# pandas: powerful Python data analysis toolkit

*Release 0.7.3*

Wes McKinney

November 08, 2012

# CONTENTS

PDF Version **Date**: November 08, 2012 **Version**: 0.7.3

**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 will soon become a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

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

See the package overview for more detail about what's in the library.

CHAPTER

ONE

# WHAT'S NEW

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

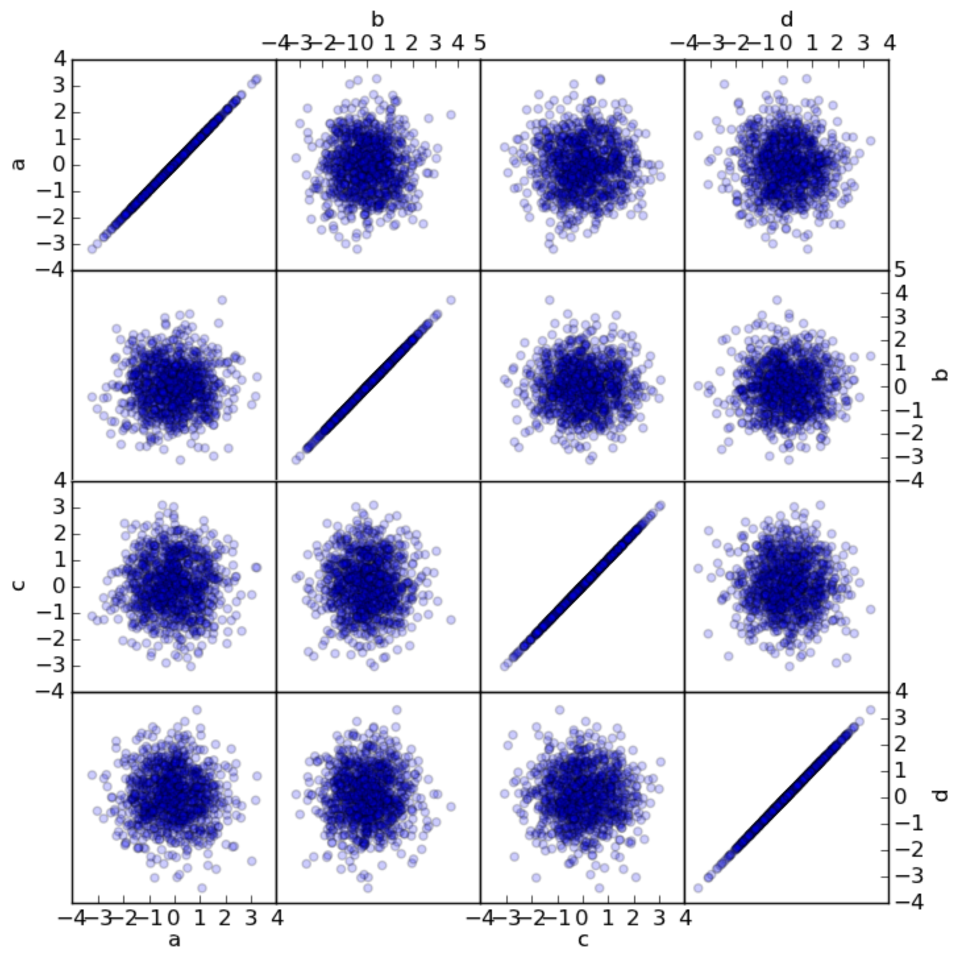
## 1.1 v.0.7.3 (April 12, 2012)

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.

### 1.1.1 New features

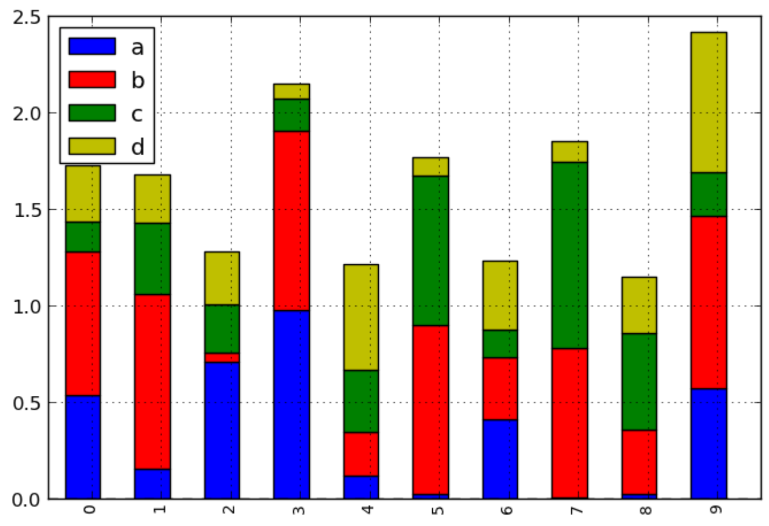
- New *fixed width file reader*, `read_fwf`
- New *scatter_matrix* function for making a scatter plot matrix

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

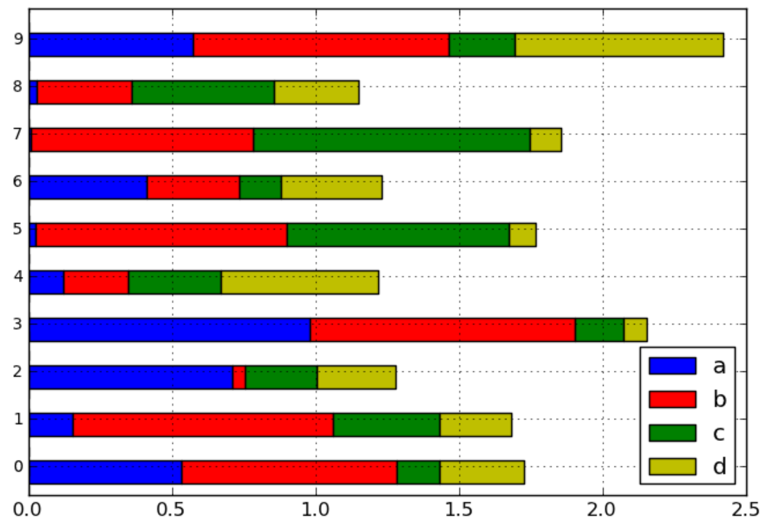


- Add `stacked` argument to Series and DataFrame's `plot` method for *stacked bar plots*.

```
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

## 1.1.2 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 [923]: series = Series(['Steve', np.nan, 'Joe'])

In [924]: series == 'Steve'
Out[924]:
0     True
1    False
2    False

In [925]: series != 'Steve'
Out[925]:
0    False
1     True
2     True
```

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

```
In [926]: mask = series == 'Steve'

In [927]: series[mask & series.notnull()]
Out[927]: 0    Steve
```

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.

### 1.1.3 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 [928]: 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 [929]: df
Out[929]:
     A      B         C         D
0  foo    one -0.541264  2.801614
1  bar    one -0.722290  1.669853
2  foo    two -0.478428  0.254501
3  bar  three  2.850221 -0.682682
4  foo    two -0.350942 -0.697727
5  bar    two -1.581790 -1.092094
6  foo    one  1.113061  0.321042
7  foo  three -1.868914  0.106481

In [930]: grouped = df.groupby('A')['C']

In [931]: grouped.describe()
Out[931]:
A
bar  count    3.000000
     mean     0.182047
     std      2.350329
     min     -1.581790
     25%     -1.152040
     50%     -0.722290
     75%      1.063965
     max      2.850221
foo  count    5.000000
     mean    -0.425297
     std      1.057399
     min     -1.868914
     25%     -0.541264
     50%     -0.478428
     75%     -0.350942
     max      1.113061

In [932]: grouped.apply(lambda x: x.order()[-2:]) # top 2 values
Out[932]:
A
bar  1   -0.722290
     3    2.850221
foo  4   -0.350942
     6    1.113061
```

## 1.2 v.0.7.2 (March 16, 2012)

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

### 1.2.1 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)

### 1.2.2 Performance improvements

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

## 1.3 v.0.7.1 (February 29, 2012)

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

### 1.3.1 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

### 1.3.2 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)

## 1.4 v.0.7.0 (February 9, 2012)

### 1.4.1 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 [933]: df = DataFrame(randn(10, 4))

In [934]: df.apply(lambda x: x.describe())
Out[934]:
               0          1          2          3
count  10.000000  10.000000  10.000000  10.000000
mean   -0.372564   0.069529   0.149059  -0.135687
std     0.544436   1.021552   1.537344   0.905893
min    -1.039777  -1.246778  -3.300939  -1.452203
25%    -0.780794  -0.464921  -0.307157  -0.676272
50%    -0.463086  -0.046551   0.563806  -0.101236
75%     0.015532   0.605550   1.146627   0.354234
max     0.666992   2.182948   1.759328   1.369669
```

- *Add* `reorder_levels` method to Series and DataFrame (PR534)
- *Add* dict-like `get` function to DataFrame and Panel (PR521)
- *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, PR552, others)
- *Add* attribute-based item access to `Panel` and add IPython completion (PR563)
- *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 (PR563)
- *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.

## 1.4.2 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 [935]: s = Series(randn(10), index=range(0, 20, 2))

In [936]: s
Out[936]:
0     1.892368
2     1.091098
4     0.296310
6    -1.381535
8    -0.219765
10   -1.370863
12    1.256251
14   -0.687987
16   -0.715853
18   -1.223952

In [937]: s[0]
Out[937]: 1.8923684617651539

In [938]: s[2]
Out[938]: 1.091097799867949

In [939]: s[4]
Out[939]: 0.2963101333219374
```

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` |

### 1.4.3 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 [940]: s = Series(randn(6), index=list('gmkaec'))

In [941]: s
Out[941]:
g   -0.566048
m    0.240007
k    0.780541
a    0.060935
e    1.546728
c    1.180750
```

Then this is OK:

```
In [942]: s.ix['k':'e']
Out[942]:
k    0.780541
a    0.060935
e    1.546728
```

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 [943]: s2 = s.sort_index()

In [944]: s2
Out[944]:
a    0.060935
c    1.180750
e    1.546728
g   -0.566048
k    0.780541
m    0.240007

In [945]: s2.ix['b':'h']
Out[945]:
c    1.180750
e    1.546728
g   -0.566048
```

### 1.4.4 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 [946]: s = Series(randn(6), index=list('acegkm'))

In [947]: s
Out[947]:
a    0.730258
c    0.294420
e   -0.299092
g    0.923881
k   -1.110300
m    1.559852

In [948]: s[['m', 'a', 'c', 'e']]
Out[948]:
m    1.559852
a    0.730258
c    0.294420
e   -0.299092

In [949]: s['b':'l']
Out[949]:
c    0.294420
e   -0.299092
g    0.923881
k   -1.110300

In [950]: s['c':'k']
Out[950]:
c    0.294420
e   -0.299092
g    0.923881
k   -1.110300
```

In the case of integer indexes, the behavior will be exactly as before (shadowing `ndarray`):

```
In [951]: s = Series(randn(6), index=range(0, 12, 2))

In [952]: s[[4, 0, 2]]
Out[952]:
4   -1.536269
0    1.313771
2    1.389703

In [953]: s[1:5]
Out[953]:
2    1.389703
4   -1.536269
6   -1.643498
8    1.332007
```

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

### 1.4.5 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)

### 1.4.6 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)

## 1.5 v.0.6.1 (December 13, 2011)

### 1.5.1 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 (PR435)
- 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* (PR453)
- 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 (PR482)
- *Can pass* DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH #462)
- MultiIndex.get_level_values can *accept the level name*

### 1.5.2 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)

## 1.6 v.0.6.0 (November 25, 2011)

### 1.6.1 New Features

- *Added* `melt` function to `pandas.core.reshape`
- *Added* `level` parameter to group by level in Series and DataFrame descriptive statistics (PR313)

- *Added* `head` and `tail` methods to Series, analogous to to DataFrame (PR296)
- *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 (PR321)
- *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, PR362)
- *Added* fast `get_value` and `put_value` methods to DataFrame (GH360)
- *Added* `cov` instance methods to Series and DataFrame (GH194, PR362)
- *Added* `kind='bar'` option to `DataFrame.plot` (PR348)
- *Added* `idxmin` and `idxmax` to Series and DataFrame (PR286)
- *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 (PR387)
- *Added* support for MaskedArray data in DataFrame, masked values converted to NaN (PR396)
- *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)

### 1.6.2 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 (PR355)
- 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)

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

### 1.7.1 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. (PR233, GH230)
- *Implemented* `Series.describe` for Series containing objects (PR241)
- *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 (PR334)
- *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` (PR244)
- TODO: DOCS ABOUT TAKE METHODS

### 1.7.2 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

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

### 1.8.1 New Features

- Added Python 3 support using 2to3 (PR200)
- *Added* `name` attribute to `Series`, now prints as part of `Series.__repr__`
- *Added* instance methods `isnull` and `notnull` to Series (PR209, GH203)
- *Added* `Series.align` method for aligning two series with choice of join method (ENH56)
- *Added* method `get_level_values` to `MultiIndex` (IS188)
- *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` (PR146)
- `read_csv` can *read* multiple columns into a `MultiIndex`; DataFrame's `to_csv` method writes out a corresponding `MultiIndex` (PR151)
- `DataFrame.rename` has a new `copy` parameter to *rename* a DataFrame in place (ENHed)
- *Enable* unstacking by name (PR142)
- *Enable* `sortlevel` to work by level (PR141)

## 1.8.2 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

CHAPTER

TWO

# 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 2.5 to 2.7 and Python 3.1+, although Python 3 support is less well tested. Python 2.4 support is being phased out since the userbase has shrunk significantly. Continuing Python 2.4 support will require either monetary development support or someone contributing to the project to maintain compatibility.

## 2.2 Binary installers

Available on PyPI

## 2.3 Dependencies

- NumPy: 1.4.0 or higher. Recommend 1.5.1 or higher
- python-dateutil 1.5

## 2.4 Optional dependencies

- SciPy: miscellaneous statistical functions
- PyTables: necessary for HDF5-based storage
- matplotlib: for plotting
- **scikits.statsmodels**
  - Needed for parts of `pandas.stats`
- **pytz**
  - Needed for time zone support with `DateRange`

**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.

## 2.5 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
```

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

```
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
```

## 2.6 Running the test suite

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

```
$ nosetests pandas
..........................................................................
.......................S..................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
.................S........................................................
....
----------------------------------------------------------------------
Ran 818 tests in 21.631s

OK (SKIP=2)
```

CHAPTER

THREE

# FREQUENTLY ASKED QUESTIONS (FAQ)

CHAPTER

FOUR

# 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 (DateRange) 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-efficent "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 | TimeSeries | 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-contiguousness matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set there is likely to be a "right" way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

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

```
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

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

CHAPTER

FIVE

# 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 [225]: import numpy as np

# will use a lot in examples
In [226]: randn = np.random.randn

In [227]: from pandas import *
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. 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.

## 5.1 Series

`Series` is a one-dimensional labeled array (technically a subclass of ndarray) 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 [228]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [229]: s
Out[229]:
```

```
a   -0.284
b   -1.537
c    0.163
d   -0.648
e   -1.703

In [230]: s.index
Out[230]: Index([a, b, c, d, e], dtype=object)

In [231]: Series(randn(5))
Out[231]:
0    0.654
1   -1.146
2    1.144
3    0.167
4    0.148
```

---

**Note:** The values in the index must be unique. If they are not, an exception will **not** be raised immediately, but attempting any operation involving the index will later result in an exception. In other words, the Index object containing the labels "lazily" checks whether the values are unique. 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 [232]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [233]: Series(d)
Out[233]:
a    0
b    1
c    2

In [234]: Series(d, index=['b', 'c', 'd', 'a'])
Out[234]:
b      1
c      2
d    NaN
a      0
```

---

**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 [235]: Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[235]:
a    5
b    5
c    5
d    5
e    5
```

### 5.1.1 Series is ndarray-like

As a subclass of ndarray, Series is a valid argument to most NumPy functions and behaves similarly to a NumPy array. However, things like slicing also slice the index.

```
In [236]: s[0]
Out[236]: -0.28367872471747613

In [237]: s[:3]
Out[237]:
a   -0.284
b   -1.537
c    0.163

In [238]: s[s > s.median()]
Out[238]:
a   -0.284
c    0.163

In [239]: s[[4, 3, 1]]
Out[239]:
e   -1.703
d   -0.648
b   -1.537

In [240]: np.exp(s)
Out[240]:
a    0.753
b    0.215
c    1.177
d    0.523
e    0.182
```

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

### 5.1.2 Series is dict-like

A Series is alike a fixed-size dict in that you can get and set values by index label:

```
In [241]: s['a']
Out[241]: -0.28367872471747613

In [242]: s['e'] = 12.

In [243]: s
Out[243]:
a    -0.284
b    -1.537
c     0.163
d    -0.648
e    12.000

In [244]: 'e' in s
Out[244]: True

In [245]: 'f' in s
Out[245]: False
```

If a label is not contained, an exception

```
>>> s['f']
KeyError: 'f'

>>> s.get('f')
nan
```

### 5.1.3 Vectorized operations and label alignment with Series

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

```
In [246]: s + s
Out[246]:
a    -0.567
b    -3.074
c     0.326
d    -1.296
e    24.000

In [247]: s * 2
Out[247]:
a    -0.567
b    -3.074
c     0.326
d    -1.296
e    24.000

In [248]: np.exp(s)
Out[248]:
a         0.753
b         0.215
c         1.177
d         0.523
e    162754.791
```

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 [249]: s[1:] + s[:-1]
Out[249]:
a      NaN
b   -3.074
c    0.326
d   -1.296
e      NaN
```

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.

### 5.1.4 Name attribute

Series can also have a `name` attribute:

```
In [250]: s = Series(np.random.randn(5), name='something')

In [251]: s
Out[251]:
0   -1.334
1   -0.171
2    0.050
3   -0.650
4   -1.084
Name: something

In [252]: s.name
Out[252]: 'something'
```

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

## 5.2 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.

### 5.2.1 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 [253]: d = {'one' : Series([1., 2., 3.], index=['a', 'b', 'c']),
   .....:      'two' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}

In [254]: df = DataFrame(d)
In [255]: df
Out[255]:
   one  two
a    1    1
b    2    2
c    3    3
d  NaN    4

In [256]: DataFrame(d, index=['d', 'b', 'a'])
Out[256]:
   one  two
d  NaN    4
b    2    2
a    1    1

In [257]: DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[257]:
   two  three
d    4    NaN
b    2    NaN
a    1    NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

**Note:** When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [258]: df.index
Out[258]: Index([a, b, c, d], dtype=object)

In [259]: df.columns
Out[259]: Index([one, two], dtype=object)
```

### 5.2.2 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 [260]: d = {'one' : [1., 2., 3., 4.],
   .....:      'two' : [4., 3., 2., 1.]}

In [261]: DataFrame(d)
Out[261]:
   one  two
0    1    4
1    2    3
2    3    2
3    4    1

In [262]: DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[262]:
   one  two
a    1    4
b    2    3
```

```
c    3    2
d    4    1
```

### 5.2.3 From structured or record array

This case is handled identically to a dict of arrays.

```
In [263]: data = np.zeros((2,),dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])

In [264]: data[:] = [(1,2.,'Hello'),(2,3.,"World")]

In [265]: DataFrame(data)
Out[265]:
   A  B      C
0  1  2  Hello
1  2  3  World

In [266]: DataFrame(data, index=['first', 'second'])
Out[266]:
        A  B      C
first   1  2  Hello
second  2  3  World

In [267]: DataFrame(data, columns=['C', 'A', 'B'])
Out[267]:
       C  A  B
0  Hello  1  2
1  World  2  3
```

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

### 5.2.4 From a list of dicts

```
In [268]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [269]: DataFrame(data2)
Out[269]:
   a   b   c
0  1   2 NaN
1  5  10  20

In [270]: DataFrame(data2, index=['first', 'second'])
Out[270]:
        a   b   c
first   1   2 NaN
second  5  10  20

In [271]: DataFrame(data2, columns=['a', 'b'])
Out[271]:
   a   b
0  1   2
1  5  10
```

### 5.2.5 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.

### 5.2.6 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 [272]: data
Out[272]:
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', '|S10')])

In [273]: DataFrame.from_records(data, index='C')
Out[273]:
       A  B
Hello  1  2
World  2  3
```

**DataFrame.from_items**

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

```
In [274]: DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
Out[274]:
   A  B
0  1  4
1  2  5
2  3  6
```

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

```
In [275]: DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
   .....:                      orient='index', columns=['one', 'two', 'three'])
Out[275]:
   one  two  three
A    1    2      3
B    4    5      6
```

### 5.2.7 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 [276]: df['one']
Out[276]:
a      1
b      2
c      3
d    NaN
Name: one

In [277]: df['three'] = df['one'] * df['two']

In [278]: df['flag'] = df['one'] > 2

In [279]: df
Out[279]:
   one  two  three   flag
a    1    1      1  False
b    2    2      4  False
c    3    3      9   True
d  NaN    4    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [280]: del df['two']

In [281]: three = df.pop('three')

In [282]: df
Out[282]:
   one   flag
a    1  False
b    2  False
c    3   True
d  NaN  False
```

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

```
In [283]: df['foo'] = 'bar'

In [284]: df
Out[284]:
   one   flag  foo
a    1  False  bar
b    2  False  bar
c    3   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [285]: df['one_trunc'] = df['one'][:2]

In [286]: df
Out[286]:
   one   flag  foo  one_trunc
a    1  False  bar          1
```

```
b    2  False  bar          2
c    3   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [287]: df.insert(1, 'bar', df['one'])

In [288]: df
Out[288]:
   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
```

### 5.2.8 Indexing / Selection

The basics of indexing are as follows:

| Operation | Syntax | Result |
|---|---|---|
| Select column | `df[col]` | Series |
| Select row by label | `df.xs(label)` or `df.ix[label]` | Series |
| Select row by location (int) | `df.ix[loc]` | Series |
| Slice rows | `df[5:10]` | DataFrame |
| Select rows by boolean vector | `df[bool_vec]` | DataFrame |

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

```
In [289]: df.xs('b')
Out[289]:
one              2
bar              2
flag         False
foo            bar
one_trunc        2
Name: b

In [290]: df.ix[2]
Out[290]:
one             3
bar             3
flag         True
foo           bar
one_trunc     NaN
Name: c
```

Note if a DataFrame contains columns of multiple dtypes, the dtype of the row will be chosen to accommodate all of the data types (dtype=object is the most general).

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 lables in the *section on reindexing*.

### 5.2.9 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 [291]: df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])

In [292]: df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])

In [293]: df + df2
Out[293]:
       A      B      C   D
0  0.002  0.831 -1.171 NaN
1  0.269  3.238  0.268 NaN
2 -0.513  0.090  3.120 NaN
3  0.684  1.042  2.282 NaN
4  1.188 -1.805  1.166 NaN
5 -1.972  0.407 -3.513 NaN
6 -0.850  0.306  0.237 NaN
7    NaN    NaN    NaN NaN
8    NaN    NaN    NaN NaN
9    NaN    NaN    NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus broadcasting row-wise. For example:

```
In [294]: df - df.ix[0]
Out[294]:
       A      B      C      D
0  0.000  0.000  0.000  0.000
1  0.808  1.358  2.420 -1.339
2 -0.070 -0.814  4.027 -0.293
3  0.073 -0.519  2.195 -0.002
4  1.342  0.372  2.510 -1.543
5 -0.523  0.665  0.942  0.018
6  0.101 -0.066  1.943 -0.817
7  0.744  0.834  3.473 -0.665
8  0.045  0.772  1.406 -0.965
9  0.860  1.677  1.462 -1.165
```

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 [295]: index = DateRange('1/1/2000', periods=8)

In [296]: df = DataFrame(randn(8, 3), index=index,
   .....:                columns=['A', 'B', 'C'])

In [297]: df
Out[297]:
                A      B      C
2000-01-03  0.361 -0.192 -0.058
2000-01-04 -0.646 -1.051 -0.716
2000-01-05  0.613  0.501 -1.380
2000-01-06  0.624  0.790  0.818
2000-01-07  1.559  0.335  0.919
2000-01-10 -1.381  0.365 -1.811
2000-01-11 -0.673 -1.968 -0.401
2000-01-12 -0.583 -0.999 -0.629
```

```
In [298]: type(df['A'])
Out[298]: pandas.core.series.TimeSeries

In [299]: df - df['A']
Out[299]:
            A      B      C
2000-01-03  0 -0.553 -0.419
2000-01-04  0 -0.405 -0.070
2000-01-05  0 -0.112 -1.993
2000-01-06  0  0.166  0.195
2000-01-07  0 -1.224 -0.640
2000-01-10  0  1.746 -0.429
2000-01-11  0 -1.294  0.272
2000-01-12  0 -0.416 -0.046
```

Technical purity aside, this case is so common in practice that supporting the special case is preferable to the alternative of forcing the user to transpose and do column-based alignment like so:

```
In [300]: (df.T - df['A']).T
Out[300]:
            A      B      C
2000-01-03  0 -0.553 -0.419
2000-01-04  0 -0.405 -0.070
2000-01-05  0 -0.112 -1.993
2000-01-06  0  0.166  0.195
2000-01-07  0 -1.224 -0.640
2000-01-10  0  1.746 -0.429
2000-01-11  0 -1.294  0.272
2000-01-12  0 -0.416 -0.046
```

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 [301]: df * 5 + 2
Out[301]:
                A      B      C
2000-01-03  3.805  1.040  1.708
2000-01-04 -1.230 -3.257 -1.580
2000-01-05  5.064  4.504 -4.902
2000-01-06  5.118  5.948  6.091
2000-01-07  9.795  3.676  6.597
2000-01-10 -4.906  3.826 -7.053
2000-01-11 -1.367 -7.838 -0.006
2000-01-12 -0.915 -2.993 -1.146

In [302]: 1 / df
Out[302]:
                A      B       C
2000-01-03  2.770 -5.211 -17.148
2000-01-04 -1.548 -0.951  -1.397
2000-01-05  1.632  1.997  -0.724
2000-01-06  1.604  1.266   1.222
2000-01-07  0.641  2.983   1.088
2000-01-10 -0.724  2.738  -0.552
2000-01-11 -1.485 -0.508  -2.493
2000-01-12 -1.715 -1.001  -1.589

In [303]: df ** 4
```

```
Out[303]:
                A       B       C
2000-01-03  0.017   0.001   0.000
2000-01-04  0.174   1.222   0.263
2000-01-05  0.141   0.063   3.631
2000-01-06  0.151   0.389   0.448
2000-01-07  5.908   0.013   0.715
2000-01-10  3.639   0.018  10.748
2000-01-11  0.206  14.988   0.026
2000-01-12  0.116   0.995   0.157
```

Boolean operators work as well:

```
In [304]: df1 = DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)

In [305]: df2 = DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)

In [306]: df1 & df2
Out[306]:
       a      b
0  False  False
1  False   True
2   True  False

In [307]: df1 | df2
Out[307]:
      a     b
0  True  True
1  True  True
2  True  True

In [308]: df1 ^ df2
Out[308]:
       a      b
0   True   True
1   True  False
2  False   True

In [309]: -df1
Out[309]:
       a      b
0  False   True
1   True  False
2  False  False
```

## 5.2.10 Transposing

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

```
# only show the first 5 rows
In [310]: df[:5].T
Out[310]:
   2000-01-03  2000-01-04  2000-01-05  2000-01-06  2000-01-07
A       0.361      -0.646       0.613       0.624       1.559
B      -0.192      -1.051       0.501       0.790       0.335
C      -0.058      -0.716      -1.380       0.818       0.919
```

### 5.2.11 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 [311]: np.exp(df)
Out[311]:
                A      B      C
2000-01-03  1.435  0.825  0.943
2000-01-04  0.524  0.349  0.489
2000-01-05  1.846  1.650  0.251
2000-01-06  1.866  2.203  2.267
2000-01-07  4.754  1.398  2.508
2000-01-10  0.251  1.441  0.164
2000-01-11  0.510  0.140  0.670
2000-01-12  0.558  0.368  0.533

In [312]: np.asarray(df)
Out[312]:
array([[ 0.361 , -0.1919, -0.0583],
       [-0.646 , -1.0514, -0.716 ],
       [ 0.6128,  0.5007, -1.3804],
       [ 0.6236,  0.7896,  0.8183],
       [ 1.5591,  0.3352,  0.9194],
       [-1.3812,  0.3652, -1.8106],
       [-0.6734, -1.9676, -0.4012],
       [-0.583 , -0.9986, -0.6293]])
```

The dot method on DataFrame implements matrix multiplication:

```
In [313]: df.T.dot(df)
Out[313]:
       A      B      C
A  6.444  3.334  4.677
B  3.334  7.131  1.784
C  4.677  1.784  7.772
```

Similarly, the dot method on Series implements dot product:

```
In [314]: s1 = Series(np.arange(5,10))

In [315]: s1.dot(s1)
Out[315]: 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.

