as\_recarray=True*)
  - *dtype* option: explicit column dtypes

- *usecols* option: specify list of columns to be read from a file. Good for reading very wide files with many irrelevant columns ([GH1216](#) [GH926](#), [GH2465](#))
- Enhanced unicode decoding support via *encoding* option
- *skipinitialspace* dialect option
- Can specify strings to be recognized as True (*true\_values*) or False (*false\_values*)
- High-performance *delim\_whitespace* option for whitespace-delimited files; a preferred alternative to the 's+' regular expression delimiter
- Option to skip “bad” lines (wrong number of fields) that would otherwise have caused an error in the past (*error\_bad\_lines* and *warn\_bad\_lines* options)
- Substantially improved performance in the parsing of integers with thousands markers and lines with comments
- Easy of European (and other) decimal formats (*decimal* option) ([GH584](#), [GH2466](#))
- Custom line terminators (e.g. *lineterminator*='~') ([GH2457](#))
- Handling of no trailing commas in CSV files ([GH2333](#))
- Ability to handle fractional seconds in *date\_converters* ([GH2209](#))
- *read\_csv* allow scalar arg to *na\_values* ([GH1944](#))
- Explicit column dtype specification in *read\_\** functions ([GH1858](#))
- Easier CSV dialect specification ([GH1743](#))
- Improve parser performance when handling special characters ([GH1204](#))
- Google Analytics API integration with easy oauth2 workflow ([GH2283](#))
- Add error handling to *Series.str.encode/decode* ([GH2276](#))
- Add *where* and *mask* to *Series* ([GH2337](#))
- Grouped histogram via *by* keyword in *Series/DataFrame.hist* ([GH2186](#))
- Support optional *min\_periods* keyword in *corr* and *cov* for both *Series* and *DataFrame* ([GH2002](#))
- Add *duplicated* and *drop\_duplicates* functions to *Series* ([GH1923](#))
- Add docs for *HDFStore table* format
- ‘density’ property in *SparseSeries* ([GH2384](#))
- Add *ffill* and *bfill* convenience functions for forward- and backfilling time series data ([GH2284](#))
- New option configuration system and functions *set\_option*, *get\_option*, *describe\_option*, and *reset\_option*. Deprecate *set\_printoptions* and *reset\_printoptions* ([GH2393](#)). You can also access options as attributes via *pandas.options.X*
- Wide *DataFrames* can be viewed more easily in the console with new *expand\_frame\_repr* and *line\_width* configuration options. This is on by default now ([GH2436](#))
- Scikits.timeseries-like moving window functions via *rolling\_window* ([GH1270](#))

## 28.5.2 Experimental Features

- Add support for Panel4D, a named 4 Dimensional stucture
- Add support for *ndpanel* factory functions, to create custom, domain-specific N-Dimensional containers

### 28.5.3 API Changes

- The default binning/labeling behavior for `resample` has been changed to `closed='left'`, `label='left'` for daily and lower frequencies. This had been a large source of confusion for users. See “what’s new” page for more on this. (GH2410)
- Methods with `inplace` option now return `None` instead of the calling (modified) object (GH1893)
- The special case `DataFrame - TimeSeries` doing column-by-column broadcasting has been deprecated. Users should explicitly do e.g. `df.sub(ts, axis=0)` instead. This is a legacy hack and can lead to subtle bugs.
- `inf/-inf` are no longer considered as NA by `isnull/notnull`. To be clear, this is legacy cruft from early pandas. This behavior can be globally re-enabled using the new option `mode.use_inf_as_null` (GH2050, GH1919)
- `pandas.merge` will now default to `sort=False`. For many use cases sorting the join keys is not necessary, and doing it by default is wasteful
- Specify `header=0` explicitly to replace existing column names in file in `read_*` functions.
- Default column names for header-less parsed files (yielded by `read_csv`, etc.) are now the integers 0, 1, .... A new argument `prefix` has been added; to get the v0.9.x behavior specify `prefix='X'` (GH2034). This API change was made to make the default column names more consistent with the `DataFrame` constructor’s default column names when none are specified.
- `DataFrame` selection using a boolean frame now preserves input shape
- If function passed to `Series.apply` yields a `Series`, result will be a `DataFrame` (GH2316)
- Values like YES/NO/yes/no will not be considered as boolean by default any longer in the file parsers. This can be customized using the new `true_values` and `false_values` options (GH2360)
- `obj.fillna()` is no longer valid; make `method='pad'` no longer the default option, to be more explicit about what kind of filling to perform. Add `ffill/bfill` convenience functions per above (GH2284)
- `HDFStore.keys()` now returns an absolute path-name for each key
- `to_string()` now always returns a unicode string. (GH2224)
- File parsers will not handle NA sentinel values arising from passed converter functions

### 28.5.4 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](#))

### 28.5.5 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)

## 28.6 pandas 0.9.1

Release date: 2012-11-14

### 28.6.1 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)

### 28.6.2 API Changes

- Upsampling period index “spans” intervals. Example: annual periods upsampled to monthly will span all months in each year
- `Period.end_time` will yield timestamp at last nanosecond in the interval (GH2124, GH2125, GH1764)
- File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)