### 5.2.12 Console display

For very large DataFrame objects, only a summary will be printed to the console (here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [316]: baseball = read_csv('data/baseball.csv')

In [317]: print baseball
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 88641 to 89534
Data 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 [318]: print baseball.ix[-20:, :12].to_string()
              id  year  stint team  lg    g   ab   r    h  X2b  X3b  hr
88641  womacto01  2006      2  CHN  NL   19   50   6   14    1    0   1
88643  schilcu01  2006      1  BOS  AL   31    2   0    1    0    0   0
88645  myersmi01  2006      1  NYA  AL   62    0   0    0    0    0   0
88649  helliri01  2006      1  MIL  NL   20    3   0    0    0    0   0
88650  johnsra05  2006      1  NYA  AL   33    6   0    1    0    0   0
88652  finlest01  2006      1  SFN  NL  139  426  66  105   21   12   6
88653  gonzalu01  2006      1  ARI  NL  153  586  93  159   52    2  15
88662   seleaa01  2006      1  LAN  NL   28   26   2    5    1    0   0
89177  francju01  2007      2  ATL  NL   15   40   1   10    3    0   0
89178  francju01  2007      1  NYN  NL   40   50   7   10    0    0   1
89330   zaungr01  2007      1  TOR  AL  110  331  43   80   24    1  10
89333  witasja01  2007      1  TBA  AL    3    0   0    0    0    0   0
89334  williwo02  2007      1  HOU  NL   33   59   3    6    0    0   1
89335  wickmbo01  2007      2  ARI  NL    8    0   0    0    0    0   0
89336  wickmbo01  2007      1  ATL  NL   47    0   0    0    0    0   0
89337  whitero02  2007      1  MIN  AL   38  109   8   19    4    0   4
89338  whiteri01  2007      1  HOU  NL   20    1   0    0    0    0   0
89339  wellsda01  2007      2  LAN  NL    7   15   2    4    1    0   0
89340  wellsda01  2007      1  SDN  NL   22   38   1    4    0    0   0
89341  weathda01  2007      1  CIN  NL   67    0   0    0    0    0   0
89343  walketo04  2007      1  OAK  AL   18   48   5   13    1    0   0
89345  wakefti01  2007      1  BOS  AL    1    2   0    0    0    0   0
89347  vizquom01  2007      1  SFN  NL  145  513  54  126   18    3   4
89348  villoro01  2007      1  NYA  AL    6    0   0    0    0    0   0
89352  valenjo03  2007      1  NYN  NL   51  166  18   40   11    1   3
89354  trachst01  2007      2  CHN  NL    4    7   0    1    0    0   0
89355  trachst01  2007      1  BAL  AL    3    5   0    0    0    0   0
89359  timlimi01  2007      1  BOS  AL    4    0   0    0    0    0   0
89360  thomeji01  2007      1  CHA  AL  130  432  79  119   19    0  35
```

```
89361  thomafr04  2007      1  TOR  AL  155  531   63  147   30    0  26
89363  tavarju01  2007      1  BOS  AL    2    4    0    1    0    0   0
89365  sweenma01  2007      2  LAN  NL   30   33    2    9    1    0   0
89366  sweenma01  2007      1  SFN  NL   76   90   18   23    8    0   2
89367  suppaje01  2007      1  MIL  NL   33   61    4    8    0    0   0
89368  stinnke01  2007      1  SLN  NL   26   82    7   13    3    0   1
89370  stantmi02  2007      1  CIN  NL   67    2    0    0    0    0   0
89371  stairma01  2007      1  TOR  AL  125  357   58  103   28    1  21
89372  sprinru01  2007      1  SLN  NL   72    1    0    0    0    0   0
89374   sosasa01  2007      1  TEX  AL  114  412   53  104   24    1  21
89375  smoltjo01  2007      1  ATL  NL   30   54    1    5    1    0   0
89378  sheffga01  2007      1  DET  AL  133  494  107  131   20    1  25
89381   seleaa01  2007      1  NYN  NL   31    4    0    0    0    0   0
89382  seaneru01  2007      1  LAN  NL   68    1    0    0    0    0   0
89383  schmija01  2007      1  LAN  NL    6    7    1    1    0    0   1
89384  schilcu01  2007      1  BOS  AL    1    2    0    1    0    0   0
89385  sandere02  2007      1  KCA  AL   24   73   12   23    7    0   2
89388  rogerke01  2007      1  DET  AL    1    2    0    0    0    0   0
89389  rodriiv01  2007      1  DET  AL  129  502   50  141   31    3  11
89396  ramirma02  2007      1  BOS  AL  133  483   84  143   33    1  20
89398  piazzmi01  2007      1  OAK  AL   83  309   33   85   17    1   8
89400  perezne01  2007      1  DET  AL   33   64    5   11    3    0   1
89402   parkch01  2007      1  NYN  NL    1    1    0    0    0    0   0
89406  oliveda02  2007      1  LAA  AL    5    0    0    0    0    0   0
89410  myersmi01  2007      1  NYA  AL    6    1    0    0    0    0   0
89411  mussimi01  2007      1  NYA  AL    2    2    0    0    0    0   0
89412  moyerja01  2007      1  PHI  NL   33   73    4    9    2    0   0
89420   mesajo01  2007      1  PHI  NL   38    0    0    0    0    0   0
89421  martipe02  2007      1  NYN  NL    5    9    1    1    1    0   0
89425  maddugr01  2007      1  SDN  NL   33   62    2    9    2    0   0
89426  mabryjo01  2007      1  COL  NL   28   34    4    4    1    0   1
89429  loftoke01  2007      2  CLE  AL   52  173   24   49    9    3   0
89430  loftoke01  2007      1  TEX  AL   84  317   62   96   16    3   7
89431  loaizes01  2007      1  LAN  NL    5    7    0    1    0    0   0
89438  kleskry01  2007      1  SFN  NL  116  362   51   94   27    3   6
89439   kentje01  2007      1  LAN  NL  136  494   78  149   36    1  20
89442  jonesto02  2007      1  DET  AL    5    0    0    0    0    0   0
89445  johnsra05  2007      1  ARI  NL   10   15    0    1    0    0   0
89450  hoffmtr01  2007      1  SDN  NL   60    0    0    0    0    0   0
89451  hernaro01  2007      2  LAN  NL   22    0    0    0    0    0   0
89452  hernaro01  2007      1  CLE  AL    2    0    0    0    0    0   0
89460  guarded01  2007      1  CIN  NL   15    0    0    0    0    0   0
89462  griffke02  2007      1  CIN  NL  144  528   78  146   24    1  30
89463  greensh01  2007      1  NYN  NL  130  446   62  130   30    1  10
89464  graffto01  2007      1  MIL  NL   86  231   34   55    8    0   9
89465  gordoto01  2007      1  PHI  NL   44    0    0    0    0    0   0
89466  gonzalu01  2007      1  LAN  NL  139  464   70  129   23    2  15
89467  gomezch02  2007      2  CLE  AL   19   53    4   15    2    0   0
89468  gomezch02  2007      1  BAL  AL   73  169   17   51   10    1   1
89469  glavito02  2007      1  NYN  NL   33   56    3   12    1    0   0
89473  floydcl01  2007      1  CHN  NL  108  282   40   80   10    1   9
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  clemero02  2007      1  NYA  AL    2    2    0    1    0    0   0
89498  claytro01  2007      2  BOS  AL    8    6    1    0    0    0   0
89499  claytro01  2007      1  TOR  AL   69  189   23   48   14    0   1
89501  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
```

## 5.2.13 DataFrame column types

The four main types stored in pandas objects are float, int, boolean, and object. A convenient `dtypes` attribute return a Series with the data type of each column:

```
In [319]: baseball.dtypes
Out[319]:
id          object
year         int64
stint        int64
team        object
lg          object
g            int64
ab           int64
r            int64
h            int64
X2b          int64
X3b          int64
hr           int64
rbi        float64
sb         float64
cs         float64
bb           int64
so         float64
ibb        float64
hbp        float64
sh         float64
sf         float64
gidp       float64
```

The related method `get_dtype_counts` will return the number of columns of each type:

```
In [320]: baseball.get_dtype_counts()
Out[320]:
float64     9
int64      10
object      3
```

## 5.2.14 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 [321]: df = DataFrame({'foo1' : np.random.randn(5),
   .....:                 'foo2' : np.random.randn(5)})

In [322]: df
Out[322]:
       foo1      foo2
0 -0.548001 -0.966162
1 -0.852612 -0.332601
2 -0.126250 -1.327330
3  1.765997  1.225847
4 -1.593297 -0.348395

In [323]: df.foo1
Out[323]:
0   -0.548001
1   -0.852612
2   -0.126250
3    1.765997
4   -1.593297
Name: foo1
```

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

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

## 5.3 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:

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

```
In [324]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:            major_axis=DateRange('1/1/2000', periods=5),
   .....:            minor_axis=['A', 'B', 'C', 'D'])

In [325]: wp
Out[325]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major) x 4 (minor)
Items: Item1 to Item2
Major axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor axis: A to D
```

### 5.3.2 From dict of DataFrame objects

```
In [326]: data = {'Item1' : DataFrame(randn(4, 3)),
   .....:         'Item2' : DataFrame(randn(4, 2))}

In [327]: Panel(data)
Out[327]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major) x 3 (minor)
Items: Item1 to Item2
Major axis: 0 to 3
Minor 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 |
|---|---|---|
| intersect | `False` | drops elements whose indices do not align |
| orient | `items` | use `minor` to use DataFrames' columns as panel items |

For example, compare to the construction above:

```
In [328]: Panel.from_dict(data, orient='minor')
Out[328]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major) x 2 (minor)
Items: 0 to 2
Major axis: 0 to 3
Minor 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 [329]: df = DataFrame({'a': ['foo', 'bar', 'baz'],
   .....:                 'b': np.random.randn(3)})

In [330]: df
Out[330]:
     a         b
0  foo -1.309989
1  bar -1.153000
2  baz  0.606382

In [331]: data = {'item1': df, 'item2': df}

In [332]: panel = Panel.from_dict(data, orient='minor')

In [333]: panel['a']
Out[333]:
  item1 item2
0   foo   foo
1   bar   bar
2   baz   baz

In [334]: panel['b']
Out[334]:
      item1     item2
```

```
0 -1.309989 -1.309989
1 -1.153000 -1.153000
2  0.606382  0.606382

In [335]: panel['b'].dtypes
Out[335]:
item1    float64
item2    float64
```

**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.

### 5.3.3 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 [336]: midx = MultiIndex(levels=[['one', 'two'], ['x','y']], labels=[[1,1,0,0],[1,0,1,0]])

In [337]: df = DataFrame({'A' : [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [338]: df.to_panel()
Out[338]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major) x 2 (minor)
Items: A to B
Major axis: one to two
Minor axis: x to y
```

### 5.3.4 Item selection / addition / deletion

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

```
In [339]: wp['Item1']
Out[339]:
                   A         B         C         D
2000-01-03  1.424840 -0.879202  0.544803  0.627214
2000-01-04  0.696649  0.107734 -0.776466 -1.257592
2000-01-05 -0.600682 -1.504853  1.052050  0.586573
2000-01-06 -2.637236 -0.014948 -1.208531 -1.257020
2000-01-07 -0.499747  0.429787 -0.242210 -0.723848

In [340]: 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.

### 5.3.5 Indexing / Selection

| Operation | Syntax | Result |
|---|---|---|
| Select item | `wp[item]` | DataFrame |
| Get slice at major_axis label | `wp.major_xs(val)` | DataFrame |
| Get slice at minor_axis label | `wp.minor_xs(val)` | DataFrame |

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

```
In [341]: wp['Item1']
Out[341]:
                   A         B         C         D
2000-01-03  1.424840 -0.879202  0.544803  0.627214
2000-01-04  0.696649  0.107734 -0.776466 -1.257592
2000-01-05 -0.600682 -1.504853  1.052050  0.586573
2000-01-06 -2.637236 -0.014948 -1.208531 -1.257020
2000-01-07 -0.499747  0.429787 -0.242210 -0.723848

In [342]: wp.major_xs(wp.major_axis[2])
Out[342]:
      Item1     Item2     Item3
A -0.600682 -2.138612  0.280875
B -1.504853 -0.592654  2.539177
C  1.052050 -1.059136 -0.993309
D  0.586573  0.118816  4.936815

In [343]: wp.minor_axis
Out[343]: Index([A, B, C, D], dtype=object)

In [344]: wp.minor_xs('C')
Out[344]:
               Item1     Item2     Item3
2000-01-03  0.544803 -0.543730 -1.001973
2000-01-04 -0.776466 -1.566259  0.495746
2000-01-05  1.052050 -1.059136 -0.993309
2000-01-06 -1.208531 -0.129101  9.361112
2000-01-07 -0.242210  0.759091 -0.319079
```

### 5.3.6 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 [345]: panel = Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
   .....:               major_axis=DateRange('1/1/2000', periods=5),
   .....:               minor_axis=['a', 'b', 'c', 'd'])

In [346]: panel.to_frame()
Out[346]:
                       one       two     three
major      minor
2000-01-03 a     -0.681101 -0.254002 -0.139981
           b     -0.289724  1.360151 -0.040368
           c     -0.996632 -0.059912 -1.742251
           d     -1.407699 -0.151652  0.234716
2000-01-04 a      1.014104  0.624697  0.876834
           b      0.314226 -1.124779  0.080597
           c     -0.001675  0.072594 -0.000185
           d      0.071823 -1.109831 -0.264704
2000-01-05 a      0.892566 -0.008027 -0.566820
           b      0.680594 -0.121632 -1.643966
           c     -0.339640  0.045829  1.471262
           d      0.214910  0.532980  0.677634
2000-01-06 a     -0.078410  0.228178 -0.485743
           b     -0.177665 -0.210935 -0.342272
           c      0.490838  0.039513 -1.042291
           d     -1.360102 -0.205689 -0.611457
2000-01-07 a      1.592456 -0.764740 -0.141224
           b      1.007100  0.027476  0.007220
           c      0.697835 -0.277396 -0.516147
           d     -1.890591  0.620539  0.446161
```

CHAPTER

SIX

# 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 = DateRange('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=DateRange('1/1/2000', periods=5),
   ...:             minor_axis=['A', 'B', 'C', 'D'])
```

## 6.1 Head and Tail

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

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

In [6]: long_series.head()
Out[6]:
0   -0.395255
1   -0.632260
2    0.173969
3    1.320848
4    0.226964

In [7]: long_series.tail(3)
Out[7]:
997   -1.005834
998    1.125063
999    0.222895
```

## 6.2 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]
Out[8]:
                   A         B         C
2000-01-03 -0.214829 -0.142451 -0.282057
2000-01-04  0.864059 -0.608186  1.536483

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

In [10]: df
Out[10]:
                   a         b         c
2000-01-03 -0.214829 -0.142451 -0.282057
2000-01-04  0.864059 -0.608186  1.536483
2000-01-05 -0.872287 -0.659650 -0.661710
2000-01-06 -0.091525  1.075137 -0.882660
2000-01-07 -1.621064 -0.773457 -0.770732
2000-01-10  0.974817 -0.166694 -0.727561
2000-01-11  0.308386  0.320270 -2.061148
2000-01-12 -1.740628 -0.222302  0.172190
```

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

```
In [11]: s.values
Out[11]: array([-0.4585, -1.8419,  0.0322,  0.0218, -0.9995])

In [12]: df.values
Out[12]:
array([[-0.2148, -0.1425, -0.2821],
       [ 0.8641, -0.6082,  1.5365],
       [-0.8723, -0.6596, -0.6617],
       [-0.0915,  1.0751, -0.8827],
       [-1.6211, -0.7735, -0.7707],
       [ 0.9748, -0.1667, -0.7276],
       [ 0.3084,  0.3203, -2.0611],
       [-1.7406, -0.2223,  0.1722]])

In [13]: wp.values
Out[13]:
array([[[ 0.8383,  1.6071, -1.7119, -0.2156],
        [-1.4868, -0.187 , -0.9808,  1.6427],
        [ 0.7641,  0.4294,  1.2497,  0.5067],
        [ 0.7656, -2.4271, -1.2652, -0.5706],
        [ 1.7256, -1.5106, -0.5281,  0.5593]],
       [[-2.1587,  0.4519,  1.4765, -0.9683],
        [-0.0549,  0.2323,  0.7039, -0.1495],
        [ 0.2353, -0.6546,  1.5889, -0.7716],
        [ 0.108 ,  2.4888,  2.325 ,  0.688 ],
        [ 0.0614, -0.3168, -0.2638, -0.0902]]])
```

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.

## 6.3 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.

### 6.3.1 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]: df
Out[14]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [15]: row = df.ix[1]

In [16]: column = df['two']

In [17]: df.sub(row, axis='columns')
Out[17]:
        one     three       two
a -1.670205       NaN  0.770434
b  0.000000  0.000000  0.000000
c  1.050451 -3.231782  0.182734
d       NaN -2.074872 -0.669178

In [18]: df.sub(row, axis=1)
Out[18]:
        one     three       two
a -1.670205       NaN  0.770434
b  0.000000  0.000000  0.000000
c  1.050451 -3.231782  0.182734
d       NaN -2.074872 -0.669178

In [19]: df.sub(column, axis='index')
Out[19]:
        one     three  two
a -2.725523       NaN    0
b -0.284885  2.153404    0
c  0.582831 -1.261113    0
d       NaN  0.747709    0

In [20]: df.sub(column, axis=0)
Out[20]:
        one     three  two
a -2.725523       NaN    0
b -0.284885  2.153404    0
c  0.582831 -1.261113    0
d       NaN  0.747709    0
```

With Panel, describing the matching behavior is a bit more difficult, so the arithmetic methods instead (and perhaps confusingly?) give you the option to specify the *broadcast axis*. For example, suppose we wished to demean the data over a particular axis. This can be accomplished by taking the mean over an axis and broadcasting over the same axis:

```
In [21]: major_mean = wp.mean(axis='major')

In [22]: major_mean
Out[22]:
      Item1     Item2
A  0.521348 -0.361771
B -0.417654  0.440338
C -0.647243  1.166107
D  0.384496 -0.258330

In [23]: wp.sub(major_mean, axis='major')
Out[23]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major) x 4 (minor)
Items: Item1 to Item2
Major axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor 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...

### 6.3.2 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 [24]: df
Out[24]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650
```

```
In [25]: df2
Out[25]:
        one     three       two
a -1.803562  1.000000  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [26]: df + df2
Out[26]:
        one     three       two
a -3.607123       NaN  1.843923
b -0.266714  4.609863  0.303055
c  1.834187 -1.853702  0.668524
d       NaN  0.460118 -1.035300

In [27]: df.add(df2, fill_value=0)
Out[27]:
        one     three       two
a -3.607123  1.000000  1.843923
b -0.266714  4.609863  0.303055
c  1.834187 -1.853702  0.668524
d       NaN  0.460118 -1.035300
```

### 6.3.3 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 [28]: df1 = DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
   ....:                 'B' : [np.nan, 2., 3., np.nan, 6.]})

In [29]: df2 = DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
   ....:                 'B' : [np.nan, np.nan, 3., 4., 6., 8.]})

In [30]: df1
Out[30]:
    A    B
0   1  NaN
1 NaN    2
2   3    3
3   5  NaN
4 NaN    6

In [31]: df2
Out[31]:
    A    B
0   5  NaN
1   2  NaN
2   4    3
3 NaN    4
4   3    6
5   7    8
```

```
In [32]: df1.combine_first(df2)
Out[32]:
   A   B
0  1 NaN
1  2   2
2  3   3
3  5   4
4  3   6
5  7   8
```

### 6.3.4 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 [33]: combiner = lambda x, y: np.where(isnull(x), y, x)

In [34]: df1.combine(df2, combiner)
Out[34]:
   A   B
0  1 NaN
1  2   2
2  3   3
3  5   4
4  3   6
5  7   8
```

## 6.4 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 [35]: df
Out[35]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [36]: df.mean(0)
Out[36]:
one     -0.339942
```

```
three    0.536047
two      0.222525

In [37]: df.mean(1)
Out[37]:
a   -0.440800
b    0.774367
c    0.108168
d   -0.143796
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

```
In [38]: df.sum(0, skipna=False)
Out[38]:
one           NaN
three         NaN
two      0.890101

In [39]: df.sum(axis=1, skipna=True)
Out[39]:
a   -0.881600
b    2.323102
c    0.324505
d   -0.287591
```

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 [40]: ts_stand = (df - df.mean()) / df.std()

In [41]: ts_stand.std()
Out[41]:
one      1
three    1
two      1

In [42]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [43]: xs_stand.std(1)
Out[43]:
a    1
b    1
c    1
d    1
```

Note that methods like **cumsum** and **cumprod** preserve the location of NA values:

```
In [44]: df.cumsum()
Out[44]:
        one     three       two
a -1.803562       NaN  0.921961
b -1.936919  2.304932  1.073489
c -1.019825  1.378081  1.407751
d       NaN  1.608140  0.890101
```

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 |
| `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 [45]: np.mean(df['one'])
Out[45]: -0.33994174173412101

In [46]: np.mean(df['one'].values)
Out[46]: nan
```

`Series` also has a method `nunique` which will return the number of unique non-null values:

```
In [47]: series = Series(randn(500))

In [48]: series[20:500] = np.nan

In [49]: series[10:20]  = 5

In [50]: series.nunique()
Out[50]: 11
```

### 6.4.1 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 [51]: series = Series(randn(1000))

In [52]: series[::2] = np.nan

In [53]: series.describe()
Out[53]:
count    500.000000
mean      -0.022562
std        1.016534
min       -3.909767
25%       -0.695256
50%       -0.026351
```

```
75%         0.655005
max         2.818375

In [54]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [55]: frame.ix[::2] = np.nan

In [56]: frame.describe()
Out[56]:
                  a           b           c           d           e
count    500.000000  500.000000  500.000000  500.000000  500.000000
mean       0.087006    0.016977    0.002514    0.064459    0.047516
std        1.010381    1.028309    1.003938    0.982985    1.029576
min       -3.033923   -3.186314   -2.650052   -2.754301   -3.504149
25%       -0.581764   -0.653484   -0.713372   -0.595791   -0.638621
50%        0.105008    0.031949   -0.052945    0.031084    0.050620
75%        0.754432    0.724837    0.709446    0.710579    0.725168
max        3.082986    2.666830    2.879336    3.065742    3.151635
```

For a non-numerical Series object, *describe* will give a simple summary of the number of unique values and most frequently occurring values:

```
In [57]: s = Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [58]: s.describe()
Out[58]:
count     9
unique    4
top       a
freq      5
```

There also is a utility function, `value_range` which takes a DataFrame and returns a series with the minimum/maximum values in the DataFrame.

### 6.4.2 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 [59]: s1 = Series(randn(5))

In [60]: s1
Out[60]:
0    1.408007
1   -0.650623
2   -0.750666
3    0.013871
4    1.246326

In [61]: s1.idxmin(), s1.idxmax()
Out[61]: (2, 0)

In [62]: df1 = DataFrame(randn(5,3), columns=['A','B','C'])

In [63]: df1
Out[63]:
          A         B         C
```

```
0 -0.105404 -0.665238  0.470341
1 -1.140010  0.842451  1.362074
2 -0.300080 -0.188657  0.066332
3 -0.441389 -1.186129 -0.535485
4  0.173416  0.192418 -0.647838

In [64]: df1.idxmin(axis=0)
Out[64]:
A    1
B    3
C    4

In [65]: df1.idxmax(axis=1)
Out[65]:
0    C
1    C
2    C
3    A
4    B
```

## 6.5 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 [66]: df.apply(np.mean)
Out[66]:
one     -0.339942
three    0.536047
two      0.222525

In [67]: df.apply(np.mean, axis=1)
Out[67]:
a   -0.440800
b    0.774367
c    0.108168
d   -0.143796

In [68]: df.apply(lambda x: x.max() - x.min())
Out[68]:
one      2.720655
three    3.231782
two      1.439612

In [69]: df.apply(np.cumsum)
Out[69]:
        one     three       two
a -1.803562       NaN  0.921961
b -1.936919  2.304932  1.073489
c -1.019825  1.378081  1.407751
d       NaN  1.608140  0.890101