### 28.6.3 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](#))

### 28.6.4 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 'l' ([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)

## 28.7 pandas 0.9.0

**Release date:** 10/7/2012

### 28.7.1 New features

- Add `str.encode` and `str.decode` to Series (GH1706)
- Add `to_latex` method to DataFrame (GH1735)

- Add convenient expanding window equivalents of all rolling\_\* ops ([GH1785](#))
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance ([GH1748](#), [GH1739](#))
- Recognize and convert more boolean values in file parsing (Yes, No, TRUE, FALSE, variants thereof) ([GH1691](#), [GH1295](#))
- Add Panel.update method, analogous to DataFrame.update ([GH1999](#), [GH1988](#))

## 28.7.2 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](#))

## 28.7.3 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*

## 28.7.4 Bug Fixes

- Perform arithmetic column-by-column in mixed-type DataFrame to avoid type upcasting issues. Caused downstream DataFrame.diff bug (GH1896)
- Fix matplotlib auto-color assignment when no custom spectrum passed. Also respect passed color keyword argument (GH1711)
- Fix resampling logical error with closed='left' (GH1726)
- Fix critical DatetimeIndex.union bugs (GH1730, GH1719, GH1745, GH1702, GH1753)
- Fix critical DatetimeIndex.intersection bug with unanchored offsets (GH1708)
- Fix MM-YYYY time series indexing case (GH1672)
- Fix case where Categorical group key was not being passed into index in GroupBy result (GH1701)
- Handle Ellipsis in Series.\_\_getitem\_\_/\_setitem\_\_ (GH1721)
- Fix some bugs with handling datetime64 scalars of other units in NumPy 1.6 and 1.7 (GH1717)
- Fix performance issue in MultiIndex.format (GH1746)
- Fixed GroupBy bugs interacting with DatetimeIndex asof / map methods (GH1677)
- Handle factors with NAs in pandas.rpy (GH1615)
- Fix statsmodels import in pandas.stats.var (GH1734)
- Fix DataFrame repr/info summary with non-unique columns (GH1700)
- Fix Series.iget\_value for non-unique indexes (GH1694)
- Don't lose tzinfo when passing DatetimeIndex as DataFrame column (GH1682)
- Fix tz conversion with time zones that haven't had any DST transitions since first date in the array (GH1673)
- Fix field access with UTC->local conversion on unsorted arrays (GH1756)
- Fix isnull handling of array-like (list) inputs (GH1755)
- Fix regression in handling of Series in Series constructor (GH1671)
- Fix comparison of Int64Index with DatetimeIndex (GH1681)
- Fix min\_periods handling in new rolling\_max/min at array start (GH1695)
- Fix errors with how='median' and generic NumPy resampling in some cases caused by SeriesBinGrouper (GH1648, GH1688)
- When grouping by level, exclude unobserved levels (GH1697)
- Don't lose tzinfo in DatetimeIndex when shifting by different offset (GH1683)
- Hack to support storing data with a zero-length axis in HDFStore (GH1707)
- Fix DatetimeIndex tz-aware range generation issue (GH1674)

- Fix method='time' interpolation with intraday data ([GH1698](#))
- Don't plot all-NA DataFrame columns as zeros ([GH1696](#))
- Fix bug in scatter\_plot with by option ([GH1716](#))
- Fix performance problem in infer\_freq with lots of non-unique stamps ([GH1686](#))
- Fix handling of PeriodIndex as argument to create MultiIndex ([GH1705](#))
- Fix re: unicode MultiIndex level names in Series/DataFrame repr ([GH1736](#))
- Handle PeriodIndex in to\_datetime instance method ([GH1703](#))
- Support StaticTzInfo in DatetimeIndex infrastructure ([GH1692](#))
- Allow MultiIndex setops with length-0 other type indexes ([GH1727](#))
- Fix handling of DatetimeIndex in DataFrame.to\_records ([GH1720](#))
- Fix handling of general objects in isnull on which bool(...) fails ([GH1749](#))
- Fix .ix indexing with MultiIndex ambiguity ([GH1678](#))
- Fix .ix setting logic error with non-unique MultiIndex ([GH1750](#))
- Basic indexing now works on MultiIndex with > 1000000 elements, regression from earlier version of pandas ([GH1757](#))
- Handle non-float64 dtypes in fast DataFrame.corr/cov code paths ([GH1761](#))
- Fix DatetimeIndex.isin to function properly ([GH1763](#))
- Fix conversion of array of tz-aware datetime.datetime to DatetimeIndex with right time zone ([GH1777](#))
- Fix DST issues with generating anchored date ranges ([GH1778](#))
- Fix issue calling sort on result of Series.unique ([GH1807](#))
- Fix numerical issue leading to square root of negative number in rolling\_std ([GH1840](#))
- Let Series.str.split accept no arguments (like str.split) ([GH1859](#))
- Allow user to have dateutil 2.1 installed on a Python 2 system ([GH1851](#))
- Catch ImportError less aggressively in pandas/\_\_init\_\_.py ([GH1845](#))
- Fix pip source installation bug when installing from GitHub ([GH1805](#))
- Fix error when window size > array size in rolling\_apply ([GH1850](#))
- Fix pip source installation issues via SSH from GitHub
- Fix OLS.summary when column is a tuple ([GH1837](#))
- Fix bug in \_\_doc\_\_ patching when -OO passed to interpreter ([GH1792](#) [GH1741](#) [GH1774](#))
- Fix unicode console encoding issue in IPython notebook ([GH1782](#), [GH1768](#))
- Fix unicode formatting issue with Series.name ([GH1782](#))
- Fix bug in DataFrame.duplicated with datetime64 columns ([GH1833](#))
- Fix bug in Panel internals resulting in error when doing fillna after truncate not changing size of panel ([GH1823](#))
- Prevent segfault due to MultiIndex not being supported in HDFStore table format ([GH1848](#))
- Fix UnboundLocalError in Panel.\_\_setitem\_\_ and add better error ([GH1826](#))
- Fix to\_csv issues with list of string entries. Isnull works on list of strings now too ([GH1791](#))

- Fix Timestamp comparisons with datetime values outside the nanosecond range (1677-2262)
- Revert to prior behavior of `normalize_date` with `datetime.date` objects (return `datetime`)
- Fix broken interaction between `np.nansum` and `Series.any/all`
- Fix bug with multiple column date parsers ([GH1866](#))
- `DatetimeIndex.union(Int64Index)` was broken
- Make plot `x` vs `y` interface consistent with integer indexing ([GH1842](#))
- `set_index` inplace modified data even if unique check fails ([GH1831](#))
- Only use Q-OCT/NOV/DEC in quarterly frequency inference ([GH1789](#))
- Upcast to `dtype=object` when unstacking boolean `DataFrame` ([GH1820](#))
- Fix `float64/float32` merging bug ([GH1849](#))
- Fixes to `Period.start_time` for non-daily frequencies ([GH1857](#))
- Fix failure when converter used on `index_col` in `read_csv` ([GH1835](#))
- Implement `PeriodIndex.append` so that `pandas.concat` works correctly ([GH1815](#))
- Avoid Cython out-of-bounds access causing segfault sometimes in `pad_2d`, `backfill_2d`
- Fix resampling error with intraday times and anchored target time (like AS-DEC) ([GH1772](#))
- Fix `.ix` indexing bugs with mixed-integer indexes ([GH1799](#))
- Respect passed color keyword argument in `Series.plot` ([GH1890](#))
- Fix `rolling_min/max` when the window is larger than the size of the input array. Check other malformed inputs ([GH1899](#), [GH1897](#))
- Rolling variance / standard deviation with only a single observation in window ([GH1884](#))
- Fix unicode sheet name failure in `to_excel` ([GH1828](#))
- Override `DatetimeIndex.min/max` to return `Timestamp` objects ([GH1895](#))
- Fix column name formatting issue in length-truncated column ([GH1906](#))
- Fix broken handling of copying `Index` metadata to new instances created by `view(...)` calls inside the NumPy infrastructure
- Support `datetime.date` again in `DateOffset.rollback/rollforward`
- Raise `Exception` if `set` passed to `Series` constructor ([GH1913](#))
- Add `TypeError` when appending `HDFStore` table w/ wrong index type ([GH1881](#))
- Don't raise exception on empty inputs in EW functions (e.g. `ewma`) ([GH1900](#))
- Make `asof` work correctly with `PeriodIndex` ([GH1883](#))
- Fix extlinks in doc build
- Fill boolean `DataFrame` with `NaN` when calling `shift` ([GH1814](#))
- Fix `setuptools` bug causing `pip` not to Cythonize `.pyx` files sometimes
- Fix negative integer indexing regression in `.ix` from 0.7.x ([GH1888](#))
- Fix error while retrieving timezone and utc offset from subclasses of `datetime.tzinfo` without `.zone` and `._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](#))