In [70]: df.apply(np.exp)
Out[70]:
        one      three       two
a  0.164711        NaN  2.514217
```

```
b  0.875153  10.023492  1.163610
c  2.502008   0.395798  1.396909
d       NaN   1.258674  0.595919
```

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 [71]: tsdf = DataFrame(randn(1000, 3), columns=['A', 'B', 'C'],
   ....:                  index=DateRange('1/1/2000', periods=1000))

In [72]: tsdf.apply(lambda x: x.index[x.dropna().argmax()])
Out[72]:
A    2003-04-23 00:00:00
B    2003-09-05 00:00:00
C    2000-06-15 00:00:00
```

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 [73]: tsdf
Out[73]:
                   A         B         C
2000-01-03  0.220295  0.027538 -0.643213
2000-01-04 -0.216840 -0.077551  1.760229
2000-01-05  0.354672 -0.869926 -1.838944
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN
2000-01-12  1.072984  2.086809  0.707775
2000-01-13  0.791807 -0.757370 -0.234677
2000-01-14 -1.481734 -0.724198  1.839899

In [74]: tsdf.apply(Series.interpolate)
Out[74]:
                   A         B         C
2000-01-03  0.220295  0.027538 -0.643213
2000-01-04 -0.216840 -0.077551  1.760229
2000-01-05  0.354672 -0.869926 -1.838944
2000-01-06  0.498335 -0.278579 -1.329600
2000-01-07  0.641997  0.312768 -0.820256
2000-01-10  0.785659  0.904115 -0.310912
2000-01-11  0.929321  1.495462  0.198432
2000-01-12  1.072984  2.086809  0.707775
2000-01-13  0.791807 -0.757370 -0.234677
2000-01-14 -1.481734 -0.724198  1.839899
```

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.

### 6.5.1 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 [75]: f = lambda x: len(str(x))

In [76]: df['one'].map(f)
Out[76]:
a    14
b    15
c    14
d     3
Name: one

In [77]: df.applymap(f)
Out[77]:
   one  three  two
a   14      3   13
b   15     13   14
c   14     15   14
d    3     14   15
```

`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 [78]: s = Series(['six', 'seven', 'six', 'seven', 'six'],
   ....:            index=['a', 'b', 'c', 'd', 'e'])

In [79]: t = Series({'six' : 6., 'seven' : 7.})

In [80]: s
Out[80]:
a      six
b    seven
c      six
d    seven
e      six

In [81]: s.map(t)
Out[81]:
a    6
b    7
c    6
d    7
e    6
```

## 6.6 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 [82]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [83]: s
Out[83]:
a   -0.559624
b    1.008419
c    1.626928
d    0.692417
e    0.452548

In [84]: s.reindex(['e', 'b', 'f', 'd'])
Out[84]:
e    0.452548
b    1.008419
f         NaN
d    0.692417
```

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 [85]: df
Out[85]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [86]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[86]:
      three       two       one
c -0.926851  0.334262  0.917093
f       NaN       NaN       NaN
b  2.304932  0.151528 -0.133357
```

For convenience, you may utilize the `reindex_axis` method, which takes the labels and a keyword `axis` paramater.

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 [87]: rs = s.reindex(df.index)

In [88]: rs
Out[88]:
a   -0.559624
```

```
b    1.008419
c    1.626928
d    0.692417

In [89]: rs.index is df.index
Out[89]: 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 sprinking a few explicit `reindex` calls here and there can have an impact.

### 6.6.1 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 [90]: df
Out[90]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [91]: df2
Out[91]:
        one       two
a -1.463620  0.452711
b  0.206585 -0.317723
c  1.257035 -0.134988

In [92]: df.reindex_like(df2)
Out[92]:
        one       two
a -1.803562  0.921961
b -0.133357  0.151528
c  0.917093  0.334262
```

### 6.6.2 Reindexing with `reindex_axis`

### 6.6.3 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 [93]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [94]: s1 = s[:4]

In [95]: s2 = s[1:]

In [96]: s1.align(s2)
Out[96]:
(a    1.125606
b   -0.426032
c   -0.061063
d   -0.644590
e         NaN,
 a         NaN
b   -0.426032
c   -0.061063
d   -0.644590
e    1.551388)

In [97]: s1.align(s2, join='inner')
Out[97]:
(b   -0.426032
c   -0.061063
d   -0.644590,
 b   -0.426032
c   -0.061063
d   -0.644590)

In [98]: s1.align(s2, join='left')
Out[98]:
(a    1.125606
b   -0.426032
c   -0.061063
d   -0.644590,
 a         NaN
b   -0.426032
c   -0.061063
d   -0.644590)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [99]: df.align(df2, join='inner')
Out[99]:
(        one       two
a -1.803562  0.921961
b -0.133357  0.151528
c  0.917093  0.334262,
         one       two
a -1.463620  0.452711
b  0.206585 -0.317723
c  1.257035 -0.134988)
```

You can also pass an `axis` option to only align on the specified axis:

```
In [100]: df.align(df2, join='inner', axis=0)
Out[100]:
(         one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262,
         one       two
a -1.463620  0.452711
b  0.206585 -0.317723
c  1.257035 -0.134988)
```

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 [101]: df.align(df2.ix[0], axis=1)
Out[101]:
(         one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650,
 one      -1.463620
three          NaN
two       0.452711
Name: a)
```

### 6.6.4 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 [102]: rng = DateRange('1/3/2000', periods=8)

In [103]: ts = Series(randn(8), index=rng)

In [104]: ts2 = ts[[0, 3, 6]]

In [105]: ts
Out[105]:
2000-01-03   -0.759826
2000-01-04    1.154537
2000-01-05   -0.199039
2000-01-06   -0.261202
2000-01-07    0.041710
2000-01-10   -0.476623
2000-01-11   -0.725007
2000-01-12   -0.432569
```

```
In [106]: ts2
Out[106]:
2000-01-03   -0.759826
2000-01-06   -0.261202
2000-01-11   -0.725007

In [107]: ts2.reindex(ts.index)
Out[107]:
2000-01-03   -0.759826
2000-01-04         NaN
2000-01-05         NaN
2000-01-06   -0.261202
2000-01-07         NaN
2000-01-10         NaN
2000-01-11   -0.725007
2000-01-12         NaN

In [108]: ts2.reindex(ts.index, method='ffill')
Out[108]:
2000-01-03   -0.759826
2000-01-04   -0.759826
2000-01-05   -0.759826
2000-01-06   -0.261202
2000-01-07   -0.261202
2000-01-10   -0.261202
2000-01-11   -0.725007
2000-01-12   -0.725007

In [109]: ts2.reindex(ts.index, method='bfill')
Out[109]:
2000-01-03   -0.759826
2000-01-04   -0.261202
2000-01-05   -0.261202
2000-01-06   -0.261202
2000-01-07   -0.725007
2000-01-10   -0.725007
2000-01-11   -0.725007
2000-01-12         NaN
```

Note the same result could have been achieved using *fillna*:

```
In [110]: ts2.reindex(ts.index).fillna(method='ffill')
Out[110]:
2000-01-03   -0.759826
2000-01-04   -0.759826
2000-01-05   -0.759826
2000-01-06   -0.261202
2000-01-07   -0.261202
2000-01-10   -0.261202
2000-01-11   -0.725007
2000-01-12   -0.725007
```

Note these methods generally assume that the indexes are **sorted**. They may be modified in the future to be a bit more flexible but as time series data is ordered most of the time anyway, this has not been a major priority.

### 6.6.5 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 [111]: df
Out[111]:
        one     three       two
a -1.803562       NaN  0.921961
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
d       NaN  0.230059 -0.517650

In [112]: df.drop(['a', 'd'], axis=0)
Out[112]:
        one     three       two
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262

In [113]: df.drop(['one'], axis=1)
Out[113]:
      three       two
a       NaN  0.921961
b  2.304932  0.151528
c -0.926851  0.334262
d  0.230059 -0.517650
```

Note that the following also works, but a bit less obvious / clean:

```
In [114]: df.reindex(df.index - ['a', 'd'])
Out[114]:
        one     three       two
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
```

### 6.6.6 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 [115]: s
Out[115]:
a    1.125606
b   -0.426032
c   -0.061063
d   -0.644590
e    1.551388

In [116]: s.rename(str.upper)
Out[116]:
A    1.125606
B   -0.426032
C   -0.061063
D   -0.644590
E    1.551388
```

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 [117]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
   .....:           index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
Out[117]:
             foo     three       bar
apple  -1.803562       NaN  0.921961
```

```
banana -0.133357  2.304932  0.151528
c       0.917093 -0.926851  0.334262
durian       NaN  0.230059 -0.517650
```

The `rename` method also provides a `copy` named parameter that is by default `True` and copies the underlying data. Pass `copy=False` to rename the data in place. The Panel class has an a related `rename_axis` class which can rename any of its three axes.

## 6.7 Iteration

Considering the pandas as somewhat dict-like structure, basic iteration produces the "keys" of the objects, namely:

- **Series**: the index label
- **DataFrame**: the column labels
- **Panel**: the item labels

Thus, for example:

```
In [118]: for col in df:
   .....:     print col
   .....:
one
three
two
```

### 6.7.1 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 [119]: for item, frame in wp.iteritems():
   .....:     print item
   .....:     print frame
   .....:
Item1
                   A         B         C         D
2000-01-03  0.838258  1.607060 -1.711896 -0.215590
2000-01-04 -1.486802 -0.186982 -0.980778  1.642659
2000-01-05  0.764063  0.429431  1.249702  0.506673
2000-01-06  0.765594 -2.427144 -1.265159 -0.570580
2000-01-07  1.725629 -1.510635 -0.528086  0.559317
Item2
                   A         B         C         D
2000-01-03 -2.158684  0.451897  1.476465 -0.968314
2000-01-04 -0.054924  0.232306  0.703899 -0.149485
2000-01-05  0.235345 -0.654572  1.588949 -0.771629
2000-01-06  0.107966  2.488819  2.324974  0.687952
2000-01-07  0.061444 -0.316759 -0.263752 -0.090173
```

### 6.7.2 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 [120]: for row_index, row in df2.iterrows():
   .....:     print '%s\n%s' % (row_index, row)
   .....:
a
one   -1.463620
two    0.452711
Name: a
b
one    0.206585
two   -0.317723
Name: b
c
one    1.257035
two   -0.134988
Name: c
```

For instance, a contrived way to transpose the dataframe would be:

```
In [121]: df2 = DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [122]: print df2
   x  y
0  1  4
1  2  5
2  3  6

In [123]: print df2.T
   0  1  2
x  1  2  3
y  4  5  6

In [124]: df2_t = DataFrame(dict((idx,values) for idx, values in df2.iterrows()))

In [125]: print df2_t
   0  1  2
x  1  2  3
y  4  5  6
```

### 6.7.3 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 [126]: for r in df2.itertuples(): print r
(0, 1, 4)
(1, 2, 5)
(2, 3, 6)
```

## 6.8 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 [127]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....:                          columns=['three', 'two', 'one'])

In [128]: unsorted_df.sort_index()
Out[128]:
      three       two       one
a       NaN  0.921961 -1.803562
b  2.304932  0.151528 -0.133357
c -0.926851  0.334262  0.917093
d  0.230059 -0.517650       NaN

In [129]: unsorted_df.sort_index(ascending=False)
Out[129]:
      three       two       one
d  0.230059 -0.517650       NaN
c -0.926851  0.334262  0.917093
b  2.304932  0.151528 -0.133357
a       NaN  0.921961 -1.803562

In [130]: unsorted_df.sort_index(axis=1)
Out[130]:
        one     three       two
a -1.803562       NaN  0.921961
d       NaN  0.230059 -0.517650
c  0.917093 -0.926851  0.334262
b -0.133357  2.304932  0.151528
```

`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 [131]: df.sort_index(by='two')
Out[131]:
        one     three       two
d       NaN  0.230059 -0.517650
b -0.133357  2.304932  0.151528
c  0.917093 -0.926851  0.334262
a -1.803562       NaN  0.921961
```

The `by` argument can take a list of column names, e.g.:

```
In [132]: df = DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})

In [133]: df[['one', 'two', 'three']].sort_index(by=['one','two'])
Out[133]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

Series has the method `order` (analogous to R's order function) which sorts by value, with special treatment of NA values via the `na_last` argument:

```
In [134]: s[2] = np.nan

In [135]: s.order()
Out[135]:
d   -0.644590
b   -0.426032
a    1.125606
e    1.551388
c         NaN

In [136]: s.order(na_last=False)
Out[136]:
c         NaN
d   -0.644590
b   -0.426032
a    1.125606
e    1.551388
```

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`.

## 6.9 Copying, type casting

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.

Data can be explicitly cast to a NumPy dtype by using the `astype` method or alternately passing the `dtype` keyword argument to the object constructor.

```
In [137]: df = DataFrame(np.arange(12).reshape((4, 3)))

In [138]: df[0].dtype
Out[138]: dtype('int64')

In [139]: df.astype(float)[0].dtype
Out[139]: dtype('float64')

In [140]: df = DataFrame(np.arange(12).reshape((4, 3)), dtype=float)

In [141]: df[0].dtype
Out[141]: dtype('float64')
```

### 6.9.1 Inferring better types for object columns

The `convert_objects` DataFrame method will attempt to convert `dtype=object` columns to a better NumPy dtype. Occasionally (after transposing multiple times, for example), a mixed-type DataFrame will end up with everything as `dtype=object`. This method attempts to fix that:

```
In [142]: df = DataFrame(randn(6, 3), columns=['a', 'b', 'c'])

In [143]: df['d'] = 'foo'

In [144]: df
Out[144]:
          a         b         c    d
0 -0.455471  0.031559 -0.099764  foo
1 -0.700640 -1.481563  1.759569  foo
2  0.108328 -0.601951  0.540880  foo
3  0.035645  0.249108 -0.715238  foo
4  0.674059  1.026316 -0.983394  foo
5 -0.819144 -1.939507 -1.617302  foo

In [145]: df = df.T.T

In [146]: df.dtypes
Out[146]:
a    object
b    object
c    object
d    object

In [147]: converted = df.convert_objects()

In [148]: converted.dtypes
Out[148]:
a    float64
b    float64
c    float64
d     object
```

## 6.10 Pickling and serialization

All pandas objects are equipped with `save` methods which use Python's `cPickle` module to save data structures to disk using the pickle format.

```
In [149]: df
Out[149]:
            a           b           c    d
0  -0.4554712  0.03155879 -0.09976363  foo
1  -0.7006397   -1.481563    1.759569  foo
2   0.1083282  -0.6019514   0.5408803  foo
3  0.03564486   0.2491077  -0.7152381  foo
4   0.6740586    1.026316  -0.9833944  foo
5  -0.8191439   -1.939507   -1.617302  foo

In [150]: df.save('foo.pickle')
```

The `load` function in the `pandas` namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [151]: load('foo.pickle')
Out[151]:
           a           b           c    d
0  -0.4554712  0.03155879 -0.09976363  foo
1  -0.7006397   -1.481563    1.759569  foo
2   0.1083282  -0.6019514   0.5408803  foo
3  0.03564486   0.2491077  -0.7152381  foo
4   0.6740586    1.026316  -0.9833944  foo
5  -0.8191439   -1.939507   -1.617302  foo
```

There is also a `save` function which takes any object as its first argument:

```
In [152]: save(df, 'foo.pickle')

In [153]: load('foo.pickle')
Out[153]:
           a           b           c    d
0  -0.4554712  0.03155879 -0.09976363  foo
1  -0.7006397   -1.481563    1.759569  foo
2   0.1083282  -0.6019514   0.5408803  foo
3  0.03564486   0.2491077  -0.7152381  foo
4   0.6740586    1.026316  -0.9833944  foo
5  -0.8191439   -1.939507   -1.617302  foo
```

## 6.11 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 [154]: set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [155]: df['a']/1.e3
Out[155]:
0   -455.471u
1   -700.640u
2    108.328u
3     35.645u
4    674.059u
5   -819.144u
Name: a

In [156]: df['a']/1.e6
Out[156]:
0   -455.471n
1   -700.640n
2    108.328n
3     35.645n
4    674.059n
5   -819.144n
Name: a
```

The `set_printoptions` function has a number of options for controlling how floating point numbers are formatted (using hte `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.

CHAPTER

SEVEN

# INDEXING AND SELECTING DATA

The axis labeling information in pandas objects serves many purposes:

- Identifies data (i.e. provides *metadata*) using known indicators, important for for analysis, visualization, and interactive console display
- Enables automatic and explicit data alignment
- Allows intuitive getting and setting of subsets of the data set

In this section / chapter, 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.

## 7.1 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,

- **Series**: `series[label]` returns a scalar value
- **DataFrame**: `frame[colname]` returns a Series corresponding to the passed column name
- **Panel**: `panel[itemname]` returns a DataFrame corresponding to the passed item name

Here we construct a simple time series data set to use for illustrating the indexing functionality:

```
In [432]: dates = np.asarray(DateRange('1/1/2000', periods=8))

In [433]: df = DataFrame(randn(8, 4), index=dates, columns=['A', 'B', 'C', 'D'])

In [434]: df
Out[434]:
                   A         B         C         D
2000-01-03  0.469112 -0.282863 -1.509059 -1.135632
2000-01-04  1.212112 -0.173215  0.119209 -1.044236
2000-01-05 -0.861849 -2.104569 -0.494929  1.071804
2000-01-06  0.721555 -0.706771 -1.039575  0.271860
2000-01-07 -0.424972  0.567020  0.276232 -1.087401
2000-01-10 -0.673690  0.113648 -1.478427  0.524988
2000-01-11  0.404705  0.577046 -1.715002 -1.039268
2000-01-12 -0.370647 -1.157892 -1.344312  0.844885
```

```
In [435]: panel = Panel({'one' : df, 'two' : df - df.mean()})

In [436]: panel
Out[436]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major) x 4 (minor)
Items: one to two
Major axis: 2000-01-03 00:00:00 to 2000-01-12 00:00:00
Minor 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 [437]: s = df['A']

In [438]: s[dates[5]]
Out[438]: -0.67368970808837059

In [439]: panel['two']
Out[439]:
                   A         B         C         D
2000-01-03  0.409571  0.113086 -0.610826 -0.936507
2000-01-04  1.152571  0.222735  1.017442 -0.845111
2000-01-05 -0.921390 -1.708620  0.403304  1.270929
2000-01-06  0.662014 -0.310822 -0.141342  0.470985
2000-01-07 -0.484513  0.962970  1.174465 -0.888276
2000-01-10 -0.733231  0.509598 -0.580194  0.724113
2000-01-11  0.345164  0.972995 -0.816769 -0.840143
2000-01-12 -0.430188 -0.761943 -0.446079  1.044010
```

### 7.1.1 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 `get_value` method, which is implemented on all of the data structures:

```
In [440]: s.get_value(dates[5])
Out[440]: -0.67368970808837059

In [441]: df.get_value(dates[5], 'A')
Out[441]: -0.67368970808837059
```

There is an analogous `set_value` method which has the additional capability of enlarging an object. This method *always* returns a reference to the object it modified, which in the fast of enlargement, will be a **new object**:

```
In [442]: df.set_value(dates[5], 'E', 7)
Out[442]:
                   A         B         C         D    E
2000-01-03  0.469112 -0.282863 -1.509059 -1.135632  NaN
2000-01-04  1.212112 -0.173215  0.119209 -1.044236  NaN
2000-01-05 -0.861849 -2.104569 -0.494929  1.071804  NaN
2000-01-06  0.721555 -0.706771 -1.039575  0.271860  NaN
2000-01-07 -0.424972  0.567020  0.276232 -1.087401  NaN
2000-01-10 -0.673690  0.113648 -1.478427  0.524988    7
```

```
2000-01-11  0.404705  0.577046 -1.715002 -1.039268 NaN
2000-01-12 -0.370647 -1.157892 -1.344312  0.844885 NaN
```

### 7.1.2 Additional Column Access

You may access a column on a dataframe directly as an attribute:

```
In [443]: df.A
Out[443]:
2000-01-03    0.469112
2000-01-04    1.212112
2000-01-05   -0.861849
2000-01-06    0.721555
2000-01-07   -0.424972
2000-01-10   -0.673690
2000-01-11    0.404705
2000-01-12   -0.370647
Name: A
```

If you are using the IPython environment, you may also use tab-completion to see the accessible columns of a DataFrame.

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 [444]: df
Out[444]:
                   A         B         C         D
2000-01-03  0.469112 -0.282863 -1.509059 -1.135632
2000-01-04  1.212112 -0.173215  0.119209 -1.044236
2000-01-05 -0.861849 -2.104569 -0.494929  1.071804
2000-01-06  0.721555 -0.706771 -1.039575  0.271860
2000-01-07 -0.424972  0.567020  0.276232 -1.087401
2000-01-10 -0.673690  0.113648 -1.478427  0.524988
2000-01-11  0.404705  0.577046 -1.715002 -1.039268
2000-01-12 -0.370647 -1.157892 -1.344312  0.844885

In [445]: df[['B', 'A']] = df[['A', 'B']]

In [446]: df
Out[446]:
                   A         B         C         D
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804
2000-01-06 -0.706771  0.721555 -1.039575  0.271860
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
2000-01-10  0.113648 -0.673690 -1.478427  0.524988
2000-01-11  0.577046  0.404705 -1.715002 -1.039268
2000-01-12 -1.157892 -0.370647 -1.344312  0.844885
```

You may find this useful for applying a transform (in-place) to a subset of the columns.

### 7.1.3 Data slices on other axes

It's certainly possible to retrieve data slices along the other axes of a DataFrame or Panel. We tend to refer to these slices as *cross-sections*. DataFrame has the `xs` function for retrieving rows as Series and Panel has the analogous major_xs and minor_xs functions for retrieving slices as DataFrames for a given major_axis or minor_axis label, respectively.

```
In [447]: date = dates[5]

In [448]: df.xs(date)
Out[448]:
A    0.113648
B   -0.673690
C   -1.478427
D    0.524988
Name: 2000-01-10 00:00:00

In [449]: panel.major_xs(date)
Out[449]:
        one       two
A -0.673690 -0.733231
B  0.113648  0.509598
C -1.478427 -0.580194
D  0.524988  0.724113

In [450]: panel.minor_xs('A')
Out[450]:
                 one       two
2000-01-03  0.469112  0.409571
2000-01-04  1.212112  1.152571
2000-01-05 -0.861849 -0.921390
2000-01-06  0.721555  0.662014
2000-01-07 -0.424972 -0.484513
2000-01-10 -0.673690 -0.733231
2000-01-11  0.404705  0.345164
2000-01-12 -0.370647 -0.430188
```

### 7.1.4 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the *Advanced indexing* section detailing the .ix 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 [451]: s[:5]
Out[451]:
2000-01-03   -0.282863
2000-01-04   -0.173215
2000-01-05   -2.104569
2000-01-06   -0.706771
2000-01-07    0.567020
Name: A

In [452]: s[::2]
Out[452]:
2000-01-03   -0.282863
2000-01-05   -2.104569
2000-01-07    0.567020
2000-01-11    0.577046
Name: A

In [453]: s[::-1]
```

```
Out[453]:
2000-01-12   -1.157892
2000-01-11    0.577046
2000-01-10    0.113648
2000-01-07    0.567020
2000-01-06   -0.706771
2000-01-05   -2.104569
2000-01-04   -0.173215
2000-01-03   -0.282863
Name: A
```

Note that setting works as well:

```
In [454]: s2 = s.copy()

In [455]: s2[:5] = 0

In [456]: s2
Out[456]:
2000-01-03    0.000000
2000-01-04    0.000000
2000-01-05    0.000000
2000-01-06    0.000000
2000-01-07    0.000000
2000-01-10    0.113648
2000-01-11    0.577046
2000-01-12   -1.157892
Name: A
```

With DataFrame, slicing inside of `[]` **slices the rows**. This is provided largely as a convenience since it is such a common operation.

```
In [457]: df[:3]
Out[457]:
                   A         B         C         D
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804

In [458]: df[::-1]
Out[458]:
                   A         B         C         D
2000-01-12 -1.157892 -0.370647 -1.344312  0.844885
2000-01-11  0.577046  0.404705 -1.715002 -1.039268
2000-01-10  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
2000-01-06 -0.706771  0.721555 -1.039575  0.271860
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632
```

### 7.1.5 Boolean indexing

Using a boolean vector to index a Series works exactly as in a numpy ndarray:

```
In [459]: s[s > 0]
Out[459]:
2000-01-07    0.567020
```

```
2000-01-10    0.113648
2000-01-11    0.577046
Name: A

In [460]: s[(s < 0) & (s > -0.5)]
Out[460]:
2000-01-03   -0.282863
2000-01-04   -0.173215
Name: A
```

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 [461]: df[df['A'] > 0]
Out[461]:
                   A         B         C         D
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
2000-01-10  0.113648 -0.673690 -1.478427  0.524988
2000-01-11  0.577046  0.404705 -1.715002 -1.039268
```

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 out rows where one or more columns have values you want:

```
In [462]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   .....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                  'c' : np.random.randn(7)})

In [463]: df2[df2['a'].isin(['one', 'two'])]
Out[463]:
     a  b         c
0  one  x  1.075770
1  one  y -0.109050
2  two  y  1.643563
4  two  y  0.357021
5  one  x -0.674600
```

Note, with the *advanced indexing* `ix` method, you may select along more than one axis using boolean vectors combined with other indexing expressions.

## 7.1.6 Indexing a DataFrame with a boolean DataFrame

You may wish to set values on a DataFrame based on some boolean criteria derived from itself or another DataFrame or set of DataFrames. This can be done intuitively like so:

```
In [464]: df2 = df.copy()

In [465]: df2 < 0
Out[465]:
                A      B      C      D
2000-01-03   True  False   True   True
2000-01-04   True  False  False   True
2000-01-05   True   True   True  False
2000-01-06   True  False   True  False
2000-01-07  False   True  False   True
2000-01-10  False   True   True  False
2000-01-11  False  False   True   True
2000-01-12   True   True   True  False
```

```
In [466]: df2[df2 < 0] = 0

In [467]: df2
Out[467]:
                   A         B         C         D
2000-01-03  0.000000  0.469112  0.000000  0.000000
2000-01-04  0.000000  1.212112  0.119209  0.000000
2000-01-05  0.000000  0.000000  0.000000  1.071804
2000-01-06  0.000000  0.721555  0.000000  0.271860
2000-01-07  0.567020  0.000000  0.276232  0.000000
2000-01-10  0.113648  0.000000  0.000000  0.524988
2000-01-11  0.577046  0.404705  0.000000  0.000000
2000-01-12  0.000000  0.000000  0.000000  0.844885
```

Note that such an operation requires that the boolean DataFrame is indexed exactly the same.

### 7.1.7 Take Methods

TODO: Fill Me In

### 7.1.8 Duplicate Data

If you want to indentify 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 [468]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   .....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                  'c' : np.random.randn(7)})

In [469]: df2.duplicated(['a','b'])
Out[469]:
0    False
1    False
2    False
3    False
4     True
5     True
6    False

In [470]: df2.drop_duplicates(['a','b'])
Out[470]:
       a  b         c
0    one  x -0.968914
1    one  y -1.294524
2    two  y  0.413738
3  three  x  0.276662
6    six  x -0.362543

In [471]: df2.drop_duplicates(['a','b'], take_last=True)
```

```
Out[471]:
       a  b         c
1    one  y -1.294524
3  three  x  0.276662
4    two  y -0.472035
5    one  x -0.013960
6    six  x -0.362543
```

### 7.1.9 Dictionary-like get method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.

```
In [472]: s = Series([1,2,3], index=['a','b','c'])

In [473]: s.get('a')               # equivalent to s['a']
Out[473]: 1

In [474]: s.get('x', default=-1)
Out[474]: -1
```

## 7.2 Advanced indexing with labels

We have avoided excessively overloading the `[]` / `__getitem__` operator to keep the basic functionality of the pandas objects straightforward and simple. However, there are often times when you may wish get a subset (or analogously set a subset) of the data in a way that is not straightforward using the combination of `reindex` and `[]`. Complicated setting operations are actually quite difficult because `reindex` usually returns a copy.