## 28.8 pandas 0.8.1

**Release date:** July 22, 2012

### 28.8.1 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](#))

### 28.8.2 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](#))

### 28.8.3 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)



## 28.9 pandas 0.8.0

Release date: 6/29/2012

### 28.9.1 New features

- New unified DatetimeIndex class for nanosecond-level timestamp data
- New Timestamp datetime.datetime subclass with easy time zone conversions, and support for nanoseconds
- New PeriodIndex class for timespans, calendar logic, and Period scalar object
- High performance resampling of timestamp and period data. New *resample* method of all pandas data structures
- New frequency names plus shortcut string aliases like '15h', '1h30min'
- Time series string indexing shorthand ([GH222](#))
- Add week, dayofyear array and other timestamp array-valued field accessor functions to DatetimeIndex
- Add GroupBy.prod optimized aggregation function and 'prod' fast time series conversion method ([GH1018](#))
- Implement robust frequency inference function and *inferred\_freq* attribute on DatetimeIndex ([GH391](#))
- New *tz\_convert* and *tz\_localize* methods in Series / DataFrame
- Convert DatetimeIndexes to UTC if time zones are different in join/setops ([GH864](#))
- Add limit argument for forward/backward filling to reindex, fillna, etc. ([GH825](#) and others)
- Add support for indexes (dates or otherwise) with duplicates and common sense indexing/selection functionality
- Series/DataFrame.update methods, in-place variant of combine\_first ([GH961](#))
- Add *match* function to API ([GH502](#))
- Add Cython-optimized first, last, min, max, prod functions to GroupBy ([GH994](#), [GH1043](#))
- Dates can be split across multiple columns ([GH1227](#), [GH1186](#))
- Add experimental support for converting pandas DataFrame to R data.frame via rpy2 ([GH350](#), [GH1212](#))
- Can pass list of (name, function) to GroupBy.aggregate to get aggregates in a particular order ([GH610](#))
- Can pass dicts with lists of functions or dicts to GroupBy aggregate to do much more flexible multiple function aggregation ([GH642](#), [GH610](#))
- New ordered\_merge functions for merging DataFrames with ordered data. Also supports group-wise merging for panel data ([GH813](#))
- Add keys() method to DataFrame
- Add flexible replace method for replacing potentially values to Series and DataFrame ([GH929](#), [GH1241](#))
- Add 'kde' plot kind for Series/DataFrame.plot ([GH1059](#))
- More flexible multiple function aggregation with GroupBy
- Add pct\_change function to Series/DataFrame
- Add option to interpolate by Index values in Series.interpolate ([GH1206](#))
- Add max\_colwidth option for DataFrame, defaulting to 50
- Conversion of DataFrame through rpy2 to R data.frame ([GH1282](#), )
- Add keys() method on DataFrame ([GH1240](#))

- Add new `match` function to API (similar to R) (GH502)
- Add `dayfirst` option to parsers (GH854)
- Add `method` argument to `align` method for forward/backward fillin (GH216)
- Add `Panel.transpose` method for rearranging axes (GH695)
- Add new `cut` function (patterned after R) for discretizing data into equal range-length bins or arbitrary breaks of your choosing (GH415)
- Add new `qcut` for cutting with quantiles (GH1378)
- Add `value_counts` top level array method (GH1392)
- Added Andrews curves plot tupe (GH1325)
- Add lag plot (GH1440)
- Add `autocorrelation_plot` (GH1425)
- Add support for tox and Travis CI (GH1382)
- Add support for Categorical use in GroupBy (GH292)
- Add `any` and `all` methods to DataFrame (GH1416)
- Add `secondary_y` option to Series.plot
- Add experimental `reshape` function for reshaping wide to long

## 28.9.2 Improvements to existing features

- Switch to klib/khash-based hash tables in Index classes for better performance in many cases and lower memory footprint
- Shipping some functions from `scipy.stats` to reduce dependency, e.g. `Series.describe` and `DataFrame.describe` (GH1092)
- Can create MultiIndex by passing list of lists or list of arrays to Series, DataFrame constructor, etc. (GH831)
- Can pass arrays in addition to column names to `DataFrame.set_index` (GH402)
- Improve the speed of “square” reindexing of homogeneous DataFrame objects by significant margin (GH836)
- Handle more dtypes when passed MaskedArrays in DataFrame constructor (GH406)
- Improved performance of join operations on integer keys (GH682)
- Can pass multiple columns to GroupBy object, e.g. `grouped[[col1, col2]]` to only aggregate a subset of the value columns (GH383)
- Add histogram / kde plot options for `scatter_matrix` diagonals (GH1237)
- Add `inplace` option to `Series/DataFrame.rename` and `sort_index`, `DataFrame.drop_duplicates` (GH805, GH207)
- More helpful error message when nothing passed to `Series.reindex` (GH1267)
- Can mix array and scalars as dict-value inputs to DataFrame ctor (GH1329)
- Use DataFrame columns’ name for legend title in plots
- Preserve frequency in DatetimeIndex when possible in boolean indexing operations
- Promote `datetime.date` values in data alignment operations (GH867)
- Add `order` method to Index classes (GH1028)

- Avoid hash table creation in large monotonic hash table indexes (GH1160)
- Store time zones in HDFStore (GH1232)
- Enable storage of sparse data structures in HDFStore (GH85)
- Enable Series.asof to work with arrays of timestamp inputs
- Cython implementation of DataFrame.corr speeds up by > 100x (GH1349, GH1354)
- Exclude “nuisance” columns automatically in GroupBy.transform (GH1364)
- Support functions-as-strings in GroupBy.transform (GH1362)
- Use index name as xlabel/ylabel in plots (GH1415)
- Add `convert_dtype` option to Series.apply to be able to leave data as dtype=object (GH1414)
- Can specify all index level names in concat (GH1419)
- Add `dialect` keyword to parsers for quoting conventions (GH1363)
- Enable DataFrame[bool\_DataFrame] += value (GH1366)
- Add `retries` argument to `get_data_yahoo` to try to prevent Yahoo! API 404s (GH826)
- Improve performance of reshaping by using O(N) categorical sorting
- Series names will be used for index of DataFrame if no index passed (GH1494)
- Header argument in DataFrame.to\_csv can accept a list of column names to use instead of the object’s columns (GH921)
- Add `raise_conflict` argument to DataFrame.update (GH1526)
- Support file-like objects in ExcelFile (GH1529)

### 28.9.3 API Changes

- Rename `pandas._tseries` 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)

## 28.9.4 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)

## 28.10 pandas 0.7.3

**Release date:** April 12, 2012

### 28.10.1 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](#))

## 28.10.2 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](#))

## 28.10.3 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 grouby results ([GH995](#))
- DataFrame.from\_records no longer mutate input columns ([GH975](#))
- Use Index name when grouping by it ([GH1313](#))

## 28.11 pandas 0.7.2

**Release date:** March 16, 2012

### 28.11.1 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

### 28.11.2 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)

### 28.11.3 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)

### 28.11.4 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-existent 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](#))

## 28.12 pandas 0.7.1

**Release date:** February 29, 2012

### 28.12.1 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

### 28.12.2 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](#))

### 28.12.3 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](#))

## 28.13 pandas 0.7.0

Release date: 2/9/2012

### 28.13.1 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

### 28.13.2 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 GH395)
- The default delimiter for `read_csv` is comma rather than letting `csv.Sniffer` infer it
- Rename `col_or_columns` argument in `DataFrame.drop_duplicates` (GH GH734)

### 28.13.3 Improvements to existing features

- Better error message in DataFrame constructor when passed column labels don't match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don't print length by default in `Series.to_string`, add `length` option (GH GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of `Series.__getitem__` for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in `setup.py` if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also (GH536)
- Default name assignment when calling `reset_index` on DataFrame with a regular (non-hierarchical) index (GH476)
- Use 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(icol)` and `Series.iget`, (GH [GH654](#))
- Prototypes for some vectorized string functions
- Add float64 hash table to solve the `Series.unique` problem with NAs ([GH714](#))
- Memoize objects when reading from file to reduce memory footprint
- Can get and set a column of a DataFrame with hierarchical columns containing “empty” (“”) lower levels without passing the empty levels (PR [GH768](#))

### 28.13.4 Bug Fixes

- Raise exception in out-of-bounds indexing of `Series` instead of seg-faulting, regression from earlier releases ([GH495](#))
- Fix error when joining DataFrames of different dtypes within the same typeclass (e.g. `float32` and `float64`) ([GH486](#))
- Fix bug in `Series.min`/`Series.max` on objects like `datetime.datetime` (GH [GH487](#))
- Preserve index names in `Index.union` ([GH501](#))
- Fix bug in `Index` joining causing subclass information (like `DateRange` type) to be lost in some cases ([GH500](#))
- Accept empty list as input to `DataFrame` constructor, regression from 0.6.0 ([GH491](#))
- Can output `DataFrame` and `Series` with ndarray objects in a `dtype=object` array ([GH490](#))
- Return empty string from `Series.to_string` when called on empty `Series` (GH [GH488](#))
- Fix exception passing empty list to `DataFrame.from_records`
- Fix `Index.format` bug (excluding name field) with datetimes with time info
- Fix scalar value access in `Series` to always return NumPy scalars, regression from prior versions ([GH510](#))
- Handle rows skipped at beginning of file in `read_*` functions ([GH505](#))
- Handle improper dtype casting in `set_value` methods
- Unary `'-'` / `__neg__` operator on `DataFrame` was returning integer values

- Unbox 0-dim ndarrays from certain operators like all, any in Series
- Fix handling of missing columns (was combine\_first-specific) in DataFrame.combine for general case (GH529)
- Fix type inference logic with boolean lists and arrays in DataFrame indexing
- Use centered sum of squares in R-square computation if entity\_effects=True in panel regression
- Handle all NA case in Series.{corr, cov}, was raising exception (GH548)
- Aggregating by multiple levels with level argument to DataFrame, Series stat method, was broken (GH545)
- Fix Cython buf when converter passed to read\_csv produced a numeric array (buffer dtype mismatch when passed to Cython type inference function) (GH GH546)
- Fix exception when setting scalar value using .ix on a DataFrame with a MultiIndex (GH551)
- Fix outer join between two DateRanges with different offsets that returned an invalid DateRange
- Cleanup DataFrame.from\_records failure where index argument is an integer
- Fix Data.from\_records failure when passed a dictionary
- Fix NA handling in {Series, DataFrame}.rank with non-floating point dtypes
- Fix bug related to integer type-checking in .ix-based indexing
- Handle non-string index name passed to DataFrame.from\_records
- DataFrame.insert caused the columns name(s) field to be discarded (GH527)
- Fix erroneous in monotonic many-to-one left joins
- Fix DataFrame.to\_string to remove extra column white space (GH571)
- Format floats to default to same number of digits (GH395)
- Added decorator to copy docstring from one function to another (GH449)
- Fix error in monotonic many-to-one left joins
- Fix \_\_eq\_\_ comparison between DateOffsets with different relativedelta keywords passed
- Fix exception caused by parser converter returning strings (GH583)
- Fix MultiIndex formatting bug with integer names (GH601)
- Fix bug in handling of non-numeric aggregates in Series.groupby (GH612)
- Fix TypeError with tuple subclasses (e.g. namedtuple) in DataFrame.from\_records (GH611)
- Catch misreported console size when running IPython within Emacs
- Fix minor bug in pivot table margins, loss of index names and length-1 'All' tuple in row labels
- Add support for legacy WidePanel objects to be read from HDFStore
- Fix out-of-bounds segfault in pad\_object and backfill\_object methods when either source or target array are empty
- Could not create a new column in a DataFrame from a list of tuples
- Fix bugs preventing SparseDataFrame and SparseSeries working with groupby (GH666)
- Use sort kind in Series.sort / argsort (GH668)
- Fix DataFrame operations on non-scalar, non-pandas objects (GH672)
- Don't convert DataFrame column to integer type when passing integer to \_\_setitem\_\_ (GH669)
- Fix downstream bug in pivot\_table caused by integer level names in MultiIndex (GH678)