By *advanced* indexing we are referring to a special `.ix` attribute on pandas objects which enable you to do getting/setting operations on a DataFrame, for example, with matrix/ndarray-like semantics. Thus you can combine the following kinds of indexing:

- 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 [475]: subindex = dates[[3,4,5]]

In [476]: df.reindex(index=subindex, columns=['C', 'B'])
Out[476]:
                   C         B
2000-01-06 -1.039575  0.721555
2000-01-07  0.276232 -0.424972
2000-01-10 -1.478427 -0.673690

In [477]: df.ix[subindex, ['C', 'B']]
Out[477]:
                   C         B
2000-01-06 -1.039575  0.721555
2000-01-07  0.276232 -0.424972
2000-01-10 -1.478427 -0.673690
```

Assignment / setting values is possible when using `ix`:

```
In [478]: df2 = df.copy()

In [479]: df2.ix[subindex, ['C', 'B']] = 0

In [480]: df2
Out[480]:
                   A         B         C         D
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804
2000-01-06 -0.706771  0.000000  0.000000  0.271860
2000-01-07  0.567020  0.000000  0.000000 -1.087401
2000-01-10  0.113648  0.000000  0.000000  0.524988
2000-01-11  0.577046  0.404705 -1.715002 -1.039268
2000-01-12 -1.157892 -0.370647 -1.344312  0.844885
```

Indexing with an array of integers can also be done:

```
In [481]: df.ix[[4,3,1]]
Out[481]:
                   A         B         C         D
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
2000-01-06 -0.706771  0.721555 -1.039575  0.271860
2000-01-04 -0.173215  1.212112  0.119209 -1.044236

In [482]: df.ix[dates[[4,3,1]]]
Out[482]:
                   A         B         C         D
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
2000-01-06 -0.706771  0.721555 -1.039575  0.271860
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
```

**Slicing** has standard Python semantics for integer slices:

```
In [483]: df.ix[1:7, :2]
Out[483]:
                   A         B
2000-01-04 -0.173215  1.212112
2000-01-05 -2.104569 -0.861849
2000-01-06 -0.706771  0.721555
2000-01-07  0.567020 -0.424972
2000-01-10  0.113648 -0.673690
2000-01-11  0.577046  0.404705
```

Slicing with labels is semantically slightly different because the slice start and stop are **inclusive** in the label-based case:

```
In [484]: date1, date2 = dates[[2, 4]]

In [485]: print date1, date2
2000-01-05 00:00:00 2000-01-07 00:00:00

In [486]: df.ix[date1:date2]
Out[486]:
                   A         B         C         D
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804
2000-01-06 -0.706771  0.721555 -1.039575  0.271860
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
```

```
In [487]: df['A'].ix[date1:date2]
Out[487]:
2000-01-05   -2.104569
2000-01-06   -0.706771
2000-01-07    0.567020
Name: A
```

Getting and setting rows in a DataFrame, especially by their location, is much easier:

```
In [488]: df2 = df[:5].copy()

In [489]: df2.ix[3]
Out[489]:
A   -0.706771
B    0.721555
C   -1.039575
D    0.271860
Name: 2000-01-06 00:00:00

In [490]: df2.ix[3] = np.arange(len(df2.columns))

In [491]: df2
Out[491]:
                   A         B         C         D
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632
2000-01-04 -0.173215  1.212112  0.119209 -1.044236
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804
2000-01-06  0.000000  1.000000  2.000000  3.000000
2000-01-07  0.567020 -0.424972  0.276232 -1.087401
```

Column or row selection can be combined as you would expect with arrays of labels or even boolean vectors:

```
In [492]: df.ix[df['A'] > 0, 'B']
Out[492]:
2000-01-07   -0.424972
2000-01-10   -0.673690
2000-01-11    0.404705
Name: B

In [493]: df.ix[date1:date2, 'B']
Out[493]:
2000-01-05   -0.861849
2000-01-06    0.721555
2000-01-07   -0.424972
Name: B

In [494]: df.ix[date1, 'B']
Out[494]: -0.86184896334779992
```

Slicing with labels is closely related to the `truncate` method which does precisely `.ix[start:stop]` but returns a copy (for legacy reasons).

## 7.2.1 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.

### 7.2.2 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 [495]: df.select(lambda x: x == 'A', axis=1)
Out[495]:
                   A
2000-01-03 -0.282863
2000-01-04 -0.173215
2000-01-05 -2.104569
2000-01-06 -0.706771
2000-01-07  0.567020
2000-01-10  0.113648
2000-01-11  0.577046
2000-01-12 -1.157892
```

### 7.2.3 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 [496]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])

In [497]: dflookup.lookup(xrange(0,10,2), ['B','C','A','B','D'])
Out[497]: array([ 0.4973,  0.9423,  0.8626,  0.6341,  0.5629])
```

### 7.2.4 Advanced indexing with integer labels

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, advanced indexing with `.ix` will always attempt label-based indexing, before falling back on integer-based indexing.

### 7.2.5 Setting values in mixed-type DataFrame

Setting values on a mixed-type DataFrame or Panel is supported when using scalar values, though setting arbitrary vectors is not yet supported:

```
In [498]: df2 = df[:4]

In [499]: df2['foo'] = 'bar'

In [500]: print df2
                   A         B         C         D  foo
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632  bar
2000-01-04 -0.173215  1.212112  0.119209 -1.044236  bar
2000-01-05 -2.104569 -0.861849 -0.494929  1.071804  bar
2000-01-06 -0.706771  0.721555 -1.039575  0.271860  bar

In [501]: df2.ix[2] = np.nan

In [502]: print df2
```

```
                   A         B         C         D  foo
2000-01-03 -0.282863  0.469112 -1.509059 -1.135632  bar
2000-01-04 -0.173215  1.212112  0.119209 -1.044236  bar
2000-01-05       NaN       NaN       NaN       NaN  NaN
2000-01-06 -0.706771  0.721555 -1.039575  0.271860  bar

In [503]: print df2.dtypes
A      float64
B      float64
C      float64
D      float64
foo     object
```

## 7.3 Index objects

The pandas Index class and its subclasses can be viewed as implementing an *ordered set* in addition to providing the support infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create one directly is to pass a list or other sequence to `Index`:

```
In [504]: index = Index(['e', 'd', 'a', 'b'])

In [505]: index
Out[505]: Index([e, d, a, b], dtype=object)

In [506]: 'd' in index
Out[506]: True
```

You can also pass a `name` to be stored in the index:

```
In [507]: index = Index(['e', 'd', 'a', 'b'], name='something')

In [508]: index.name
Out[508]: 'something'
```

Starting with pandas 0.5, the name, if set, will be shown in the console display:

```
In [509]: index = Index(range(5), name='rows')

In [510]: columns = Index(['A', 'B', 'C'], name='cols')

In [511]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [512]: df
Out[512]:
cols         A         B         C
rows
0     3.357427 -0.317441 -1.236269
1     0.896171 -0.487602 -0.082240
2    -2.182937  0.380396  0.084844
3     0.432390  1.519970 -0.493662
4     0.600178  0.274230  0.132885

In [513]: df['A']
Out[513]:
rows
0       3.357427
```

```
1       0.896171
2      -2.182937
3       0.432390
4       0.600178
Name: A
```

### 7.3.1 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 [514]: a = Index(['c', 'b', 'a'])

In [515]: b = Index(['c', 'e', 'd'])

In [516]: a.union(b)
Out[516]: Index([a, b, c, d, e], dtype=object)

In [517]: a | b
Out[517]: Index([a, b, c, d, e], dtype=object)

In [518]: a & b
Out[518]: Index([c], dtype=object)

In [519]: a - b
Out[519]: Index([a, b], dtype=object)
```

### 7.3.2 `isin` method of Index objects

One additional operation is the `isin` method that works analogously to the `Series.isin` method found *here*.

## 7.4 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 effectively store and manipulate arbitrarily high dimension data in a 2-dimensional tabular structure (DataFrame), for example. It is not limited to DataFrame

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.

**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.

### 7.4.1 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 [520]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   .....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]

In [521]: tuples = zip(*arrays)

In [522]: tuples
Out[522]:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [523]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [524]: s = Series(randn(8), index=index)

In [525]: s
Out[525]:
first  second
bar    one      -0.023688
       two       2.410179
baz    one       1.450520
       two       0.206053
foo    one      -0.251905
       two      -2.213588
qux    one       1.063327
       two       1.266143
```

All of the `MultiIndex` constructors accept a `names` argument which stores string names for the levels themselves. If no names are provided, some arbitrary ones will be assigned:

```
In [526]: index.names
Out[526]: ['first', 'second']
```

This index can back any axis of a pandas object, and the number of **levels** of the index is up to you:

```
In [527]: df = DataFrame(randn(3, 8), index=['A', 'B', 'C'], columns=index)

In [528]: df
Out[528]:
first        bar                 baz                 foo                 qux
second       one       two       one       two       one       two       one       two
A       0.299368 -0.863838  0.408204 -1.048089 -0.025747 -0.988387  0.094055  1.262731
B       1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374 -0.034571 -2.484478
C      -0.281461  0.030711  0.109121  1.126203 -0.977349  1.474071 -0.064034 -1.282782

In [529]: DataFrame(randn(6, 6), index=index[:6], columns=index[:6])
Out[529]:
```

```
first                bar                 baz                 foo
second               one       two       one       two       one       two
first second
bar   one       0.781836 -1.071357  0.441153  2.353925  0.583787  0.221471
      two      -0.744471  0.758527  1.729689 -0.964980 -0.845696 -1.340896
baz   one       1.846883 -1.328865  1.682706 -1.717693  0.888782  0.228440
      two       0.901805  1.171216  0.520260 -1.197071 -1.066969 -0.303421
foo   one      -0.858447  0.306996 -0.028665  0.384316  1.574159  1.588931
      two       0.476720  0.473424 -0.242861 -0.014805 -0.284319  0.650776
```

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 [530]: Series(randn(8), index=tuples)
Out[530]:
('bar', 'one')   -1.461665
('bar', 'two')   -1.137707
('baz', 'one')   -0.891060
('baz', 'two')   -0.693921
('foo', 'one')    1.613616
('foo', 'two')    0.464000
('qux', 'one')    0.227371
('qux', 'two')   -0.496922
```

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.

### 7.4.2 Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

```
In [531]: index.get_level_values(0)
Out[531]: array([bar, bar, baz, baz, foo, foo, qux, qux], dtype=object)

In [532]: index.get_level_values('second')
Out[532]: array([one, two, one, two, one, two, one, two], dtype=object)
```

### 7.4.3 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 [533]: df['bar']
Out[533]:
second        one       two
A        0.299368 -0.863838
B        1.289997  0.082423
C       -0.281461  0.030711

In [534]: df['bar', 'one']
Out[534]:
A    0.299368
```

```
B    1.289997
C   -0.281461
Name: ('bar', 'one')

In [535]: df['bar']['one']
Out[535]:
A    0.299368
B    1.289997
C   -0.281461
Name: one

In [536]: s['qux']
Out[536]:
second
one       1.063327
two       1.266143
```

### 7.4.4 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 [537]: s + s[:-2]
Out[537]:
first  second
bar    one      -0.047377
       two       4.820357
baz    one       2.901041
       two       0.412107
foo    one      -0.503810
       two      -4.427175
qux    one            NaN
       two            NaN

In [538]: s + s[::2]
Out[538]:
first  second
bar    one      -0.047377
       two            NaN
baz    one       2.901041
       two            NaN
foo    one      -0.503810
       two            NaN
qux    one       2.126655
       two            NaN
```

`reindex` can be called with another `MultiIndex` or even a list or array of tuples:

```
In [539]: s.reindex(index[:3])
Out[539]:
first  second
bar    one      -0.023688
       two       2.410179
baz    one       1.450520

In [540]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
Out[540]:
```

```
first  second
foo    two       -2.213588
bar    one       -0.023688
qux    one        1.063327
baz    one        1.450520
```

### 7.4.5 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 [541]: df = df.T

In [542]: df
Out[542]:
                     A         B         C
first second
bar   one     0.299368  1.289997 -0.281461
      two    -0.863838  0.082423  0.030711
baz   one     0.408204 -0.055758  0.109121
      two    -1.048089  0.536580  1.126203
foo   one    -0.025747 -0.489682 -0.977349
      two    -0.988387  0.369374  1.474071
qux   one     0.094055 -0.034571 -0.064034
      two     1.262731 -2.484478 -1.282782

In [543]: df.ix['bar']
Out[543]:
               A         B         C
second
one     0.299368  1.289997 -0.281461
two    -0.863838  0.082423  0.030711

In [544]: df.ix['bar', 'two']
Out[544]:
A   -0.863838
B    0.082423
C    0.030711
Name: ('bar', 'two')
```

"Partial" slicing also works quite nicely:

```
In [545]: df.ix['baz':'foo']
Out[545]:
                     A         B         C
first second
baz   one     0.408204 -0.055758  0.109121
      two    -1.048089  0.536580  1.126203
foo   one    -0.025747 -0.489682 -0.977349
      two    -0.988387  0.369374  1.474071

In [546]: df.ix[('baz', 'two'):('qux', 'one')]
Out[546]:
                     A         B         C
first second
baz   two    -1.048089  0.536580  1.126203
foo   one    -0.025747 -0.489682 -0.977349
```

```
      two    -0.988387  0.369374  1.474071
qux   one     0.094055 -0.034571 -0.064034

In [547]: df.ix[('baz', 'two'):'foo']
Out[547]:
                    A         B         C
first second
baz   two    -1.048089  0.536580  1.126203
foo   one    -0.025747 -0.489682 -0.977349
      two    -0.988387  0.369374  1.474071
```

Passing a list of labels or tuples works similar to reindexing:

```
In [548]: df.ix[[('bar', 'two'), ('qux', 'one')]]
Out[548]:
                    A         B         C
first second
bar   two    -0.863838  0.082423  0.030711
qux   one     0.094055 -0.034571 -0.064034
```

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.

### 7.4.6 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 [549]: df.xs('one', level='second')
Out[549]:
              A         B         C
first
bar    0.299368  1.289997 -0.281461
baz    0.408204 -0.055758  0.109121
foo   -0.025747 -0.489682 -0.977349
qux    0.094055 -0.034571 -0.064034
```

### 7.4.7 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 [550]: midx = MultiIndex(levels=[['zero', 'one'], ['x','y']],
   .....:                  labels=[[1,1,0,0],[1,0,1,0]])

In [551]: df = DataFrame(randn(4,2), index=midx)

In [552]: print df
               0         1
one  y  0.306389 -2.290613
     x -1.134623 -1.561819
```

```
zero y -0.260838  0.281957
     x  1.523962 -0.902937

In [553]: df2 = df.mean(level=0)

In [554]: print df2
              0         1
key_0
zero   0.631562 -0.310490
one   -0.414117 -1.926216

In [555]: print df2.reindex(df.index, level=0)
               0         1
one  y -0.414117 -1.926216
     x -0.414117 -1.926216
zero y  0.631562 -0.310490
     x  0.631562 -0.310490

In [556]: df_aligned, df2_aligned = df.align(df2, level=0)

In [557]: print df_aligned
               0         1
one  y  0.306389 -2.290613
     x -1.134623 -1.561819
zero y -0.260838  0.281957
     x  1.523962 -0.902937

In [558]: print df2_aligned
               0         1
one  y -0.414117 -1.926216
     x -0.414117 -1.926216
zero y  0.631562 -0.310490
     x  0.631562 -0.310490
```

### 7.4.8 The need for sortedness

**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 [559]: import random; random.shuffle(tuples)

In [560]: s = Series(randn(8), index=MultiIndex.from_tuples(tuples))

In [561]: s
Out[561]:
foo  two    0.068159
bar  one   -0.057873
     two   -0.368204
foo  one   -1.144073
qux  one    0.861209
baz  one    0.800193
```

```
     two    0.782098
qux  two   -1.069094

In [562]: s.sortlevel(0)
Out[562]:
bar  one   -0.057873
     two   -0.368204
baz  one    0.800193
     two    0.782098
foo  one   -1.144073
     two    0.068159
qux  one    0.861209
     two   -1.069094

In [563]: s.sortlevel(1)
Out[563]:
bar  one   -0.057873
baz  one    0.800193
foo  one   -1.144073
qux  one    0.861209
bar  two   -0.368204
baz  two    0.782098
foo  two    0.068159
qux  two   -1.069094
```

Note, you may also pass a level name to `sortlevel` if the MultiIndex levels are named.

```
In [564]: s.index.names = ['L1', 'L2']

In [565]: s.sortlevel(level='L1')
Out[565]:
L1   L2
bar  one   -0.057873
     two   -0.368204
baz  one    0.800193
     two    0.782098
foo  one   -1.144073
     two    0.068159
qux  one    0.861209
     two   -1.069094

In [566]: s.sortlevel(level='L2')
Out[566]:
L1   L2
bar  one   -0.057873
baz  one    0.800193
foo  one   -1.144073
qux  one    0.861209
bar  two   -0.368204
baz  two    0.782098
foo  two    0.068159
qux  two   -1.069094
```

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 [567]: s['qux']
Out[567]:
L2
```

```
one    0.861209
two   -1.069094

In [568]: s.sortlevel(1)['qux']
Out[568]:
L2
one    0.861209
two   -1.069094
```

On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

```
In [569]: df.T.sortlevel(1, axis=1)
Out[569]:
       zero       one      zero       one
          x         x         y         y
0  1.523962 -1.134623 -0.260838  0.306389
1 -0.902937 -1.561819  0.281957 -2.290613
```

The `MultiIndex` object has code to **explicity 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 [570]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]

In [571]: idx = MultiIndex.from_tuples(tuples)

In [572]: idx.lexsort_depth
Out[572]: 2

In [573]: reordered = idx[[1, 0, 3, 2]]

In [574]: reordered.lexsort_depth
Out[574]: 1

In [575]: s = Series(randn(4), index=reordered)

In [576]: s.ix['a':'a']
Out[576]:
a  b   -1.099248
   a    0.255269
```

However:

```
>>> s.ix[('a', 'b'):('b', 'a')]
Exception: MultiIndex lexsort depth 1, key was length 2
```

## 7.4.9 Swapping levels with `swaplevel`

The `swaplevel` function can switch the order of two levels:

```
In [577]: df[:5]
Out[577]:
               0         1
one  y  0.306389 -2.290613
     x -1.134623 -1.561819
zero y -0.260838  0.281957
     x  1.523962 -0.902937

In [578]: df[:5].swaplevel(0, 1, axis=0)
```

```
Out[578]:
               0         1
y one   0.306389 -2.290613
x one  -1.134623 -1.561819
y zero -0.260838  0.281957
x zero  1.523962 -0.902937
```

### 7.4.10 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 [579]: df[:5].reorder_levels([1,0], axis=0)
Out[579]:
               0         1
y one   0.306389 -2.290613
x one  -1.134623 -1.561819
y zero -0.260838  0.281957
x zero  1.523962 -0.902937
```

### 7.4.11 Some gory internal details

Internally, the `MultiIndex` consists of a few things: the **levels**, the integer **labels**, and the level **names**:

```
In [580]: index
Out[580]:
MultiIndex([('bar', 'one'), ('bar', 'two'), ('baz', 'one'), ('baz', 'two'),
       ('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')], dtype=object)

In [581]: index.levels
Out[581]: [Index([bar, baz, foo, qux], dtype=object), Index([one, two], dtype=object)]

In [582]: index.labels
Out[582]:
[array([0, 0, 1, 1, 2, 2, 3, 3], dtype=int32),
 array([0, 1, 0, 1, 0, 1, 0, 1], dtype=int32)]

In [583]: index.names
Out[583]: ['first', 'second']
```

You can probably guess that the labels determine which unique element is identified with that location at each layer of the index. It's important to note that sortedness is determined **solely** from the integer labels and does not check (or care) whether the levels themselves are sorted. Fortunately, the constructors `from_tuples` and `from_arrays` ensure that this is true, but if you compute the levels and labels yourself, please be careful.

## 7.5 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.

### 7.5.1 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 [584]: data
Out[584]:
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4

In [585]: indexed1 = data.set_index('c')

In [586]: indexed1
Out[586]:
     a    b  d
c
z  bar  one  1
y  bar  two  2
x  foo  one  3
w  foo  two  4

In [587]: indexed2 = data.set_index(['a', 'b'])

In [588]: indexed2
Out[588]:
         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
    two  w  4
```

Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):

```
In [589]: data.set_index('c', drop=False)
Out[589]:
     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

In [590]: df = data.set_index(['a', 'b'], inplace=True)

In [591]: data
Out[591]:
         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
    two  w  4
```

### 7.5.2 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 [592]: df
Out[592]:
         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
    two  w  4

In [593]: df.reset_index()
Out[593]:
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4
```

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the `names` attribute.

`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.

### 7.5.3 Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
df.index = index
```

## 7.6 Indexing internal details

**Note:** The following is largely relevant for those actually working on the pandas codebase. And 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
- `DateRange`: 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. For example, we plan to add a more efficient datetime index which leverages the new `numpy.datetime64` dtype in the relatively near future.