- Fix `SparseSeries.combine_first` when passed a dense Series ([GH687](#))
- Fix performance regression in `HDFStore` loading when `DataFrame` or `Panel` stored in table format with datetimes
- Raise `Exception` in `DateRange` when offset with `n=0` is passed ([GH683](#))
- Fix `get/set` inconsistency with `.ix` property and integer location but non-integer index ([GH707](#))
- Use right `dropna` function for `SparseSeries`. Return dense Series for NA fill value ([GH730](#))
- Fix `Index.format` bug causing incorrectly string-formatted Series with datetime indexes ([GH726](#), [GH758](#))
- Fix errors caused by object dtype arrays passed to `ols` ([GH759](#))
- Fix error where column names lost when passing list of labels to `DataFrame.__getitem__`, ([GH662](#))
- Fix error whereby top-level week iterator overwrote week instance
- Fix circular reference causing memory leak in sparse array / series / frame, ([GH663](#))
- Fix integer-slicing from integers-as-floats ([GH670](#))
- Fix zero division errors in `nanops` from object dtype arrays in all NA case ([GH676](#))
- Fix csv encoding when using unicode ([GH705](#), [GH717](#), [GH738](#))
- Fix assumption that each object contains every unique block type in `concat`, ([GH708](#))
- Fix sortedness check of multiindex in `to_panel` ([GH719](#), 720)
- Fix that `None` was not treated as NA in `PyObjectHashtable`
- Fix hashing dtype because of endianness confusion ([GH747](#), [GH748](#))
- Fix `SparseSeries.dropna` to return dense Series in case of NA fill value ([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 Cobzarencu
- 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

## 28.14 pandas 0.6.1

**Release date:** 12/13/2011

### 28.14.1 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](#))

### 28.14.2 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)

### 28.14.3 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

### 28.14.4 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 ([GH481](#))

#### Thanks

- Ralph Bean
- Luca Beltrame
- Marius Cobzarencu
- 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

## 28.15 pandas 0.6.0

Release date: 11/25/2011

### 28.15.1 API Changes

- Arithmetic methods like *sum* will attempt to sum dtype=object values by default instead of excluding them (GH382)

### 28.15.2 New features

- Add *melt* function to *pandas.core.reshape*
- Add *level* parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Add *head* and *tail* methods to Series, analogous to to DataFrame (PR GH296)
- Add *Series.isin* function which checks if each value is contained in a passed sequence (GH289)
- Add *float\_format* option to *Series.to\_string*
- Add *skip\_footer* (GH291) and *converters* (GH343) options to *read\_csv* and *read\_table*
- Add proper, tested weighted least squares to standard and panel OLS (GH GH303)
- Add *drop\_duplicates* and *duplicated* functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- Implement logical (boolean) operators *&*, *|*, *^* on DataFrame (GH347)
- Add *Series.mad*, mean absolute deviation, matching DataFrame
- Add *QuarterEnd* DateOffset (GH321)
- Add matrix multiplication function *dot* to DataFrame (GH65)
- Add *orient* option to *Panel.from\_dict* to ease creation of mixed-type Panels (GH359, GH301)
- Add *DataFrame.from\_dict* with similar *orient* option
- Can now pass list of tuples or list of lists to *DataFrame.from\_records* for fast conversion to DataFrame (GH357)
- Can pass multiple levels to groupby, e.g. *df.groupby(level=[0, 1])* (GH GH103)
- Can sort by multiple columns in *DataFrame.sort\_index* (GH92, GH362)
- Add fast *get\_value* and *put\_value* methods to DataFrame and micro-performance tweaks (GH360)
- Add *cov* instance methods to Series and DataFrame (GH194, GH362)
- Add bar plot option to *DataFrame.plot* (GH348)
- Add *idxmin* and *idxmax* functions to Series and DataFrame for computing index labels achieving maximum and minimum values (GH286)
- Add *read\_clipboard* function for parsing DataFrame from OS clipboard, should work across platforms (GH300)
- Add *nunique* function to Series for counting unique elements (GH297)
- DataFrame constructor will use Series name if no columns passed (GH373)
- Support regular expressions and longer delimiters in *read\_table/read\_csv*, but does not handle quoted strings yet (GH364)

- Add *DataFrame.to\_html* for formatting DataFrame to HTML (GH387)
- MaskedArray can be passed to DataFrame constructor and masked values will be converted to NaN (GH396)
- Add *DataFrame.boxplot* function (GH368, others)
- Can pass extra args, kwds to DataFrame.apply (GH376)

### 28.15.3 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

### 28.15.4 Bug Fixes

- Fix bug in *DataFrame.to\_csv* when writing a DataFrame with an index name (GH290)
- DataFrame should clear its Series caches on consolidation, was causing “stale” Series to be returned in some corner cases (GH304)