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
- `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

CHAPTER

EIGHT

# COMPUTATIONAL TOOLS

## 8.1 Statistical functions

### 8.1.1 Covariance

The `Series` object has a method `cov` to compute covariance between series (excluding NA/null values).

```
In [157]: s1 = Series(randn(1000))

In [158]: s2 = Series(randn(1000))

In [159]: s1.cov(s2)
Out[159]: 0.019465636696791695
```

Analogously, `DataFrame` has a method `cov` to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

```
In [160]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [161]: frame.cov()
Out[161]:
          a         b         c         d         e
a  0.953751 -0.029550 -0.006415  0.001020 -0.004134
b -0.029550  0.997223 -0.044276  0.005967  0.044884
c -0.006415 -0.044276  1.050236  0.077775  0.010642
d  0.001020  0.005967  0.077775  0.998485 -0.007345
e -0.004134  0.044884  0.010642 -0.007345  1.025446
```

### 8.1.2 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 [162]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [163]: frame.ix[::2] = np.nan
```

```
# Series with Series
In [164]: frame['a'].corr(frame['b'])
Out[164]: 0.013306883832198543

In [165]: frame['a'].corr(frame['b'], method='spearman')
Out[165]: 0.022530330121320486

# Pairwise correlation of DataFrame columns
In [166]: frame.corr()
Out[166]:
          a         b         c         d         e
a  1.000000  0.013307 -0.037801 -0.021905  0.001165
b  0.013307  1.000000 -0.017259  0.079246 -0.043606
c -0.037801 -0.017259  1.000000  0.061657  0.078945
d -0.021905  0.079246  0.061657  1.000000 -0.036978
e  0.001165 -0.043606  0.078945 -0.036978  1.000000
```

Note that non-numeric columns will be automatically excluded from the correlation calculation.

A related method `corrwith` is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

```
In [167]: index = ['a', 'b', 'c', 'd', 'e']

In [168]: columns = ['one', 'two', 'three', 'four']

In [169]: df1 = DataFrame(randn(5, 4), index=index, columns=columns)

In [170]: df2 = DataFrame(randn(4, 4), index=index[:4], columns=columns)

In [171]: df1.corrwith(df2)
Out[171]:
one      0.344149
two      0.837438
three    0.458904
four     0.712401

In [172]: df2.corrwith(df1, axis=1)
Out[172]:
a    0.404019
b    0.772204
c    0.420390
d   -0.142959
e         NaN
```

### 8.1.3 Data ranking

The `rank` method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```
In [173]: s = Series(np.random.randn(5), index=list('abcde'))

In [174]: s['d'] = s['b'] # so there's a tie

In [175]: s.rank()
Out[175]:
a    2.0
b    3.5
```

```
c    1.0
d    3.5
e    5.0
```

`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 [176]: df = DataFrame(np.random.randn(10, 6))

In [177]: df[4] = df[2][:5] # some ties

In [178]: df
Out[178]:
          0         1         2         3         4         5
0  0.106333  0.712162 -0.351275  1.176287 -0.351275  1.741787
1 -1.301869  0.612432 -0.577677  0.124709 -0.577677 -1.068084
2 -0.899627  0.822023  1.506319  0.998896  1.506319  0.259080
3 -0.522705 -1.473680 -1.726800  1.555343 -1.726800 -1.411978
4  0.733147  0.415881 -0.026973  0.999488 -0.026973  0.082219
5  0.995001 -1.399355  0.082244 -1.521795       NaN  0.416180
6 -0.779714 -0.226893  0.956567 -0.443664       NaN -0.610675
7 -0.635495 -0.621647  0.406259 -0.279002       NaN -1.153000
8  0.085011 -0.459422 -1.660917 -1.913019       NaN  0.833479
9 -0.557052  0.775425  0.003794  0.555351       NaN -1.169977

In [179]: df.rank(1)
Out[179]:
   0  1    2  3    4  5
0  3  4  1.5  5  1.5  6
1  1  6  3.5  5  3.5  2
2  1  3  5.5  4  5.5  2
3  5  3  1.5  6  1.5  4
4  5  4  1.5  6  1.5  3
5  5  2  3.0  1  NaN  4
6  1  4  5.0  3  NaN  2
7  2  3  5.0  4  NaN  1
8  4  3  2.0  1  NaN  5
9  2  5  3.0  4  NaN  1
```

`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

**Note:** These methods are significantly faster (around 10-20x) than `scipy.stats.rankdata`.

## 8.2 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`.

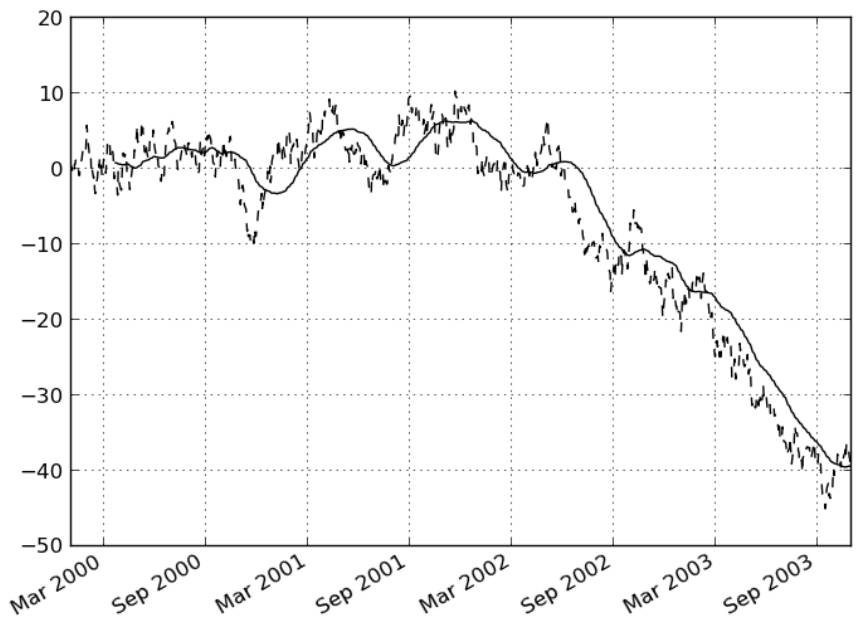
| Function | Description |
|---|---|
| `rolling_count` | Number of non-null observations |
| `rolling_sum` | Sum of values |
| `rolling_mean` | Mean of values |
| `rolling_median` | Arithmetic median of values |
| `rolling_min` | Minimum |
| `rolling_max` | Maximum |
| `rolling_std` | Unbiased standard deviation |
| `rolling_var` | Unbiased variance |
| `rolling_skew` | Unbiased skewness (3rd moment) |
| `rolling_kurt` | Unbiased kurtosis (4th moment) |
| `rolling_quantile` | Sample quantile (value at %) |
| `rolling_apply` | Generic apply |
| `rolling_cov` | Unbiased covariance (binary) |
| `rolling_corr` | Correlation (binary) |
| `rolling_corr_pairwise` | Pairwise correlation of DataFrame columns |

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)
- `time_rule`: optionally specify a *time rule* to pre-conform the data to

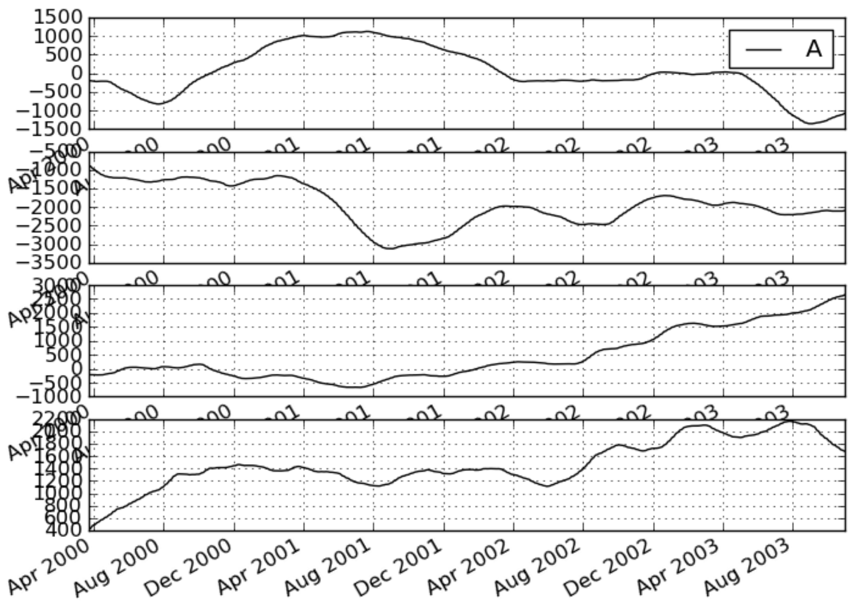
These functions can be applied to ndarrays or Series objects:

```
In [180]: ts = Series(randn(1000), index=DateRange('1/1/2000', periods=1000))

In [181]: ts = ts.cumsum()

In [182]: ts.plot(style='k--')
Out[182]: <matplotlib.axes.AxesSubplot at 0x10a7bd6d0>

In [183]: rolling_mean(ts, 60).plot(style='k')
Out[183]: <matplotlib.axes.AxesSubplot at 0x10a7bd6d0>
```



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 [184]: df = DataFrame(randn(1000, 4), index=ts.index,
   .....:                    columns=['A', 'B', 'C', 'D'])

In [185]: df = df.cumsum()

In [186]: rolling_sum(df, 60).plot(subplots=True)
Out[186]:
array([Axes(0.125,0.747826;0.775x0.152174),
       Axes(0.125,0.565217;0.775x0.152174),
       Axes(0.125,0.382609;0.775x0.152174), Axes(0.125,0.2;0.775x0.152174)], dtype=object)
```



### 8.2.1 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 [187]: df2 = df[:20]

In [188]: rolling_corr(df2, df2['B'], window=5)
Out[188]:
                   A    B         C         D
2000-01-03       NaN  NaN       NaN       NaN
2000-01-04       NaN  NaN       NaN       NaN
2000-01-05       NaN  NaN       NaN       NaN
2000-01-06       NaN  NaN       NaN       NaN
2000-01-07  0.806980    1 -0.911973 -0.747745
2000-01-10  0.689915    1 -0.609054 -0.680394
2000-01-11  0.211679    1 -0.383565 -0.164879
2000-01-12  0.286270    1  0.104075  0.345844
2000-01-13 -0.565249    1  0.039148  0.333921
2000-01-14  0.295310    1  0.501143 -0.524100
2000-01-17  0.041252    1  0.868636 -0.577590
2000-01-18  0.205705    1  0.917778 -0.819271
2000-01-19  0.326449    1  0.933352 -0.882750
2000-01-20  0.120893    1  0.409255 -0.795062
2000-01-21  0.680531    1 -0.192045 -0.349044
2000-01-24  0.643667    1 -0.588676  0.473287
2000-01-25  0.703188    1 -0.746130  0.714265
2000-01-26  0.065322    1 -0.209789  0.635360
2000-01-27 -0.429914    1 -0.100807  0.266005
2000-01-28 -0.387498    1  0.512321  0.592033
```

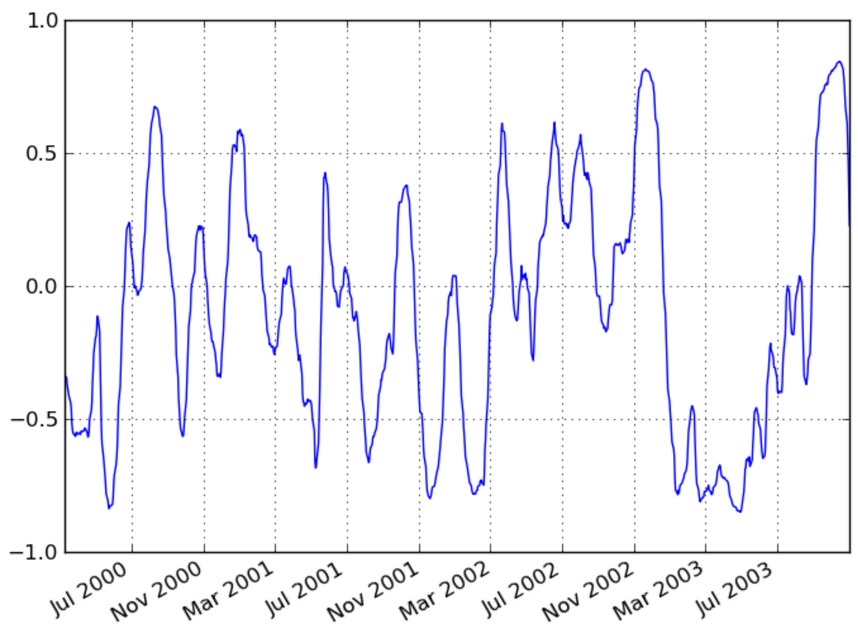
### 8.2.2 Computing rolling pairwise correlations

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 [189]: correls = rolling_corr_pairwise(df, 50)

In [190]: correls[df.index[-50]]
Out[190]:
          A         B         C         D
A  1.000000 -0.177708 -0.253742  0.303872
B -0.177708  1.000000 -0.085484  0.008572
C -0.253742 -0.085484  1.000000 -0.769233
D  0.303872  0.008572 -0.769233  1.000000
```

You can efficiently retrieve the time series of correlations between two columns using `ix` indexing:

```
In [191]: correls.ix[:, 'A', 'C'].plot()
Out[191]: <matplotlib.axes.AxesSubplot at 0x10a7bd950>
```



## 8.3 Exponentially weighted moment functions

A related set of functions are exponentially weighted versions of many of the above statistics. A number of EW (exponentially weighted) functions are provided using the blending method. For example, where $y_t$ is the result and $x_t$ the input, we compute an exponentially weighted moving average as

$$y_t = (1-\alpha)y_{t-1} + \alpha x_t$$

One must have $0 < \alpha \leq 1$, but rather than pass $\alpha$ directly, it's easier to think about either the **span** or **center of mass (com)** of an EW moment:

$$\alpha = \begin{cases} \frac{2}{s+1}, s = \text{span} \\ \frac{1}{c+1}, c = \text{center of mass} \end{cases}$$

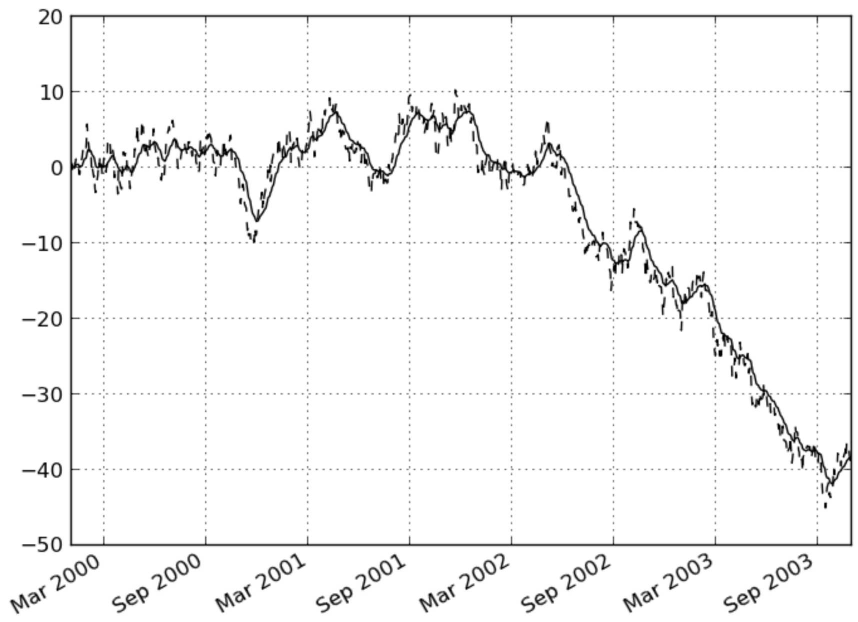
You can pass one or the other to these functions but not both. **Span** corresponds to what is commonly called a "20-day EW moving average" for example. **Center of mass** has a more physical interpretation. For example, **span** = 20 corresponds to **com** = 9.5. Here is the list of functions available:

| Function | Description |
|---|---|
| ewma | EW moving average |
| ewvar | EW moving variance |
| ewstd | EW moving standard deviation |
| ewmcorr | EW moving correlation |
| ewmcov | EW moving covariance |

Here are an example for a univariate time series:

```
In [192]: plt.close('all')

In [193]: ts.plot(style='k--')
```

```
Out[193]: <matplotlib.axes.AxesSubplot at 0x10e817dd0>

In [194]: ewma(ts, span=20).plot(style='k')
Out[194]: <matplotlib.axes.AxesSubplot at 0x10e817dd0>
```



---

**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.

---

## 8.4 Linear and panel regression

---

**Note:** We plan to move this functionality to statsmodels for the next release. Some of the result attributes may change names in order to foster naming consistency with the rest of statsmodels. We will provide every effort to provide compatibility with older versions of pandas, however.

---

We have implemented a very fast set of *moving-window linear regression* classes in pandas. Two different types of regressions are supported:

- Standard ordinary least squares (OLS) multiple regression
- Multiple regression (OLS-based) on panel data including with fixed-effects (also known as entity or individual effects) or time-effects.

Both kinds of linear models are accessed through the `ols` function in the pandas namespace. They all take the following arguments to specify either a static (full sample) or dynamic (moving window) regression:

- `window_type`: `'full sample'` (default), `'expanding'`, or `rolling`
- `window`: size of the moving window in the `window_type='rolling'` case. If `window` is specified, `window_type` will be automatically set to `'rolling'`
- `min_periods`: minimum number of time periods to require to compute the regression coefficients

Generally speaking, the `ols` works by being given a `y` (response) object and an `x` (predictors) object. These can take many forms:

- `y`: a Series, ndarray, or DataFrame (panel model)
- `x`: Series, DataFrame, dict of Series, dict of DataFrame or Panel

Based on the types of `y` and `x`, the model will be inferred to either a panel model or a regular linear model. If the `y` variable is a DataFrame, the result will be a panel model. In this case, the `x` variable must either be a Panel, or a dict of DataFrame (which will be coerced into a Panel).

## 8.4.1 Standard OLS regression

Let's pull in some sample data:

```
In [195]: from pandas.io.data import DataReader

In [196]: symbols = ['MSFT', 'GOOG', 'AAPL']

In [197]: data = dict((sym, DataReader(sym, "yahoo"))
   .....:             for sym in symbols)
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-197-f4577f08f45e> in <module>()
      1 data = dict((sym, DataReader(sym, "yahoo"))
----> 2             for sym in symbols)
<ipython-input-197-f4577f08f45e> in <genexpr>((sym,))
      1 data = dict((sym, DataReader(sym, "yahoo"))
----> 2             for sym in symbols)
/Users/changshe/code/pandas/pandas/io/data.py in DataReader(name, data_source, start, end)
     51
     52     if(data_source == "yahoo"):
---> 53         return get_data_yahoo(name=name, start=start, end=end)
     54     elif(data_source == "fred"):
     55         return get_data_fred(name=name, start=start, end=end)
/Users/changshe/code/pandas/pandas/io/data.py in get_data_yahoo(name, start, end)
    132       '&ignore=.csv'
    133
--> 134     lines = urllib.urlopen(url).read()
    135     return read_csv(StringIO(lines), index_col=0, parse_dates=True)[::-1]
    136
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/urllib.pyc in urlopen(url, data, prox
     84         opener = _urlopener
     85     if data is None:
---> 86         return opener.open(url)
     87     else:
     88         return opener.open(url, data)
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/urllib.pyc in open(self, fullurl, data
    205         try:
    206             if data is None:
--> 207                 return getattr(self, name)(url)
    208             else:
    209                 return getattr(self, name)(url, data)
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/urllib.pyc in open_http(self, url, dat
    342         if realhost: h.putheader('Host', realhost)
    343         for args in self.addheaders: h.putheader(*args)
--> 344         h.endheaders(data)
    345         errcode, errmsg, headers = h.getreply()
    346         fp = h.getfile()
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/httplib.pyc in endheaders(self, messag
    952         else:
    953             raise CannotSendHeader()
--> 954         self._send_output(message_body)
    955 
    956     def request(self, method, url, body=None, headers={}):
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/httplib.pyc in _send_output(self, mess
    812             msg += message_body
    813             message_body = None
--> 814         self.send(msg)
    815         if message_body is not None:
    816             #message_body was not a string (i.e. it is a file) and
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/httplib.pyc in send(self, data)
    774         if self.sock is None:
    775             if self.auto_open:
--> 776                 self.connect()
    777             else:
    778                 raise NotConnected()
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/httplib.pyc in connect(self)
    755         """Connect to the host and port specified in __init__."""
    756         self.sock = socket.create_connection((self.host,self.port),
--> 757                                              self.timeout, self.source_address)
    758 
    759         if self._tunnel_host:
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/socket.pyc in create_connection(addres
    551     host, port = address
    552     err = None
--> 553     for res in getaddrinfo(host, port, 0, SOCK_STREAM):
    554         af, socktype, proto, canonname, sa = res
    555         sock = None
IOError: [Errno socket error] [Errno 8] nodename nor servname provided, or not known

In [198]: panel = Panel(data).swapaxes('items', 'minor')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-198-07873b3393db> in <module>()
----> 1 panel = Panel(data).swapaxes('items', 'minor')
NameError: name 'data' is not defined

In [199]: close_px = panel['Close']
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-199-a66091f76327> in <module>()
----> 1 close_px = panel['Close']
NameError: name 'panel' is not defined

# convert closing prices to returns
In [200]: rets = close_px / close_px.shift(1) - 1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-200-4d4b48582905> in <module>()
----> 1 rets = close_px / close_px.shift(1) - 1
NameError: name 'close_px' is not defined

In [201]: rets.info()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

```
<ipython-input-201-b45c30505b95> in <module>()
----> 1 rets.info()
NameError: name 'rets' is not defined
```

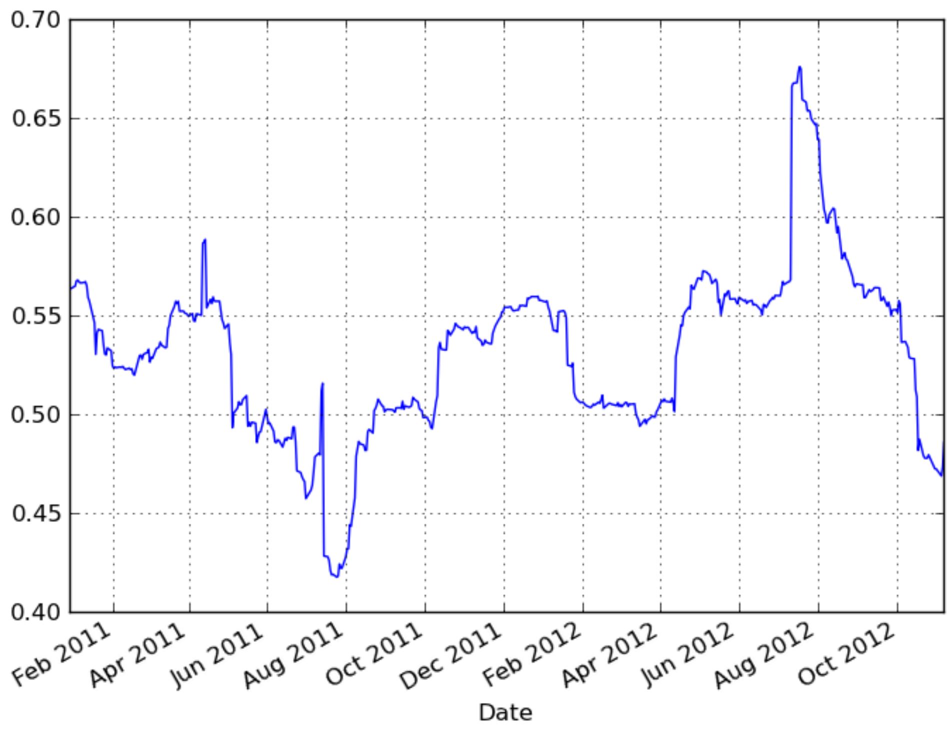
Let's do a static regression of `AAPL` returns on `GOOG` returns:

```
In [202]: model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']])
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-202-06740d688d60> in <module>()
----> 1 model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']])
NameError: name 'rets' is not defined

In [203]: model
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-203-458d5f1afc81> in <module>()
----> 1 model
NameError: name 'model' is not defined

In [204]: model.beta
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-204-0d729d4f44c2> in <module>()
----> 1 model.beta
NameError: name 'model' is not defined
```

If we had passed a Series instead of a DataFrame with the single `GOOG` column, the model would have assigned the generic name `x` to the sole right-hand side variable.

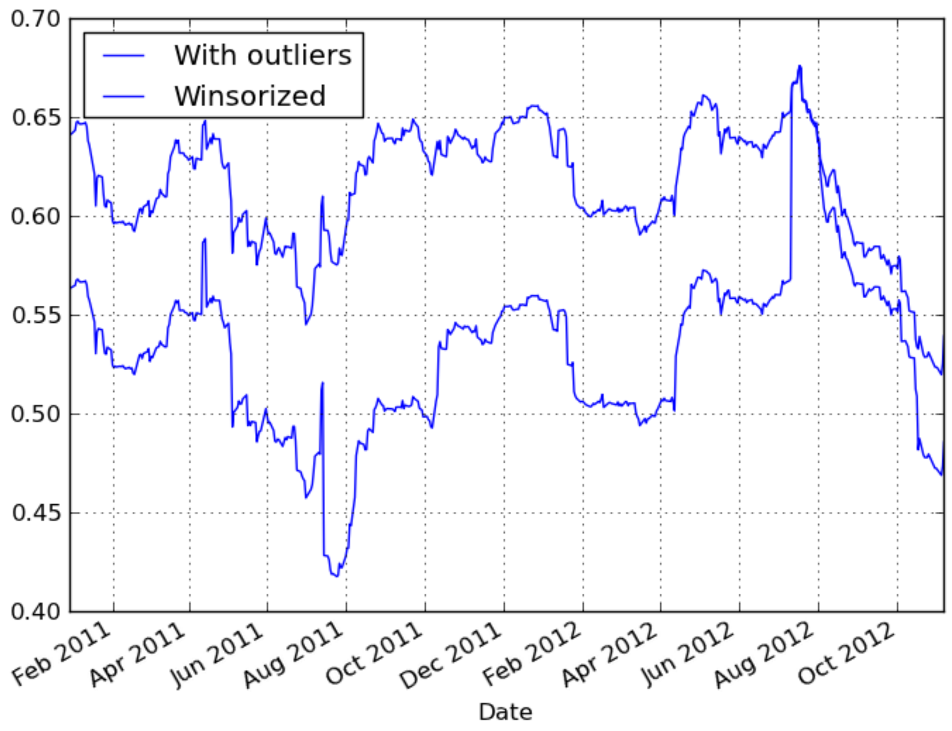
We can do a moving window regression to see how the relationship changes over time:

```
In [205]: model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']],
   .....:             window=250)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-205-77141e77cc71> in <module>()
----> 1 model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']],
      2             window=250)
NameError: name 'rets' is not defined

# just plot the coefficient for GOOG
In [206]: model.beta['GOOG'].plot()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-206-3fa2144a7140> in <module>()
----> 1 model.beta['GOOG'].plot()
NameError: name 'model' is not defined
```



It looks like there are some outliers rolling in and out of the window in the above regression, influencing the results. We could perform a simple winsorization at the 3 STD level to trim the impact of outliers:

```
In [207]: winz = rets.copy()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-207-ef159cc28d64> in <module>()
----> 1 winz = rets.copy()
NameError: name 'rets' is not defined

In [208]: std_1year = rolling_std(rets, 250, min_periods=20)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-208-498f5e1a00b5> in <module>()
----> 1 std_1year = rolling_std(rets, 250, min_periods=20)
NameError: name 'rets' is not defined

# cap at 3 * 1 year standard deviation
In [209]: cap_level = 3 * np.sign(winz) * std_1year
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-209-eceb7c33338e> in <module>()
----> 1 cap_level = 3 * np.sign(winz) * std_1year
NameError: name 'winz' is not defined

In [210]: winz[np.abs(winz) > 3 * std_1year] = cap_level
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-210-dbfbe3507388> in <module>()
----> 1 winz[np.abs(winz) > 3 * std_1year] = cap_level
NameError: name 'cap_level' is not defined

In [211]: winz_model = ols(y=winz['AAPL'], x=winz.ix[:, ['GOOG']],
   .....:               window=250)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-211-71948d79df5f> in <module>()
----> 1 winz_model = ols(y=winz['AAPL'], x=winz.ix[:, ['GOOG']],
      2                  window=250)
NameError: name 'winz' is not defined

In [212]: model.beta['GOOG'].plot(label="With outliers")
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-212-bf6780a1eec6> in <module>()
----> 1 model.beta['GOOG'].plot(label="With outliers")
NameError: name 'model' is not defined

In [213]: winz_model.beta['GOOG'].plot(label="Winsorized"); plt.legend(loc='best')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-213-815cf4b32c25> in <module>()
----> 1 winz_model.beta['GOOG'].plot(label="Winsorized"); plt.legend(loc='best')
NameError: name 'winz_model' is not defined
```



So in this simple example we see the impact of winsorization is actually quite significant. Note the correlation after winsorization remains high:

```
In [214]: winz.corrwith(rets)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-214-0230c93d5cfd> in <module>()
----> 1 winz.corrwith(rets)
NameError: name 'winz' is not defined
```

Multiple regressions can be run by passing a DataFrame with multiple columns for the predictors `x`:

```
In [215]: ols(y=winz['AAPL'], x=winz.drop(['AAPL'], axis=1))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-215-de0445896065> in <module>()
----> 1 ols(y=winz['AAPL'], x=winz.drop(['AAPL'], axis=1))
NameError: name 'winz' is not defined
```

### 8.4.2 Panel regression

We've implemented moving window panel regression on potentially unbalanced panel data (see this article if this means nothing to you). Suppose we wanted to model the relationship between the magnitude of the daily return and trading volume among a group of stocks, and we want to pool all the data together to run one big regression. This is actually quite easy:

```
# make the units somewhat comparable
In [216]: volume = panel['Volume'] / 1e8
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-216-81ef33fb51c8> in <module>()
----> 1 volume = panel['Volume'] / 1e8
NameError: name 'panel' is not defined