- DataFrame constructor failed if a column had a list of tuples (GH293)
- Ensure that *Series.apply* always returns a Series and implement *Series.round* (GH314)
- Support boolean columns in Cythonized groupby functions (GH315)
- *DataFrame.describe* should not fail if there are no numeric columns, instead return categorical describe (GH323)
- Fixed bug which could cause columns to be printed in wrong order in *DataFrame.to\_string* if specific list of columns passed (GH325)
- Fix legend plotting failure if DataFrame columns are integers (GH326)
- Shift start date back by one month for Yahoo! Finance API in *pandas.io.data* (GH329)
- Fix *DataFrame.join* failure on unconsolidated inputs (GH331)
- *DataFrame.min/max* will no longer fail on mixed-type DataFrame (GH337)
- Fix *read\_csv / read\_table* failure when passing list to *index\_col* that is not in ascending order (GH349)
- Fix failure passing *Int64Index* to *Index.union* when both are monotonic
- Fix error when passing *SparseSeries* to (dense) DataFrame constructor
- Added missing bang at top of *setup.py* (GH352)
- Change *is\_monotonic* on *MultiIndex* so it properly compares the tuples
- Fix *MultiIndex* outer join logic (GH351)
- Set index name attribute with single-key groupby (GH358)
- Bug fix in reflexive binary addition in Series and DataFrame for non-commutative operations (like string concatenation) (GH353)
- *setuptools.py* will invoke Cython (GH192)
- Fix block consolidation bug after inserting column into *MultiIndex* (GH366)
- Fix bug in join operations between *Index* and *Int64Index* (GH367)
- Handle *min\_periods=0* case in moving window functions (GH365)
- Fixed corner cases in *DataFrame.apply/pivot* with empty DataFrame (GH378)
- Fixed repr exception when Series name is a tuple
- Always return *DateRange* from *asfreq* (GH390)
- Pass level names to *swaplevel* (GH379)
- Don't lose index names in *MultiIndex.droplevel* (GH394)
- Infer more proper return type in *DataFrame.apply* when no columns or rows depending on whether the passed function is a reduction (GH389)
- Always return NA/NaN from *Series.min/max* and *DataFrame.min/max* when all of a row/column/values are NA (GH384)
- Enable partial setting with *.ix / advanced indexing* (GH397)
- Handle mixed-type DataFrames correctly in *unstack*, do not lose type information (GH403)
- Fix integer name formatting bug in *Index.format* and in *Series.\_\_repr\_\_*
- Handle label types other than string passed to groupby (GH405)
- Fix bug in *.ix*-based indexing with partial retrieval when a label is not contained in a level

- Index name was not being pickled ([GH408](#))
- Level name should be passed to result index in GroupBy.apply ([GH416](#))

### Thanks

- Craig Austin
- Marius Cobzarencu
- 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

## 28.16 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.

### 28.16.1 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

## 28.16.2 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

## 28.16.3 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)

## 28.16.4 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`
- 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

## 28.17 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!).

### 28.17.1 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](#))

### 28.17.2 Improvements to existing features

- Skip xlrld-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

### 28.17.3 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*

### 28.17.4 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

## 28.18 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

### 28.18.1 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

### 28.18.2 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

### 28.18.3 API Changes

None

### 28.18.4 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

## 28.19 pandas 0.4.1

**Release date:** 9/25/2011

This is primarily a bug fix release but includes some new features and improvements

### 28.19.1 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*

### 28.19.2 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

### 28.19.3 API Changes

None

### 28.19.4 Bug Fixes

- Fixed *DataFrame* constructor bug causing downstream problems (e.g. *.copy()* failing) when passing a *Series* as the values along with a column name and index
- Fixed single-key groupby on *DataFrame* with *as\_index=False* ([GH160](#))
- *Series.shift* was failing on integer *Series* ([GH154](#))
- *unstack* methods were producing incorrect output in the case of duplicate hierarchical labels. An exception will now be raised ([GH147](#))
- Calling *count* with level argument caused reduceat failure or segfault in earlier NumPy ([GH169](#))
- Fixed *DataFrame.corrwith* to automatically exclude non-numeric data ([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