In [217]: model = ols(y=volume, x={'return' : np.abs(rets)})
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-217-4e7bad5a4523> in <module>()
----> 1 model = ols(y=volume, x={'return' : np.abs(rets)})
NameError: name 'volume' is not defined

In [218]: model
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-218-458d5f1afc81> in <module>()
----> 1 model
NameError: name 'model' is not defined
```

In a panel model, we can insert dummy (0-1) variables for the "entities" involved (here, each of the stocks) to account the a entity-specific effect (intercept):

```
In [219]: fe_model = ols(y=volume, x={'return' : np.abs(rets)},
   .....:                entity_effects=True)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-219-123d2b0e6684> in <module>()
----> 1 fe_model = ols(y=volume, x={'return' : np.abs(rets)},
      2                entity_effects=True)
NameError: name 'volume' is not defined

In [220]: fe_model
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-220-e0aa1859f068> in <module>()
----> 1 fe_model
NameError: name 'fe_model' is not defined
```

Because we ran the regression with an intercept, one of the dummy variables must be dropped or the design matrix will not be full rank. If we do not use an intercept, all of the dummy variables will be included:

```
In [221]: fe_model = ols(y=volume, x={'return' : np.abs(rets)},
   .....:                 entity_effects=True, intercept=False)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-221-ebdd062db1f9> in <module>()
----> 1 fe_model = ols(y=volume, x={'return' : np.abs(rets)},
      2                  entity_effects=True, intercept=False)
NameError: name 'volume' is not defined

In [222]: fe_model
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-222-e0aa1859f068> in <module>()
----> 1 fe_model
NameError: name 'fe_model' is not defined
```

We can also include *time effects*, which demeans the data cross-sectionally at each point in time (equivalent to including dummy variables for each date). More mathematical care must be taken to properly compute the standard errors in this case:

```
In [223]: te_model = ols(y=volume, x={'return' : np.abs(rets)},
   .....:                 time_effects=True, entity_effects=True)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-223-c1e13ca06b73> in <module>()
----> 1 te_model = ols(y=volume, x={'return' : np.abs(rets)},
      2                  time_effects=True, entity_effects=True)
NameError: name 'volume' is not defined

In [224]: te_model
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-224-b9166339d2b5> in <module>()
----> 1 te_model
NameError: name 'te_model' is not defined
```

Here the intercept (the mean term) is dropped by default because it will be 0 according to the model assumptions, having subtracted off the group means.

### 8.4.3 Result fields and tests

We'll leave it to the user to explore the docstrings and source, especially as we'll be moving this code into statsmodels in the near future.

CHAPTER

NINE

# 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.

## 9.1 Missing data basics

### 9.1.1 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 [732]: df = DataFrame(randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   .....:                columns=['one', 'two', 'three'])

In [733]: df['four'] = 'bar'

In [734]: df['five'] = df['one'] > 0

In [735]: df
Out[735]:
        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

In [736]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [737]: df2
Out[737]:
        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
```

### 9.1.2 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". Lastly, for legacy reasons inf and -inf are also considered to be "null" in computations. Since in NumPy divide-by-zero generates inf or -inf and not NaN, I think you will find this is a worthwhile trade-off (Zen of Python: "practicality beats purity"). 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 [738]: df2['one']
Out[738]:
a    0.059117
b         NaN
c   -0.280853
d         NaN
e    0.863937
f    1.053202
g         NaN
h   -2.359958
Name: one

In [739]: isnull(df2['one'])
Out[739]:
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one

In [740]: df2['four'].notnull()
Out[740]:
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
```

**Summary:** NaN, inf, -inf, and None (in object arrays) are all considered missing by the isnull and notnull functions.

## 9.2 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [741]: a
Out[741]:
        one       two
a  0.059117  1.138469
b  0.059117  1.138469
c -0.280853  0.025653
d -0.280853  0.025653
e  0.863937  0.252462

In [742]: b
Out[742]:
        one       two     three
a  0.059117  1.138469 -2.400634
b       NaN       NaN       NaN
c -0.280853  0.025653 -1.386071
d       NaN       NaN       NaN
e  0.863937  0.252462  1.500571

In [743]: a + b
Out[743]:
        one  three       two
a  0.118234    NaN  2.276938
b       NaN    NaN       NaN
c -0.561707    NaN  0.051306
d       NaN    NaN       NaN
e  1.727874    NaN  0.504923
```

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 [744]: df
Out[744]:
        one       two     three
a  0.059117  1.138469 -2.400634
b       NaN       NaN       NaN
c -0.280853  0.025653 -1.386071
d       NaN       NaN       NaN
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595 -0.374279
g       NaN       NaN       NaN
h -2.359958 -1.157886 -0.551865

In [745]: df['one'].sum()
Out[745]: -0.66455558290247652

In [746]: df.mean(1)
Out[746]:
a   -0.401016
b         NaN
```

```
c   -0.547090
d         NaN
e    0.872323
f   -0.553224
g         NaN
h   -1.356570

In [747]: df.cumsum()
Out[747]:
        one       two     three
a  0.059117  1.138469 -2.400634
b       NaN       NaN       NaN
c -0.221736  1.164122 -3.786705
d       NaN       NaN       NaN
e  0.642200  1.416584 -2.286134
f  1.695403 -0.922011 -2.660413
g       NaN       NaN       NaN
h -0.664556 -2.079897 -3.212278
```

### 9.2.1 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example.

## 9.3 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

### 9.3.1 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 [748]: df2
Out[748]:
        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

In [749]: df2.fillna(0)
Out[749]:
        one       two     three four   five
a  0.059117  1.138469 -2.400634  bar   True
b  0.000000  0.000000  0.000000    0      0
c -0.280853  0.025653 -1.386071  bar  False
d  0.000000  0.000000  0.000000    0      0
e  0.863937  0.252462  1.500571  bar   True
f  1.053202 -2.338595 -0.374279  bar   True
```

```
g  0.000000  0.000000  0.000000    0      0
h -2.359958 -1.157886 -0.551865  bar  False

In [750]: df2['four'].fillna('missing')
Out[750]:
a        bar
b    missing
c        bar
d    missing
e        bar
f        bar
g    missing
h        bar
Name: four
```

**Fill gaps forward or backward**

Using the same filling arguments as *reindexing*, we can propagate non-null values forward or backward:

```
In [751]: df
Out[751]:
        one       two     three
a  0.059117  1.138469 -2.400634
b       NaN       NaN       NaN
c -0.280853  0.025653 -1.386071
d       NaN       NaN       NaN
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595 -0.374279
g       NaN       NaN       NaN
h -2.359958 -1.157886 -0.551865

In [752]: df.fillna(method='pad')
Out[752]:
        one       two     three
a  0.059117  1.138469 -2.400634
b  0.059117  1.138469 -2.400634
c -0.280853  0.025653 -1.386071
d -0.280853  0.025653 -1.386071
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595 -0.374279
g  1.053202 -2.338595 -0.374279
h -2.359958 -1.157886 -0.551865
```

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.

### 9.3.2 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 [753]: df
Out[753]:
        one       two     three
a  0.059117  1.138469 -2.400634
b       NaN  0.000000  0.000000
c -0.280853  0.025653 -1.386071
d       NaN  0.000000  0.000000
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595 -0.374279
g       NaN  0.000000  0.000000
h -2.359958 -1.157886 -0.551865

In [754]: df.dropna(axis=0)
Out[754]:
        one       two     three
a  0.059117  1.138469 -2.400634
c -0.280853  0.025653 -1.386071
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595 -0.374279
h -2.359958 -1.157886 -0.551865

In [755]: df.dropna(axis=1)
Out[755]:
        two     three
a  1.138469 -2.400634
b  0.000000  0.000000
c  0.025653 -1.386071
d  0.000000  0.000000
e  0.252462  1.500571
f -2.338595 -0.374279
g  0.000000  0.000000
h -1.157886 -0.551865

In [756]: df['one'].dropna()
Out[756]:
a    0.059117
c   -0.280853
e    0.863937
f    1.053202
h   -2.359958
Name: one
```

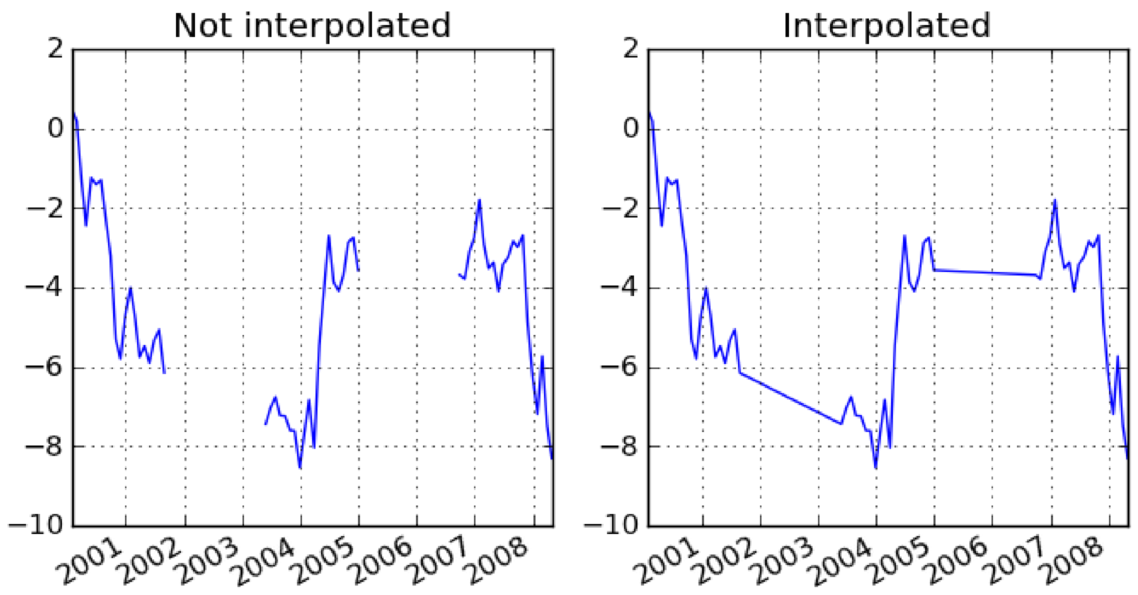
**dropna** is presently only implemented for Series and DataFrame, but will be eventually added to Panel. Series.dropna is a simpler method as it only has one axis to consider. DataFrame.dropna has considerably more options, which can be examined *in the API*.

### 9.3.3 Interpolation

A basic linear **interpolate** method has been implemented on Series with intended use for time series data. There has not been a great deal of demand for interpolation methods outside of the filling methods described above.

```
In [757]: fig, axes = plt.subplots(ncols=2, figsize=(8, 4))

In [758]: ts.plot(ax=axes[0])
Out[758]: <matplotlib.axes.AxesSubplot at 0x1124b1410>

In [759]: ts.interpolate().plot(ax=axes[1])
Out[759]: <matplotlib.axes.AxesSubplot at 0x10e803bd0>
```

```
In [760]: axes[0].set_title('Not interpolated')
Out[760]: <matplotlib.text.Text at 0x1133b0f90>

In [761]: axes[1].set_title('Interpolated')
Out[761]: <matplotlib.text.Text at 0x1146c0790>

In [762]: plt.close('all')
```



## 9.4 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 [763]: s = Series(randn(5), index=[0, 2, 4, 6, 7])

In [764]: s > 0
Out[764]:
0    False
2     True
4     True
6     True
7     True

In [765]: (s > 0).dtype
Out[765]: dtype('bool')
```

```
In [766]: crit = (s > 0).reindex(range(8))

In [767]: crit
Out[767]:
0    False
1      NaN
2     True
3      NaN
4     True
5      NaN
6     True
7     True

In [768]: crit.dtype
Out[768]: dtype('object')
```

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 [769]: reindexed = s.reindex(range(8)).fillna(0)

In [770]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-770-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]
/Users/changshe/code/pandas/pandas/core/series.py in __getitem__(self, key)
    392         # special handling of boolean data with NAs stored in object
    393         # arrays. Since we can't represent NA with dtype=bool
--> 394         if _is_bool_indexer(key):
    395             key = self._check_bool_indexer(key)
    396             key = np.asarray(key, dtype=bool)
/Users/changshe/code/pandas/pandas/core/common.py in _is_bool_indexer(key)
    330         if not lib.is_bool_array(key):
    331             if isnull(key).any():
--> 332                 raise ValueError('cannot index with vector containing '
    333                                  'NA / NaN values')
    334             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 [771]: reindexed[crit.fillna(False)]
Out[771]:
2    1.314232
4    0.690579
6    0.995761
7    2.396780

In [772]: reindexed[crit.fillna(True)]
Out[772]:
1    0.000000
2    1.314232
3    0.000000
4    0.690579
5    0.000000
6    0.995761
7    2.396780
```

CHAPTER

TEN

# GROUP BY: SPLIT-APPLY-COMBINE

By "group by" we are refer 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
- 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.

## 10.1 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 -> 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 [367]: 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 [368]: df
Out[368]:
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

We could naturally group by either the `A` or `B` columns or both:

```
In [369]: grouped = df.groupby('A')

In [370]: grouped = df.groupby(['A', 'B'])
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [371]: def get_letter_type(letter):
   .....:     if letter.lower() in 'aeiou':
   .....:         return 'vowel'
   .....:     else:
   .....:         return 'consonant'
   .....:

In [372]: grouped = df.groupby(get_letter_type, axis=1)
```

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.

### 10.1.1 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 [373]: df.groupby('A').groups
Out[373]: {'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}

In [374]: df.groupby(get_letter_type, axis=1).groups
Out[374]: {'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 [375]: grouped = df.groupby(['A', 'B'])

In [376]: grouped.groups
Out[376]:
{('bar', 'one'): [1],
 ('bar', 'three'): [3],
 ('bar', 'two'): [5],
 ('foo', 'one'): [0, 6],
 ('foo', 'three'): [7],
 ('foo', 'two'): [2, 4]}

In [377]: len(grouped)
Out[377]: 6
```

By default the group keys are sorted during the groupby operation. You may however pass `sort``=``False` for potential speedups:

```
In [378]: df2 = DataFrame({'X' : ['B', 'B', 'A', 'A'], 'Y' : [1, 2, 3, 4]})

In [379]: df2.groupby(['X'], sort=True).sum()
Out[379]:
   Y
X
A  7
B  3

In [380]: df2.groupby(['X'], sort=False).sum()
Out[380]:
   Y
X
B  3
A  7
```

### 10.1.2 GroupBy with MultiIndex

With *hierarchically-indexed data*, it's quite natural to group by one of the levels of the hierarchy.

```
In [381]: s
Out[381]:
first  second
bar    one      -0.424972
       two       0.567020
baz    one       0.276232
       two      -1.087401
```

```
foo    one     -0.673690
       two      0.113648
qux    one     -1.478427
       two      0.524988

In [382]: grouped = s.groupby(level=0)

In [383]: grouped.sum()
Out[383]:
first
bar      0.142048
baz     -0.811169
foo     -0.560041
qux     -0.953439
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```
In [384]: s.groupby(level='second').sum()
Out[384]:
second
one      -2.300857
two       0.118256
```

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 [385]: s.sum(level='second')
Out[385]:
second
one      -2.300857
two       0.118256
```

Also as of v0.6, grouping with multiple levels is supported.

```
In [386]: s
Out[386]:
first  second  third
bar    doo     one      0.404705
               two      0.577046
baz    bee     one     -1.715002
               two     -1.039268
foo    bop     one     -0.370647
               two     -1.157892
qux            one     -1.344312
               two      0.844885

In [387]: s.groupby(level=['first','second']).sum()
Out[387]:
first  second
bar    doo       0.981751
baz    bee      -2.754270
foo    bop      -1.528539
qux    bop      -0.499427
```

More on the `sum` function and aggregation later.

### 10.1.3 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 [388]: grouped = df.groupby(['A'])

In [389]: grouped_C = grouped['C']

In [390]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [391]: df['C'].groupby(df['A'])
Out[391]: <pandas.core.groupby.SeriesGroupBy at 0x11382ab10>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

## 10.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby`:

```
In [392]: grouped = df.groupby('A')

In [393]: for name, group in grouped:
   .....:        print name
   .....:        print group
   .....:
bar
     A      B         C         D
1  bar    one -0.282863 -2.104569
3  bar  three -1.135632  1.071804
5  bar    two -0.173215 -0.706771
foo
     A      B         C         D
0  foo    one  0.469112 -0.861849
2  foo    two -1.509059 -0.494929
4  foo    two  1.212112  0.721555
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

In the case of grouping by multiple keys, the group name will be a tuple:

```
In [394]: for name, group in df.groupby(['A', 'B']):
   .....:        print name
   .....:        print group
   .....:
('bar', 'one')
     A    B         C         D
1  bar  one -0.282863 -2.104569
('bar', 'three')
     A      B         C         D
3  bar  three -1.135632  1.071804
('bar', 'two')
     A    B         C         D
5  bar  two -0.173215 -0.706771
```

```
('foo', 'one')
     A    B         C         D
0  foo  one  0.469112 -0.861849
6  foo  one  0.119209 -1.039575
('foo', 'three')
     A      B         C        D
7  foo  three -1.044236  0.27186
('foo', 'two')
     A    B         C         D
2  foo  two -1.509059 -0.494929
4  foo  two  1.212112  0.721555
```

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:`.

## 10.3 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 [395]: grouped = df.groupby('A')

In [396]: grouped.aggregate(np.sum)
Out[396]:
            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938

In [397]: grouped = df.groupby(['A', 'B'])

In [398]: grouped.aggregate(np.sum)
Out[398]:
                  C         D
A   B
bar one   -0.282863 -2.104569
    three -1.135632  1.071804
    two   -0.173215 -0.706771
foo one    0.588321 -1.901424
    three -1.044236  0.271860
    two   -0.296946  0.226626
```

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 [399]: grouped = df.groupby(['A', 'B'], as_index=False)

In [400]: grouped.aggregate(np.sum)
Out[400]:
     A      B         C         D
0  bar    one -0.282863 -2.104569
1  bar  three -1.135632  1.071804
2  bar    two -0.173215 -0.706771
3  foo    one  0.588321 -1.901424
4  foo  three -1.044236  0.271860
```

```
5  foo    two -0.296946  0.226626

In [401]: df.groupby('A', as_index=False).sum()
Out[401]:
     A         C         D
0  bar -1.591710 -1.739537
1  foo -0.752861 -1.402938
```

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 [402]: df.groupby(['A', 'B']).sum().reset_index()
Out[402]:
     A      B         C         D
0  bar    one -0.282863 -2.104569
1  bar  three -1.135632  1.071804
2  bar    two -0.173215 -0.706771
3  foo    one  0.588321 -1.901424
4  foo  three -1.044236  0.271860
5  foo    two -0.296946  0.226626
```

### 10.3.1 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 [403]: grouped = df.groupby('A')

In [404]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[404]:
         mean       std       sum
A
bar -0.530570  0.526860 -1.591710
foo -0.150572  1.113308 -0.752861
```

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 [405]: grouped['D'].agg({'result1' : np.sum,
   .....:                   'result2' : np.mean})
Out[405]:
      result1   result2
A
bar -1.739537 -0.579846
foo -1.402938 -0.280588
```

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 [406]: grouped.agg([np.sum, np.mean, np.std])
Out[406]:
            C                             D
         mean       std       sum      mean       std       sum
A
bar -0.530570  0.526860 -1.591710 -0.579846  1.591986 -1.739537
foo -0.150572  1.113308 -0.752861 -0.280588  0.753219 -1.402938
```

Passing a dict of functions has different behavior by default, see the next section.

### 10.3.2 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 [407]: grouped.agg({'C' : np.sum,
   .....:              'D' : lambda x: np.std(x, ddof=1)})
Out[407]:
            C         D
A
bar -1.591710  1.591986
foo -0.752861  0.753219
```

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 [408]: grouped.agg({'C' : 'sum', 'D' : 'std'})
Out[408]:
            C         D
A
bar -1.591710  1.591986
foo -0.752861  0.753219
```

### 10.3.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, and `std`, have optimized Cython implementations:

```
In [409]: df.groupby('A').sum()
Out[409]:
            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938

In [410]: df.groupby(['A', 'B']).mean()
Out[410]:
                  C         D
A   B
bar one   -0.282863 -2.104569
    three -1.135632  1.071804
    two   -0.173215 -0.706771
foo one    0.294161 -0.950712
    three -1.044236  0.271860
    two   -0.148473  0.113313
```

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).

## 10.4 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 a data set within a group:

```
In [411]: tsdf = DataFrame(randn(1000, 3),
   .....:                  index=DateRange('1/1/2000', periods=1000),
   .....:                  columns=['A', 'B', 'C'])
```

```
In [412]: tsdf
Out[412]:
<class 'pandas.core.frame.DataFrame'>
DateRange: 1000 entries, 2000-01-03 00:00:00 to 2003-10-31 00:00:00
offset: <1 BusinessDay>
Data columns:
A    1000  non-null values
B    1000  non-null values
C    1000  non-null values
dtypes: float64(3)

In [413]: zscore = lambda x: (x - x.mean()) / x.std()

In [414]: transformed = tsdf.groupby(lambda x: x.year).transform(zscore)
```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
In [415]: grouped = transformed.groupby(lambda x: x.year)

# OK, close enough to zero
In [416]: grouped.mean()
Out[416]:
        A   B   C
key_0
2000   -0  -0   0
2001   -0   0   0
2002    0  -0  -0
2003    0  -0  -0

In [417]: grouped.std()
Out[417]:
        A   B   C
key_0
2000    1   1   1
2001    1   1   1
2002    1   1   1
2003    1   1   1
```

## 10.5 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 [418]: grouped = df.groupby('A')

In [419]: grouped.agg(lambda x: x.std())
Out[419]:
      B         C         D
A
bar NaN  0.526860  1.591986
foo NaN  1.113308  0.753219
```

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 [420]: grouped.std()
Out[420]:
            C         D
A
bar  0.526860  1.591986
foo  1.113308  0.753219
```

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 [421]: tsdf.ix[::2] = np.nan

In [422]: grouped = tsdf.groupby(lambda x: x.year)

In [423]: grouped.fillna(method='pad')
Out[423]:
<class 'pandas.core.frame.DataFrame'>
DateRange: 1000 entries, 2000-01-03 00:00:00 to 2003-10-31 00:00:00
offset: <1 BusinessDay>
Data columns:
A    997  non-null values
B    997  non-null values
C    997  non-null values
dtypes: float64(3)
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

## 10.6 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 [424]: df
Out[424]:
     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

In [425]: grouped = df.groupby('A')

# could also just call .describe()
In [426]: grouped['C'].apply(lambda x: x.describe())
Out[426]:
A
bar  count    3.000000
```

```
     mean    -0.530570
     std      0.526860
     min     -1.135632
     25%     -0.709248
     50%     -0.282863
     75%     -0.228039
     max     -0.173215
foo  count    5.000000
     mean    -0.150572
     std      1.113308
     min     -1.509059
     25%     -1.044236
     50%      0.119209
     75%      0.469112
     max      1.212112
```

The dimension of the returned result can also change:

```
In [427]: grouped = df.groupby('A')['C']

In [428]: def f(group):
   .....:     return DataFrame({'original' : group,
   .....:                       'demeaned' : group - group.mean()})
   .....:

In [429]: grouped.apply(f)
Out[429]:
   demeaned  original
0  0.619685  0.469112
1  0.247707 -0.282863
2 -1.358486 -1.509059
3 -0.605062 -1.135632
4  1.362684  1.212112
5  0.357355 -0.173215
6  0.269781  0.119209
7 -0.893664 -1.044236
```

## 10.7 Other useful features

### 10.7.1 Automatic exclusion of "nuisance" columns

Again consider the example DataFrame we've been looking at:

```
In [430]: df
Out[430]:
     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
```

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 [431]: df.groupby('A').std()
Out[431]:
            C         D
A
bar  0.526860  1.591986
foo  1.113308  0.753219
```

## 10.7.2 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).

CHAPTER

ELEVEN

# 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.

## 11.1 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 [636]: df = DataFrame(np.random.randn(10, 4))

In [637]: df
Out[637]:
          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

# break it into pieces
In [638]: pieces = [df[:3], df[3:7], df[7:]]

In [639]: concatenated = concat(pieces)

In [640]: concatenated
Out[640]:
          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
```

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 [641]: concatenated = concat(pieces, keys=['first', 'second', 'third'])

In [642]: concatenated
Out[642]:
                 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
```

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 [643]: concatenated.ix['second']
Out[643]:
          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
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

### 11.1.1 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 a example of each of these methods. First, the default `join='outer'` behavior:

```
In [644]: from pandas.util.testing import rands

In [645]: df = DataFrame(np.random.randn(10, 4), columns=['a', 'b', 'c', 'd'],
   .....:                index=[rands(5) for _ in xrange(10)])

In [646]: df
Out[646]:
              a         b         c         d
RQlXD -1.294524  0.413738  0.276662 -0.472035
p9y7F -0.013960 -0.362543 -0.006154 -0.923061
XbyT9  0.895717  0.805244 -1.206412  2.565646
8PnTm  1.431256  1.340309 -1.170299 -0.226169
JX6kv  0.410835  0.813850  0.132003 -0.827317
15BgG -0.076467 -1.187678  1.130127 -1.436737
UhTku -1.413681  1.607920  1.024180  0.569605
rJ9U4  0.875906 -2.211372  0.974466 -2.006747
TMmI9 -0.410001 -0.078638  0.545952 -1.219217
0ij9W -1.226825  0.769804 -1.281247 -0.727707

In [647]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   .....:         df.ix[-7:, ['d']]], axis=1)
Out[647]:
              a         b         c         d
0ij9W       NaN       NaN       NaN -0.727707
15BgG -0.076467 -1.187678  1.130127 -1.436737
8PnTm  1.431256  1.340309 -1.170299 -0.226169
JX6kv  0.410835  0.813850  0.132003 -0.827317
RQlXD -1.294524  0.413738       NaN       NaN
TMmI9       NaN       NaN       NaN -1.219217
UhTku -1.413681  1.607920  1.024180  0.569605
XbyT9  0.895717  0.805244 -1.206412       NaN
p9y7F -0.013960 -0.362543       NaN       NaN
rJ9U4       NaN       NaN  0.974466 -2.006747
```

Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```
In [648]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   .....:              df.ix[-7:, ['d']]], axis=1, join='inner')
Out[648]:
              a         b         c         d
8PnTm  1.431256  1.340309 -1.170299 -0.226169
JX6kv  0.410835  0.813850  0.132003 -0.827317
15BgG -0.076467 -1.187678  1.130127 -1.436737
UhTku -1.413681  1.607920  1.024180  0.569605
```

Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

```
In [649]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   .....:              df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
Out[649]:
              a         b         c         d
RQ1XD -1.294524  0.413738       NaN       NaN
p9y7F -0.013960 -0.362543       NaN       NaN
XbyT9  0.895717  0.805244 -1.206412       NaN
8PnTm  1.431256  1.340309 -1.170299 -0.226169
JX6kv  0.410835  0.813850  0.132003 -0.827317
15BgG -0.076467 -1.187678  1.130127 -1.436737
UhTku -1.413681  1.607920  1.024180  0.569605
rJ9U4       NaN       NaN  0.974466 -2.006747
TMmI9       NaN       NaN       NaN -1.219217
0ij9W       NaN       NaN       NaN -0.727707
```

### 11.1.2 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 [650]: s = Series(randn(10), index=np.arange(10))

In [651]: s1 = s[:5] # note we're slicing with labels here, so 5 is included

In [652]: s2 = s[6:]

In [653]: s1.append(s2)
Out[653]:
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
```

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

```
In [654]: df = DataFrame(randn(6, 4), index=DateRange('1/1/2000', periods=6),
   .....:                 columns=['A', 'B', 'C', 'D'])

In [655]: df1 = df.ix[:3]
In [656]: df2 = df.ix[3:, :3]

In [657]: df1
Out[657]:
                   A         B         C         D
2000-01-03  0.176444  0.403310 -0.154951  0.301624
2000-01-04 -2.179861 -1.369849 -0.954208  1.462696
2000-01-05 -1.743161 -0.826591 -0.345352  1.314232

In [658]: df2
Out[658]:
                   A         B         C
2000-01-06  0.690579  0.995761  2.396780
2000-01-07  3.357427 -0.317441 -1.236269
2000-01-10 -0.487602 -0.082240 -2.182937

In [659]: df1.append(df2)
Out[659]:
                   A         B         C         D
2000-01-03  0.176444  0.403310 -0.154951  0.301624
2000-01-04 -2.179861 -1.369849 -0.954208  1.462696
2000-01-05 -1.743161 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780       NaN
2000-01-07  3.357427 -0.317441 -1.236269       NaN
2000-01-10 -0.487602 -0.082240 -2.182937       NaN
```

`append` may take multiple objects to concatenate:

```
In [660]: df1 = df.ix[:2]

In [661]: df2 = df.ix[2:4]

In [662]: df3 = df.ix[4:]

In [663]: df1.append([df2,df3])
Out[663]:
                   A         B         C         D
2000-01-03  0.176444  0.403310 -0.154951  0.301624
2000-01-04 -2.179861 -1.369849 -0.954208  1.462696
2000-01-05 -1.743161 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780  0.014871
2000-01-07  3.357427 -0.317441 -1.236269  0.896171
2000-01-10 -0.487602 -0.082240 -2.182937  0.380396
```

**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.

### 11.1.3 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 [664]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])

In [665]: df2 = DataFrame(randn(3, 4), columns=['A', 'B', 'C', 'D'])
In [666]: df1
Out[666]:
          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

In [667]: df2
Out[667]:
          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
```

To do this, use the `ignore_index` argument:

```
In [668]: concat([df1, df2], ignore_index=True)
Out[668]:
          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
```

This is also a valid argument to `DataFrame.append`:

```
In [669]: df1.append(df2, ignore_index=True)
Out[669]:
          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
```

### 11.1.4 More concatenating with group keys

Let's consider a variation on the first example presented:

```
In [670]: df = DataFrame(np.random.randn(10, 4))

In [671]: df
Out[671]:
          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

# break it into pieces
In [672]: pieces = [df.ix[:, [0, 1]], df.ix[:, [2]], df.ix[:, [3]]]

In [673]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'])

In [674]: result
Out[674]:
        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
```

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 [675]: pieces = {'one': df.ix[:, [0, 1]],
   .....:           'two': df.ix[:, [2]],
   .....:           'three': df.ix[:, [3]]}

In [676]: concat(pieces, axis=1)
Out[676]:
        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

In [677]: concat(pieces, keys=['three', 'two'])
Out[677]:
              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
      5  1.171216       NaN
      6 -0.303421       NaN
      7  0.384316       NaN
      8  0.473424       NaN
      9  0.650776       NaN
```

The MultiIndex created has levels that are constructed from the passed keys and the columns of the DataFrame pieces:

```
In [678]: result.columns.levels
Out[678]: [Index([one, two, three], dtype=object), Int64Index([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 [679]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'],
   .....:                 levels=[['three', 'two', 'one', 'zero']],
   .....:                 names=['group_key'])

In [680]: result
Out[680]:
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

In [681]: result.columns.levels
Out[681]: [Index([three, two, one, zero], dtype=object), Int64Index([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.

### 11.1.5 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 [682]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [683]: df
Out[683]:
```

```
          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

In [684]: s = df.xs(3)

In [685]: df.append(s, ignore_index=True)
Out[685]:
          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
```

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 [686]: df = DataFrame(np.random.randn(5, 4),
   .....:                columns=['foo', 'bar', 'baz', 'qux'])

In [687]: dicts = [{'foo': 1, 'bar': 2, 'baz': 3, 'peekaboo': 4},
   .....:          {'foo': 5, 'bar': 6, 'baz': 7, 'peekaboo': 8}]

In [688]: result = df.append(dicts, ignore_index=True)

In [689]: result
Out[689]:
        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
```

## 11.2 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.

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.

### 11.2.1 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 [690]: left = DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [691]: right = DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [692]: left
Out[692]:
   key  lval
0  foo     1
1  foo     2

In [693]: right
Out[693]:
   key  rval
0  foo     4
1  foo     5

In [694]: merge(left, right, on='key')
Out[694]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Here is a more complicated example with multiple join keys:

```
In [695]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   .....:                   'key2': ['one', 'two', 'one'],
   .....:                   'lval': [1, 2, 3]})

In [696]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   .....:                    'key2': ['one', 'one', 'one', 'two'],
   .....:                    'rval': [4, 5, 6, 7]})

In [697]: merge(left, right, how='outer')
Out[697]:
  key1 key2  lval  rval
0  bar  one     3     6
1  bar  two   NaN     7
2  foo  one     1     4
3  foo  one     1     5
4  foo  two     2   NaN

In [698]: merge(left, right, how='inner')
Out[698]:
  key1 key2  lval  rval
0  bar  one     3     6
1  foo  one     1     4
2  foo  one     1     5
```

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 |

Note that if using the index from either the left or right DataFrame (or both) using the left_index / right_index options, the join operation is no longer a many-to-many join by construction, as the index values are necessarily unique. There will be some examples of this below.

## 11.2.2 Joining on index

DataFrame.join is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Here is a very basic example:

```
In [699]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [700]: df1 = df.ix[1:, ['A', 'B']]

In [701]: df2 = df.ix[:5, ['C', 'D']]

In [702]: df1
Out[702]:
          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

In [703]: df2
Out[703]:
          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

In [704]: df1.join(df2)
Out[704]:
          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
```

```
In [705]: df1.join(df2, how='outer')
Out[705]:
          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

In [706]: df1.join(df2, how='inner')
Out[706]:
          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
```

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 [707]: merge(df1, df2, left_index=True, right_index=True, how='outer')
Out[707]:
          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
```

### 11.2.3 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 [708]: df['key'] = ['foo', 'bar'] * 4

In [709]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
   .....:                     columns=['j1', 'j2'])

In [710]: df
Out[710]:
          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

In [711]: to_join
Out[711]:
           j1        j2
bar  0.503703  0.413086
foo -1.139050  0.660342

In [712]: df.join(to_join, on='key')
Out[712]:
          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

In [713]: merge(df, to_join, left_on='key', right_index=True,
   .....:       how='left', sort=False)
Out[713]:
          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
```

To join on multiple keys, the passed DataFrame must have a `MultiIndex`:

```
In [714]: 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 [715]: to_join = DataFrame(np.random.randn(10, 3), index=index,
   .....:                     columns=['j_one', 'j_two', 'j_three'])

# a little relevant example with NAs
In [716]: key1 = ['bar', 'bar', 'bar', 'foo', 'foo', 'baz', 'baz', 'qux',
   .....:         'qux', 'snap']

In [717]: key2 = ['two', 'one', 'three', 'one', 'two', 'one', 'two', 'two',
   .....:         'three', 'one']

In [718]: data = np.random.randn(len(key1))
In [719]: data = DataFrame({'key1' : key1, 'key2' : key2,
   .....:                  'data' : data})

In [720]: data
Out[720]:
       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

In [721]: to_join
Out[721]:
                 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
```

Now this can be joined by passing the two key column names:

```
In [722]: data.join(to_join, on=['key1', 'key2'])
Out[722]:
       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
```

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 [723]: data.join(to_join, on=['key1', 'key2'], how='inner')
Out[723]:
       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
```

As you can see, this drops any rows where there was no match.

### 11.2.4 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 [724]: left = DataFrame({'key': ['foo', 'foo'], 'value': [1, 2]})

In [725]: right = DataFrame({'key': ['foo', 'foo'], 'value': [4, 5]})

In [726]: merge(left, right, on='key', suffixes=['_left', '_right'])
Out[726]:
   key  value_left  value_right
0  foo           1            4
1  foo           1            5
2  foo           2            4
3  foo           2            5
```

`DataFrame.join` has `lsuffix` and `rsuffix` arguments which behave similarly.

### 11.2.5 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 [727]: df1 = df.ix[:, ['A', 'B']]

In [728]: df2 = df.ix[:, ['C', 'D']]

In [729]: df3 = df.ix[:, ['key']]

In [730]: df1
Out[730]:
          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

In [731]: df1.join([df2, df3])
Out[731]:
          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
```

CHAPTER

TWELVE

# RESHAPING AND PIVOT TABLES

## 12.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```
In [776]: df
Out[776]:
                   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
```

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 [777]: df[df['variable'] == 'A']
Out[777]:
                  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
```

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 [778]: df.pivot(index='date', columns='variable', values='value')
Out[778]:
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
```

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 [779]: df['value2'] = df['value'] * 2

In [780]: pivoted = df.pivot('date', 'variable')

In [781]: pivoted
Out[781]:
               value                                   value2
variable           A         B         C         D         A         B         C         D
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569  0.938225 -2.271265  0.238417 -4.209138
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224 -2.088472 -0.989859
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429 -1.723698  2.143608
```

You of course can then select subsets from the pivoted DataFrame:

```
In [782]: pivoted['value2']
Out[782]:
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
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

## 12.2 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 [783]: tuples = zip(*[['bar', 'bar', 'baz', 'baz',
   .....:                 'foo', 'foo', 'qux', 'qux'],
   .....:                ['one', 'two', 'one', 'two',
   .....:                 'one', 'two', 'one', 'two']])

In [784]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])
In [785]: df = DataFrame(randn(8, 2), index=index, columns=['A', 'B'])

In [786]: df2 = df[:4]

In [787]: df2
Out[787]:
                     A         B
first second
bar   one     0.721555 -0.706771
      two    -1.039575  0.271860
baz   one    -0.424972  0.567020
      two     0.276232 -1.087401
```

The stack function "compresses" a level in the DataFrame's columns to produce either:

- A Series, in the case of a simple column Index
- A DataFrame, in the case of a MultiIndex in the columns

If the columns have a MultiIndex, you can choose which level to stack. The stacked level becomes the new lowest level in a MultiIndex on the columns:

```
In [788]: stacked = df2.stack()

In [789]: stacked
Out[789]:
first  second
bar    one     A    0.721555
               B   -0.706771
       two     A   -1.039575
               B    0.271860
baz    one     A   -0.424972
               B    0.567020
       two     A    0.276232
               B   -1.087401
```

With a "stacked" DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack is unstack, which by default unstacks the **last level**:

```
In [790]: stacked.unstack()
Out[790]:
                     A         B
first second
bar   one     0.721555 -0.706771
      two    -1.039575  0.271860
baz   one    -0.424972  0.567020
      two     0.276232 -1.087401

In [791]: stacked.unstack(1)
Out[791]:
second         one       two
first
bar   A   0.721555 -1.039575
      B  -0.706771  0.271860
baz   A  -0.424972  0.276232
      B   0.567020 -1.087401

In [792]: stacked.unstack(0)
Out[792]:
first            bar       baz
```

```
second
one    A  0.721555 -0.424972
       B -0.706771  0.567020
two    A -1.039575  0.276232
       B  0.271860 -1.087401
```

If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [793]: stacked.unstack('second')
Out[793]:
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
```

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 [794]: columns = MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
   .....:                                   ('B', 'cat'), ('A', 'dog')],
   .....:                                  names=['exp', 'animal'])

In [795]: df = DataFrame(randn(8, 4), index=index, columns=columns)

In [796]: df2 = df.ix[[0, 1, 2, 4, 5, 7]]

In [797]: df2
Out[797]:
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
```

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [798]: df2.stack('exp')
Out[798]:
animal                 cat       dog
first second exp
bar   one    A   -0.370647  0.844885
             B   -1.344312 -1.157892
      two    A    1.075770 -1.469388
             B    1.643563 -0.109050
baz   one    A    0.357021 -0.968914
             B   -1.776904 -0.674600
foo          A   -0.013960 -0.923061
             B   -0.006154 -0.362543
      two    A    0.895717  2.565646
```

```
             B   -1.206412  0.805244
qux          A    0.410835 -0.827317
             B    0.132003  0.813850
```

```
In [799]: df2.stack('animal')
Out[799]:
exp                         A         B
first second animal
bar   one    cat    -0.370647 -1.344312
             dog     0.844885 -1.157892
      two    cat     1.075770  1.643563
             dog    -1.469388 -0.109050
baz   one    cat     0.357021 -1.776904
             dog    -0.968914 -0.674600
foo          cat    -0.013960 -0.006154
             dog    -0.923061 -0.362543
      two    cat     0.895717 -1.206412
             dog     2.565646  0.805244
qux          cat     0.410835  0.132003
             dog    -0.827317  0.813850
```

Unstacking when the columns are a `MultiIndex` is also careful about doing the right thing:

```
In [800]: df[:3].unstack(0)
Out[800]:
exp            A                   B                                     A
animal        cat                 dog               cat                dog
first         bar       baz       bar     baz       bar       baz       bar       baz
second
one    -0.370647  0.357021 -1.157892 -0.6746 -1.344312 -1.776904  0.844885 -0.968914
two     1.075770       NaN -0.109050     NaN  1.643563       NaN -1.469388       NaN
```

```
In [801]: df2.unstack(1)
Out[801]:
exp            A                   B                                         A
animal        cat                 dog                 cat                 dog
second        one       two       one       two       one       two       one       two
first
bar    -0.370647  1.075770 -1.157892 -0.109050 -1.344312  1.643563  0.844885 -1.469388
baz     0.357021       NaN -0.674600       NaN -1.776904       NaN -0.968914       NaN
foo    -0.013960  0.895717 -0.362543  0.805244 -0.006154 -1.206412 -0.923061  2.565646
qux          NaN  0.410835       NaN  0.813850       NaN  0.132003       NaN -0.827317
```

## 12.3 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".

For instance,

```
In [802]: cheese = DataFrame({'first' : ['John', 'Mary'],
   .....:                     'last' : ['Doe', 'Bo'],
   .....:                     'height' : [5.5, 6.0],
   .....:                     'weight' : [130, 150]})
```

```
In [803]: cheese
Out[803]:
  first  height last  weight
0  John     5.5  Doe     130
1  Mary     6.0   Bo     150

In [804]: melt(cheese, id_vars=['first', 'last'])
Out[804]:
  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
```

## 12.4 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 [805]: df
Out[805]:
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

In [806]: df.stack().mean(1).unstack()
Out[806]:
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

# same result, another way
In [807]: df.groupby(level=1, axis=1).mean()
Out[807]:
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

In [808]: df.stack().groupby(level=1).mean()
Out[808]:
exp             A         B
second
one      0.016301 -0.644049
two      0.110588  0.346200

In [809]: df.mean().unstack(0)
Out[809]:
exp             A         B
animal
cat      0.311433 -0.431481
dog     -0.184544  0.133632
```

## 12.5 Pivot tables and cross-tabulations

The function `pandas.pivot_table` can be used to create spreadsheet-style pivot tables. 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 [810]: 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 [811]: df
Out[811]:
        A  B    C         D         E
0     one  A  foo -0.076467  0.959726
1     one  B  foo -1.187678 -1.110336
2     two  C  foo  1.130127 -0.619976
3   three  A  bar -1.436737  0.149748
4     one  B  bar -1.413681 -0.732339
5     one  C  bar  1.607920  0.687738
6     two  A  foo  1.024180  0.176444
7   three  B  foo  0.569605  0.403310
8     one  C  foo  0.875906 -0.154951
9     one  A  bar -2.211372  0.301624
10    two  B  bar  0.974466 -2.179861
11  three  C  bar -2.006747 -1.369849
12    one  A  foo -0.410001 -0.954208
13    one  B  foo -0.078638  1.462696
```

```
14    two  C  foo  0.545952 -1.743161
15  three  A  bar -1.219217 -0.826591
16    one  B  bar -1.226825 -0.345352
17    one  C  bar  0.769804  1.314232
18    two  A  foo -1.281247  0.690579
19  three  B  foo -0.727707  0.995761
20    one  C  foo -0.121306  2.396780
21    one  A  bar -0.097883  0.014871
22    two  B  bar  0.695775  3.357427
23  three  C  bar  0.341734 -0.317441
```

We can produce pivot tables from this data very easily:

```
In [812]: pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
Out[812]:
C             bar       foo
A     B
one   A -1.154627 -0.243234
      B -1.320253 -0.633158
      C  1.188862  0.377300
three A -1.327977       NaN
      B       NaN -0.079051
      C -0.832506       NaN
two   A       NaN -0.128534
      B  0.835120       NaN
      C       NaN  0.838040

In [813]: pivot_table(df, values='D', rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
Out[813]:
A       one                 three                 two
C       bar       foo       bar       foo       bar       foo
B
A -2.309255 -0.486468 -2.655954       NaN       NaN -0.257067
B -2.640506 -1.266315       NaN -0.158102  1.670241       NaN
C  2.377724  0.754600 -1.665013       NaN       NaN  1.676079

In [814]: pivot_table(df, values=['D','E'], rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
Out[814]:
          D                                                           E
A       one                 three                 two                 one                 three
C       bar       foo       bar       foo       bar       foo       bar       foo       bar       foo
B
A -2.309255 -0.486468 -2.655954       NaN       NaN -0.257067  0.316495  0.005518 -0.676843       NaN
B -2.640506 -1.266315       NaN -0.158102  1.670241       NaN -1.077692  0.352360       NaN  1.39907
C  2.377724  0.754600 -1.665013       NaN       NaN  1.676079  2.001971  2.241830 -1.687290       NaN
```

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 [815]: pivot_table(df, rows=['A', 'B'], cols=['C'])
Out[815]:
                D                   E
C             bar       foo       bar       foo
A     B
one   A -1.154627 -0.243234  0.158248  0.002759
      B -1.320253 -0.633158 -0.538846  0.176180
      C  1.188862  0.377300  1.000985  1.120915
three A -1.327977       NaN -0.338421       NaN
```

```
      B       NaN -0.079051       NaN  0.699535
      C -0.832506       NaN -0.843645       NaN
two   A       NaN -0.128534       NaN  0.433512
      B  0.835120       NaN  0.588783       NaN
      C       NaN  0.838040       NaN -1.181568
```

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```
In [816]: table = pivot_table(df, rows=['A', 'B'], cols=['C'])

In [817]: print table.to_string(na_rep='')
                D                   E
C             bar       foo       bar       foo
A     B
one   A -1.154627 -0.243234  0.158248  0.002759
      B -1.320253 -0.633158 -0.538846  0.176180
      C  1.188862  0.377300  1.000985  1.120915
three A -1.327977           -0.338421
      B           -0.079051            0.699535
      C -0.832506           -0.843645
two   A           -0.128534            0.433512
      B  0.835120            0.588783
      C            0.838040           -1.181568
```

Note that `pivot_table` is also available as an instance method on DataFrame.

### 12.5.1 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 [818]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [819]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [820]: b = np.array([one, one, two, one, two, one], dtype=object)

In [821]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [822]: crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
```

```
Out[822]:
b     one         two
c     dull  shiny  dull  shiny
a
bar      1      0     0      1
foo      2      1     1      0
```

### 12.5.2 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 [823]: df.pivot_table(rows=['A', 'B'], cols='C', margins=True, aggfunc=np.std)
Out[823]:
                  D                             E
C               bar       foo       All       bar       foo       All
A     B
one   A  1.494463  0.235844  1.019752  0.202765  1.353355  0.795165
      B  0.132127  0.784210  0.606779  0.273641  1.819408  1.139647
      C  0.592638  0.705136  0.708771  0.442998  1.804346  1.074910
three A  0.153810       NaN  0.153810  0.690376       NaN  0.690376
      B       NaN  0.917338  0.917338       NaN  0.418926  0.418926
      C  1.660627       NaN  1.660627  0.744165       NaN  0.744165
two   A       NaN  1.630183  1.630183       NaN  0.363548  0.363548
      B  0.197065       NaN  0.197065  3.915454       NaN  3.915454
      C       NaN  0.413074  0.413074       NaN  0.794212  0.794212
All      1.294620  0.824989  1.064129  1.403041  1.188419  1.248988
```

CHAPTER

THIRTEEN

# 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. Over the coming year we will be looking to consolidate the various Python libraries for time series data, e.g. `scikits.timeseries`, using the new NumPy `datetime64` dtype, to create a very nice integrated solution. Everything in pandas at the moment is based on using Python `datetime` objects.

In working with time series data, we will frequently seek to:

- generate sequences of fixed-frequency dates
- 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.

**Note:** This area of pandas has gotten less development attention recently, though this should change in the near future.

## 13.1 DateOffset objects

A `DateOffset` instance represents a frequency increment. Different offset logic via subclasses:

| Class name | Description |
|---|---|
| DateOffset | Generic offset class, defaults to 1 calendar day |
| BDay | business day (weekday) |
| Week | one week, optionally anchored on a day of the week |
| MonthEnd | calendar month end |
| BMonthEnd | business month end |
| QuarterEnd | calendar quarter end |
| BQuarterEnd | business quarter end |
| YearEnd | calendar year end |
| YearBegin | calendar year begin |
| BYearEnd | business year end |
| Hour | one hour |
| Minute | one minute |
| Second | one second |

The basic `DateOffset` takes the same arguments as `dateutil.relativedelta`, which works like:

```
In [847]: d = datetime(2008, 8, 18)

In [848]: d + relativedelta(months=4, days=5)
Out[848]: datetime.datetime(2008, 12, 23, 0, 0)
```

We could have done the same thing with `DateOffset`:

```
In [849]: from pandas.core.datetools import *

In [850]: d + DateOffset(months=4, days=5)
Out[850]: datetime.datetime(2008, 12, 23, 0, 0)
```

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 [851]: d - 5 * BDay()
Out[851]: datetime.datetime(2008, 8, 11, 0, 0)

In [852]: d + BMonthEnd()
Out[852]: datetime.datetime(2008, 8, 29, 0, 0)
```

The `rollforward` and `rollback` methods do exactly what you would expect:

```
In [853]: d
Out[853]: datetime.datetime(2008, 8, 18, 0, 0)

In [854]: offset = BMonthEnd()

In [855]: offset.rollforward(d)
Out[855]: datetime.datetime(2008, 8, 29, 0, 0)

In [856]: offset.rollback(d)
Out[856]: datetime.datetime(2008, 7, 31, 0, 0)
```

It's definitely worth exploring the `pandas.core.datetools` module and the various docstrings for the classes.

### 13.1.1 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 [857]: d + Week()
Out[857]: datetime.datetime(2008, 8, 25, 0, 0)
```

```
In [858]: d + Week(weekday=4)
Out[858]: datetime.datetime(2008, 8, 22, 0, 0)

In [859]: (d + Week(weekday=4)).weekday()
Out[859]: 4
```

### 13.1.2 Time rules

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as *time rules*.

| Rule name | Description |
|---|---|
| WEEKDAY | business day frequency |
| EOM | business month end frequency |
| 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) |
| Q@JAN | quarterly frequency, starting January |
| Q@FEB | quarterly frequency, starting February |
| Q@MAR | quarterly frequency, starting March |
| A@DEC | annual frequency, year end (December) |
| A@JAN | annual frequency, anchored end of January |
| A@FEB | annual frequency, anchored end of February |
| A@MAR | annual frequency, anchored end of March |
| A@APR | annual frequency, anchored end of April |
| A@MAY | annual frequency, anchored end of May |
| A@JUN | annual frequency, anchored end of June |
| A@JUL | annual frequency, anchored end of July |
| A@AUG | annual frequency, anchored end of August |
| A@SEP | annual frequency, anchored end of September |
| A@OCT | annual frequency, anchored end of October |
| A@NOV | annual frequency, anchored end of November |

These can be used as arguments to `DateRange` and various other time series-related functions in pandas.