## 28.20 pandas 0.4.0

Release date: 9/12/2011

### 28.20.1 New features

- *pandas.core.sparse* module: “Sparse” (mostly-NA, or some other fill value) versions of *Series*, *DataFrame*, and *Panel*. For low-density data, this will result in significant performance boosts, and smaller memory footprint. Added *to\_sparse* methods to *Series*, *DataFrame*, and *Panel*. See online documentation for more on these
- Fancy indexing operator on *Series* / *DataFrame*, e.g. via *.ix* operator. Both getting and setting of values is supported; however, setting values will only currently work on homogeneously-typed *DataFrame* objects. Things like:
  - `series.ix[[d1, d2, d3]]`
  - `frame.ix[5:10, ['C', 'B', 'A']], frame.ix[5:10, 'A':'C']`
  - `frame.ix[date1:date2]`
- Significantly enhanced *groupby* functionality
  - Can groupby multiple keys, e.g. `df.groupby(['key1', 'key2'])`. Iteration with multiple groupings products a flattened tuple
  - “Nuisance” columns (non-aggregatable) will automatically be excluded from *DataFrame* aggregation operations
  - Added automatic “dispatching to *Series* / *DataFrame* methods to more easily invoke methods on groups. e.g. `s.groupby(crit).std()` will work even though *std* is not implemented on the *GroupBy* class
- Hierarchical / multi-level indexing
  - New the *MultiIndex* class. Integrated *MultiIndex* into *Series* and *DataFrame* fancy indexing, slicing, `__getitem__` and `__setitem__`, reindexing, etc. Added *level* keyword argument to *groupby* to enable grouping by a level of a *MultiIndex*
- New data reshaping functions: *stack* and *unstack* on *DataFrame* and *Series*
  - Integrate with *MultiIndex* to enable sophisticated reshaping of data
- *Index* objects (labels for axes) are now capable of holding tuples
- *Series.describe*, *DataFrame.describe*: produces an R-like table of summary statistics about each data column
- *DataFrame.quantile*, *Series.quantile* for computing sample quantiles of data across requested axis

- Added general *DataFrame.dropna* method to replace *dropIncompleteRows* and *dropEmptyRows*, deprecated those.
- *Series* arithmetic methods with optional *fill\_value* for missing data, e.g. *a.add(b, fill\_value=0)*. If a location is missing for both it will still be missing in the result though.
- *fill\_value* option has been added to *DataFrame*.{*add*, *mul*, *sub*, *div*} methods similar to *Series*
- Boolean indexing with *DataFrame* objects: *data[data > 0.1] = 0.1* or *data[data > other] = 1*.
- *pytz* / *tzinfo* support in *DateRange*
  - *tz\_localize*, *tz\_normalize*, and *tz\_validate* methods added
- Added *ExcelFile* class to *pandas.io.parsers* for parsing multiple sheets out of a single Excel 2003 document
- *GroupBy* aggregations can now optionally *broadcast*, e.g. produce an object of the same size with the aggregated value propagated
- Added *select* function in all data structures: *reindex* axis based on arbitrary criterion (function returning boolean value), e.g. *frame.select(lambda x: 'foo' in x, axis=1)*
- *DataFrame consolidate* method, API function relating to redesigned internals
- *DataFrame.insert* method for inserting column at a specified location rather than the default *\_\_setitem\_\_* behavior (which puts it at the end)
- *HDFStore* class in *pandas.io.pytables* has been largely rewritten using patches from Jeff Reback from others. It now supports mixed-type *DataFrame* and *Series* data and can store *Panel* objects. It also has the option to query *DataFrame* and *Panel* data. Loading data from legacy *HDFStore* files is supported explicitly in the code
- Added *set\_printoptions* method to modify appearance of *DataFrame* tabular output
- *rolling\_quantile* functions; a moving version of *Series.quantile* / *DataFrame.quantile*
- Generic *rolling\_apply* moving window function
- New *drop* method added to *Series*, *DataFrame*, etc. which can drop a set of labels from an axis, producing a new object
- *reindex* methods now sport a *copy* option so that data is not forced to be copied then the resulting object is indexed the same
- Added *sort\_index* methods to *Series* and *Panel*. Renamed *DataFrame.sort* to *sort\_index*. Leaving *DataFrame.sort* for now.
- Added *skipna* option to statistical instance methods on all the data structures
- *pandas.io.data* module providing a consistent interface for reading time series data from several different sources

## 28.20.2 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 be 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).

### 28.20.3 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*, *tgrouby*, *toString*
- *pandas.io.parsers* method deprecations
  - *parseCSV* is now *read\_csv* and keyword arguments have been de-camelCased
  - *parseText* is now *read\_table*
  - *parseExcel* is replaced by the *ExcelFile* class and its *parse* method
- *fillMethod* arguments (deprecated in prior release) removed, should be replaced with *method*
- *Series.fill*, *DataFrame.fill*, and *Panel.fill* removed, use *fillna* instead
- *groupby* functions now exclude NA / NaN values from the list of groups. This matches R behavior with NAs in factors e.g. with the *tapply* function
- Removed *parseText*, *parseCSV* and *parseExcel* from pandas namespace
- *Series.combineFunc* renamed to *Series.combine* and made a bit more general with a *fill\_value* keyword argument defaulting to NaN
- Removed *pandas.core.pytools* module. Code has been moved to *pandas.core.common*
- Tacked on *groupName* attribute for groups in GroupBy renamed to *name*
- Panel/LongPanel *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

## 28.20.4 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
- Chris Uga
- Chris Withers
- Jeff Reback
- Ted Square
- Craig Austin
- William Ferreira
- Daniel Fortunov
- Tony Roberts
- Martin Felder
- John Marino
- Tim McNamara
- Justin Berka
- Dieter Vandenbussche
- Shane Conway
- Skipper Seabold
- Chris Jordan-Squire

## 28.21 pandas 0.3.0

Release date: February 20, 2011

### 28.21.1 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

### 28.21.2 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.)

### 28.21.3 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

## 28.21.4 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



# PYTHON MODULE INDEX

p

pandas, [1](#)