## 13.2 Generating date ranges (DateRange)

The `DateRange` class utilizes these offsets (and any ones that we might add) to generate fixed-frequency date ranges:

```
In [860]: start = datetime(2009, 1, 1)

In [861]: end = datetime(2010, 1, 1)

In [862]: rng = DateRange(start, end, offset=BDay())

In [863]: rng
Out[863]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessDay>, tzinfo: None
[2009-01-01 00:00:00, ..., 2010-01-01 00:00:00]
length: 262
```

```
In [864]: DateRange(start, end, offset=BMonthEnd())
Out[864]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessMonthEnd>, tzinfo: None
[2009-01-30 00:00:00, ..., 2009-12-31 00:00:00]
length: 12
```

**Business day frequency** is the default for `DateRange`. You can also strictly generate a `DateRange` of a certain length by providing either a start or end date and a `periods` argument:

```
In [865]: DateRange(start, periods=20)
Out[865]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessDay>, tzinfo: None
[2009-01-01 00:00:00, ..., 2009-01-28 00:00:00]
length: 20

In [866]: DateRange(end=end, periods=20)
Out[866]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessDay>, tzinfo: None
[2009-12-07 00:00:00, ..., 2010-01-01 00:00:00]
length: 20
```

The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

### 13.2.1 DateRange is a valid Index

One of the main uses for `DateRange` is as an index for pandas objects. When working with a lot of time series data, there are several reasons to use `DateRange` objects when possible:

- A large range of dates for various offsets are pre-computed and cached under the hood in order to make generating subsequent date ranges very fast (just have to grab a slice)
- Fast shifting using the `shift` method on pandas objects
- Unioning of overlapping DateRange objects with the same frequency is very fast (important for fast data alignment)

The `DateRange` is a valid index and can even be intelligent when doing slicing, etc.

```
In [867]: rng = DateRange(start, end, offset=BMonthEnd())

In [868]: ts = Series(randn(len(rng)), index=rng)

In [869]: ts.index
Out[869]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessMonthEnd>, tzinfo: None
[2009-01-30 00:00:00, ..., 2009-12-31 00:00:00]
length: 12

In [870]: ts[:5].index
Out[870]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessMonthEnd>, tzinfo: None
[2009-01-30 00:00:00, ..., 2009-05-29 00:00:00]
length: 5
```

```
In [871]: ts[::2].index
Out[871]:
<class 'pandas.core.daterange.DateRange'>
offset: <2 BusinessMonthEnds>, tzinfo: None
[2009-01-30 00:00:00, ..., 2009-11-30 00:00:00]
length: 6
```

More complicated fancy indexing will result in an `Index` that is no longer a `DateRange`, however:

```
In [872]: ts[[0, 2, 6]].index
Out[872]: Index([2009-01-30 00:00:00, 2009-03-31 00:00:00, 2009-07-31 00:00:00], dtype=object)
```

## 13.3 Time series-related instance methods

**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.

### 13.3.1 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 [873]: ts = ts[:5]

In [874]: ts.shift(1)
Out[874]:
2009-01-30         NaN
2009-02-27    0.469112
2009-03-31   -0.282863
2009-04-30   -1.509059
2009-05-29   -1.135632
```

The shift method accepts an `offset` argument which can accept a `DateOffset` class or other `timedelta`-like object or also a *time rule*:

```
In [875]: ts.shift(5, offset=datetools.bday)
Out[875]:
2009-02-06    0.469112
2009-03-06   -0.282863
2009-04-07   -1.509059
2009-05-07   -1.135632
2009-06-05    1.212112

In [876]: ts.shift(5, offset='EOM')
Out[876]:
2009-06-30    0.469112
2009-07-31   -0.282863
2009-08-31   -1.509059
```

```
2009-09-30   -1.135632
2009-10-30    1.212112
```

### 13.3.2 Frequency conversion

The primary function for changing frequencies is the `asfreq` function. This is basically just a thin, but convenient wrapper around `reindex` which generates a `DateRange` and calls `reindex`.

```
In [877]: dr = DateRange('1/1/2010', periods=3, offset=3 * datetools.bday)

In [878]: ts = Series(randn(3), index=dr)

In [879]: ts
Out[879]:
2010-01-01    0.721555
2010-01-06   -0.706771
2010-01-11   -1.039575

In [880]: ts.asfreq(BDay())
Out[880]:
2010-01-01    0.721555
2010-01-04         NaN
2010-01-05         NaN
2010-01-06   -0.706771
2010-01-07         NaN
2010-01-08         NaN
2010-01-11   -1.039575

In [881]: ts.asfreq(BDay(), method='pad')
Out[881]:
2010-01-01    0.721555
2010-01-04    0.721555
2010-01-05    0.721555
2010-01-06   -0.706771
2010-01-07   -0.706771
2010-01-08   -0.706771
2010-01-11   -1.039575
```

### 13.3.3 Filling forward / backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the *missing data section*.

## 13.4 Up- and downsampling

We plan to add some efficient methods for doing resampling during frequency conversion. For example, converting secondly data into 5-minutely data. This is extremely common in, but not limited to, financial applications.

Until then, your best bet is a clever (or kludgy, depending on your point of view) application of GroupBy. Carry out the following steps:

1. Generate the target `DateRange` of interest

```
dr1hour = DateRange(start, end, offset=Hour())
dr5day = DateRange(start, end, offset=5 * datetools.day)
dr10day = DateRange(start, end, offset=10 * datetools.day)
```

2. Use the `asof` function ("as of") of the DateRange to do a groupby expression

```
grouped = data.groupby(dr5day.asof)
means = grouped.mean()
```

Here is a fully-worked example:

```
# some minutely data
In [882]: minutely = DateRange('1/3/2000 00:00:00', '1/3/2000 12:00:00',
   .....:                      offset=datetools.Minute())

In [883]: ts = Series(randn(len(minutely)), index=minutely)

In [884]: ts.index
Out[884]:
<class 'pandas.core.daterange.DateRange'>
offset: <1 Minute>, tzinfo: None
[2000-01-03 00:00:00, ..., 2000-01-03 12:00:00]
length: 721

In [885]: hourly = DateRange('1/3/2000', '1/4/2000', offset=datetools.Hour())

In [886]: grouped = ts.groupby(hourly.asof)

In [887]: grouped.mean()
Out[887]:
key_0
2000-01-03 00:00:00   -0.119068
2000-01-03 01:00:00    0.020282
2000-01-03 02:00:00    0.102562
2000-01-03 03:00:00   -0.106713
2000-01-03 04:00:00   -0.128935
2000-01-03 05:00:00   -0.146319
2000-01-03 06:00:00   -0.002938
2000-01-03 07:00:00   -0.131361
2000-01-03 08:00:00   -0.005749
2000-01-03 09:00:00   -0.399136
2000-01-03 10:00:00    0.097238
2000-01-03 11:00:00   -0.127307
2000-01-03 12:00:00   -0.273955
```

Some things to note about this method:

- This is rather inefficient because we haven't exploited the orderedness of the data at all. Calling the `asof` function on every date in the minutely time series is not strictly necessary. We'll be writing some significantly more efficient methods in the near future
- The dates in the result mark the **beginning of the period**. Be careful about which convention you use; you don't want to end up misaligning data because you used the wrong upsampling convention

CHAPTER

FOURTEEN

# PLOTTING WITH MATPLOTLIB

**Note:** We intend to build more plotting integration with matplotlib as time goes on.

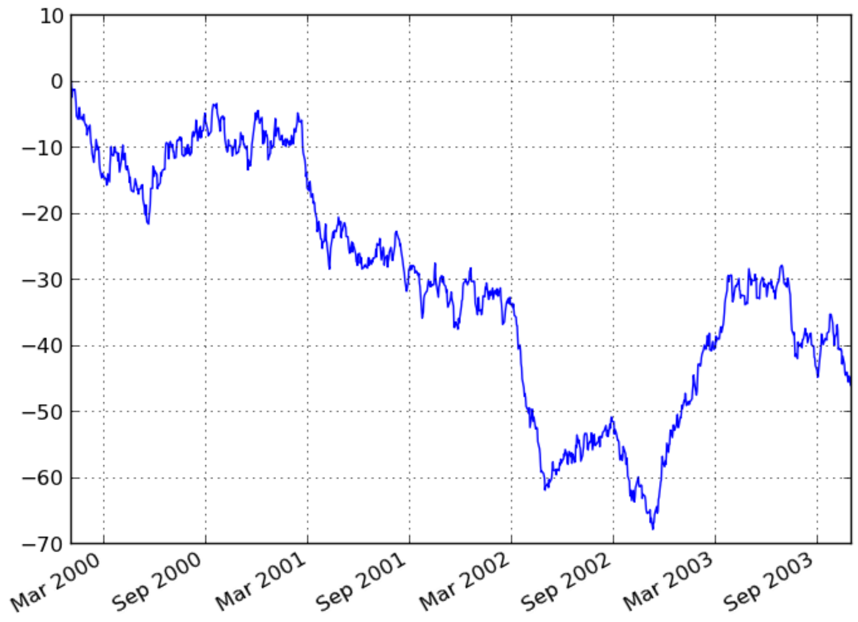
We use the standard convention for referencing the matplotlib API:

```
In [888]: import matplotlib.pyplot as plt
```

## 14.1 Basic plotting: `plot`

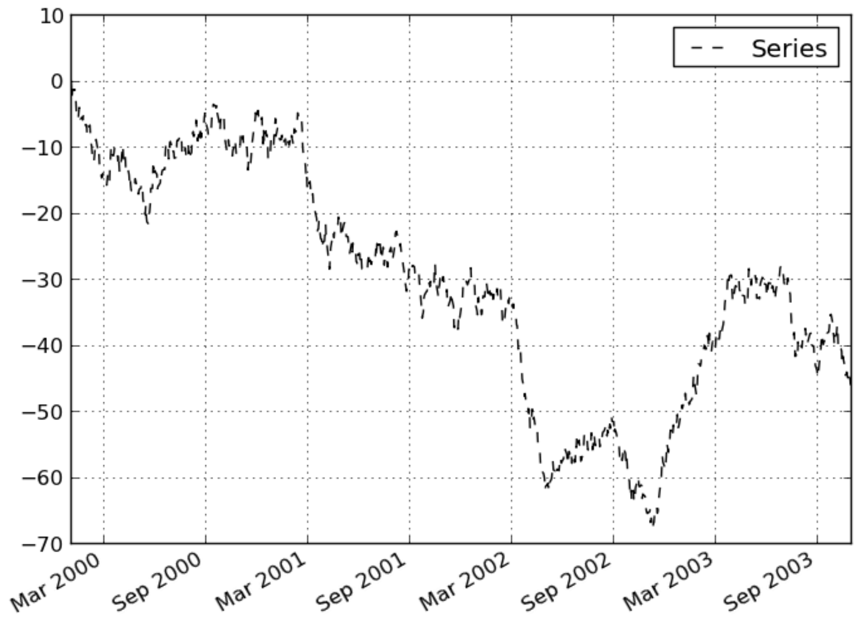
The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot`:

```
In [889]: ts = Series(randn(1000), index=DateRange('1/1/2000', periods=1000))

In [890]: ts = ts.cumsum()

In [891]: ts.plot()
Out[891]: <matplotlib.axes.AxesSubplot at 0x115ba4410>
```



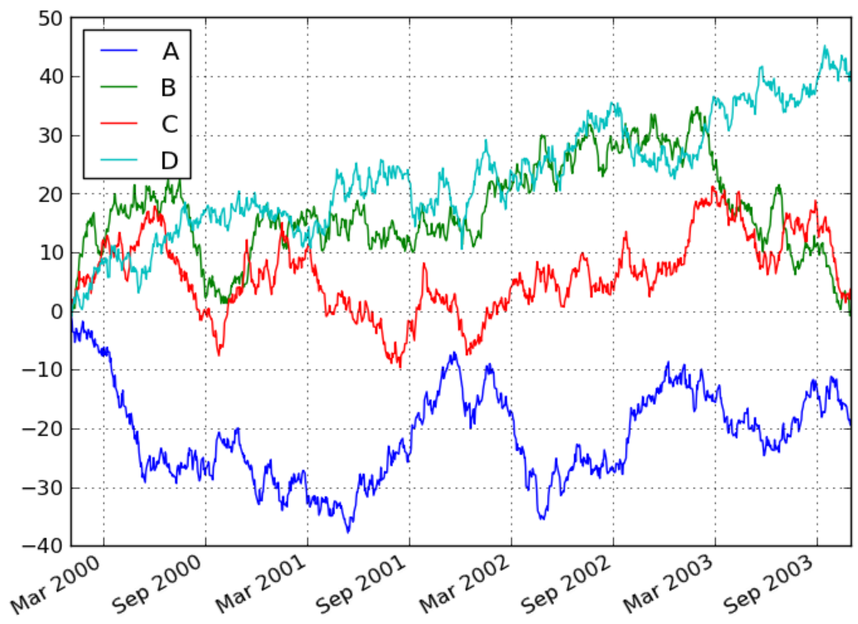
If the index consists of dates, it calls `gca().autofmt_xdate()` to try to format the x-axis nicely as per above. The method takes a number of arguments for controlling the look of the plot:

```
In [892]: plt.figure(); ts.plot(style='k--', label='Series'); plt.legend()
Out[892]: <matplotlib.legend.Legend at 0x115c07650>
```



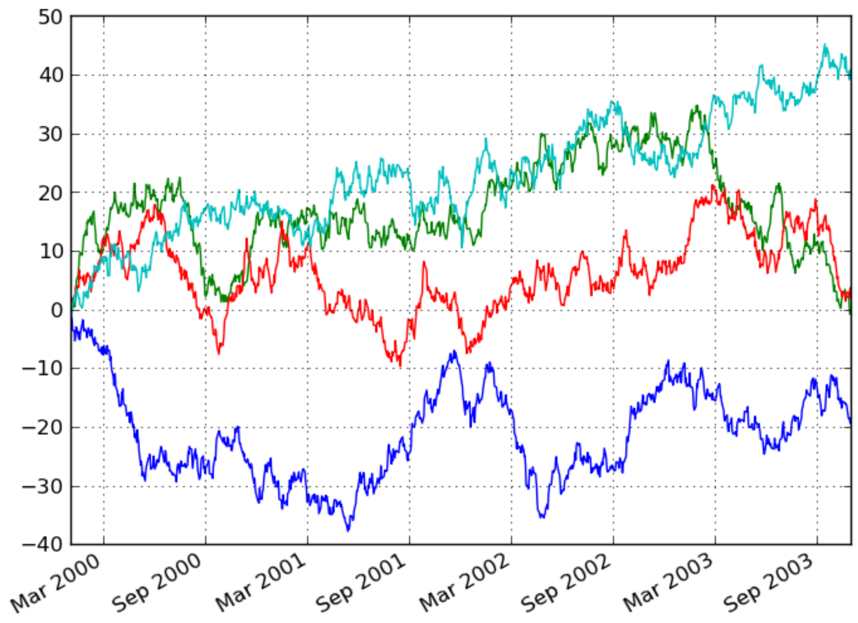
On DataFrame, `plot` is a convenience to plot all of the columns with labels:

```
In [893]: df = DataFrame(randn(1000, 4), index=ts.index,
   .....:                columns=['A', 'B', 'C', 'D'])

In [894]: df = df.cumsum()

In [895]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[895]: <matplotlib.legend.Legend at 0x115aab890>
```



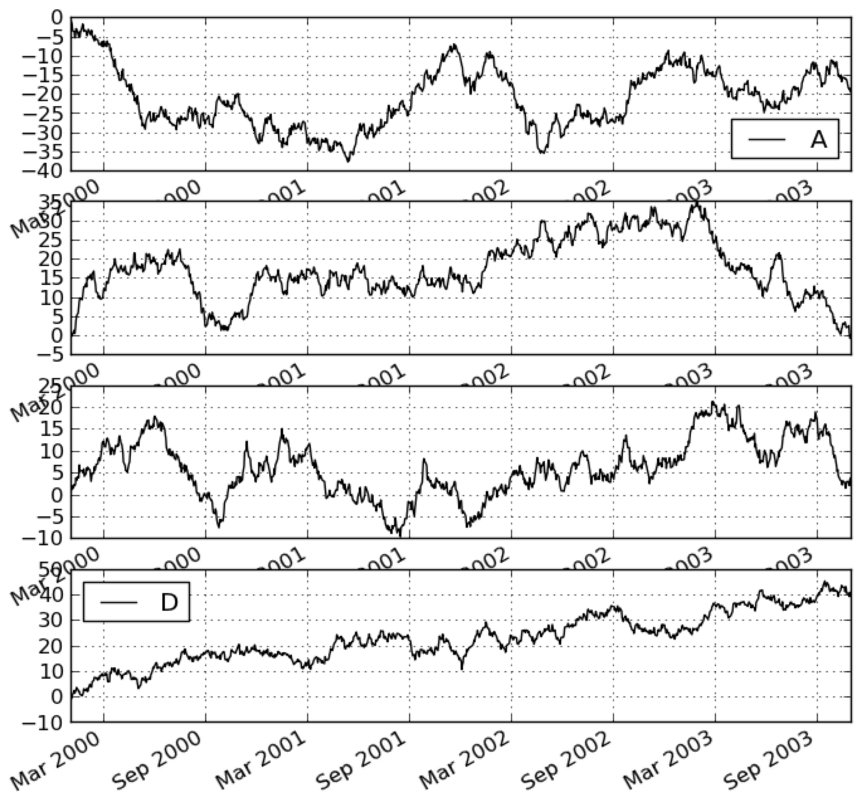
You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```
In [896]: df.plot(legend=False)
Out[896]: <matplotlib.axes.AxesSubplot at 0x115a69a90>
```



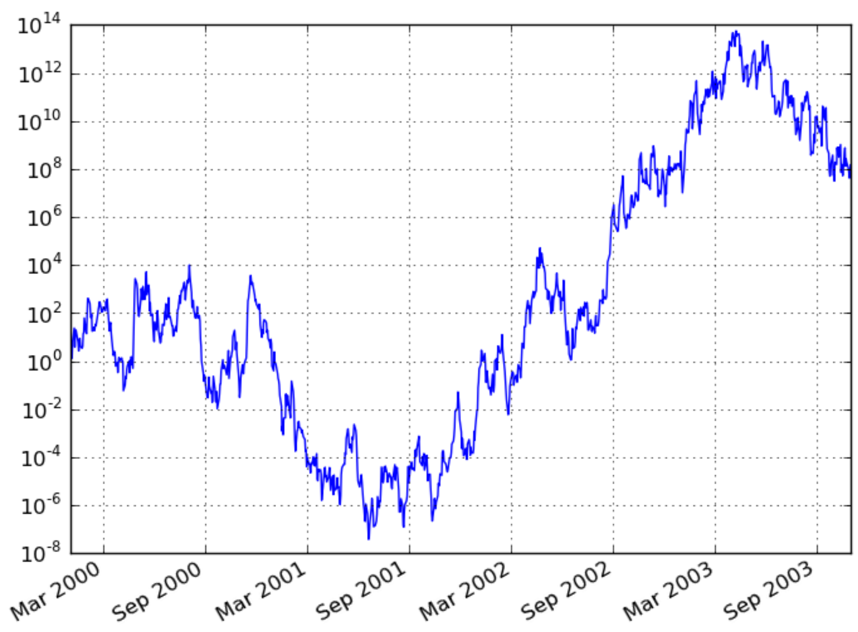
Some other options are available, like plotting each Series on a different axis:

```
In [897]: df.plot(subplots=True, figsize=(8, 8)); plt.legend(loc='best')
Out[897]: <matplotlib.legend.Legend at 0x115acd110>
```



You may pass `logy` to get a log-scale Y axis.

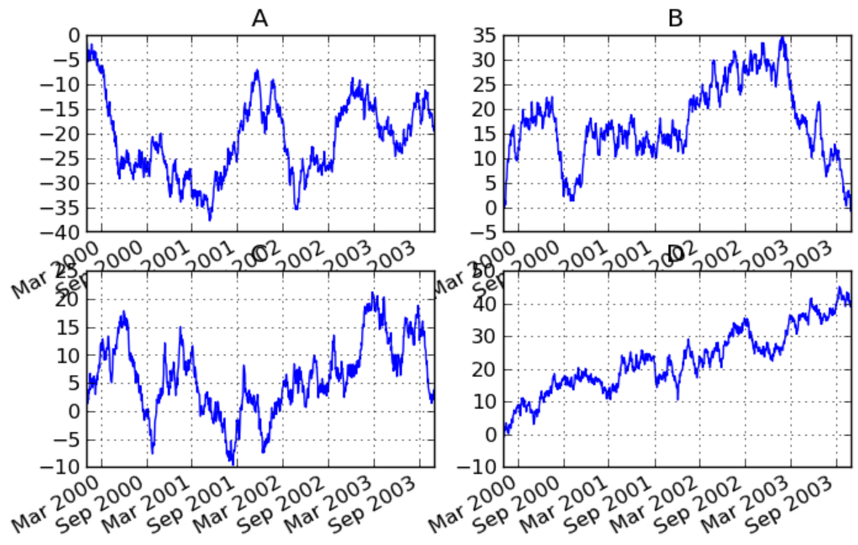
```
In [898]: plt.figure();
In [898]: ts = Series(randn(1000), index=DateRange('1/1/2000', periods=1000))

In [899]: ts = np.exp(ts.cumsum())

In [900]: ts.plot(logy=True)
Out[900]: <matplotlib.axes.AxesSubplot at 0x11970f990>
```



## 14.1.1 Targeting different subplots

You can pass an `ax` argument to `Series.plot` to plot on a particular axis:

```
In [901]: fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 5))

In [902]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A')
Out[902]: <matplotlib.text.Text at 0x11ae83a10>

In [903]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B')
Out[903]: <matplotlib.text.Text at 0x11babee50>

In [904]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C')
Out[904]: <matplotlib.text.Text at 0x11bad6e90>

In [905]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D')
Out[905]: <matplotlib.text.Text at 0x11bad8350>
```

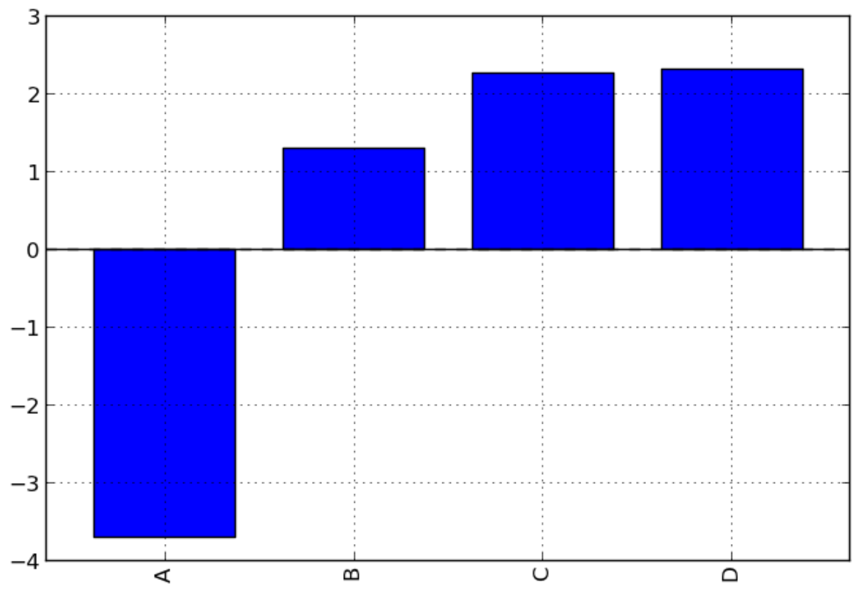


## 14.2 Other plotting features

### 14.2.1 Bar plots

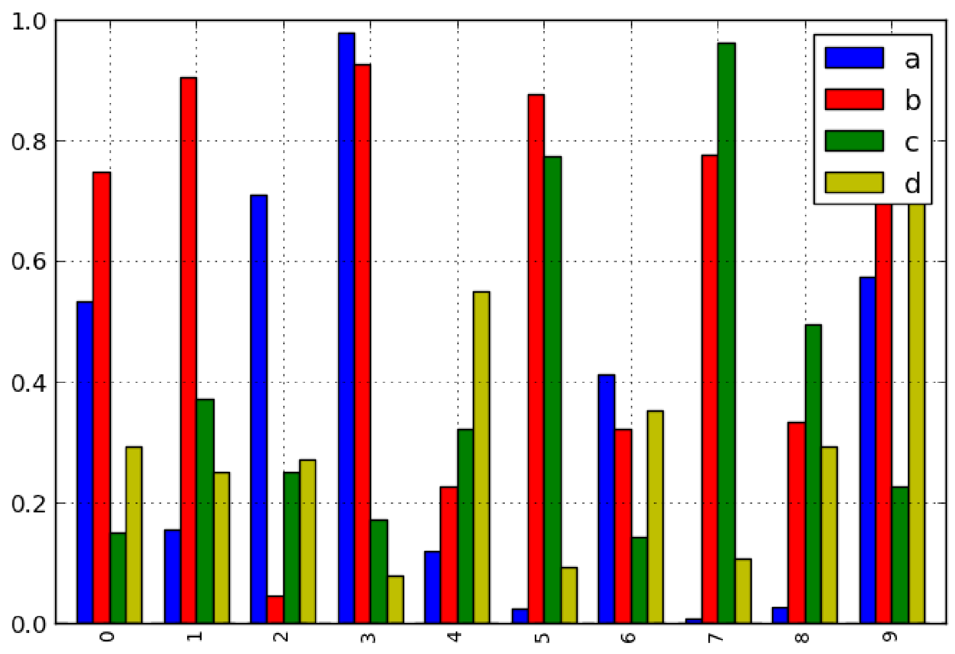
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [906]: plt.figure();
In [906]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
Out[906]: <matplotlib.lines.Line2D at 0x1197d3710>
```



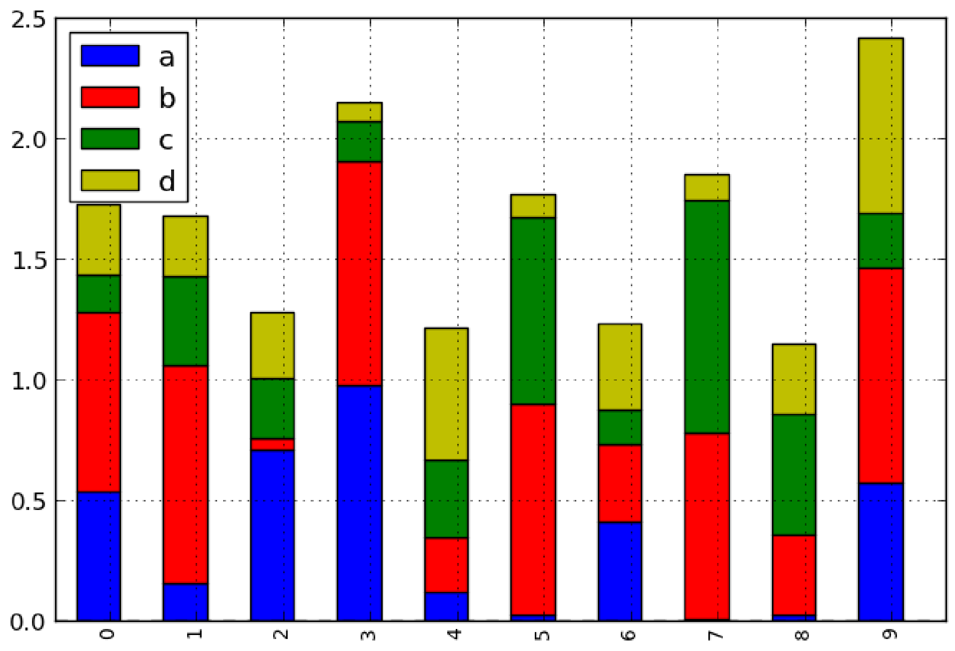
Calling a DataFrame's `plot` method with `kind='bar'` produces a multiple bar plot:

```
In [907]: df2 = DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [908]: df2.plot(kind='bar');
```



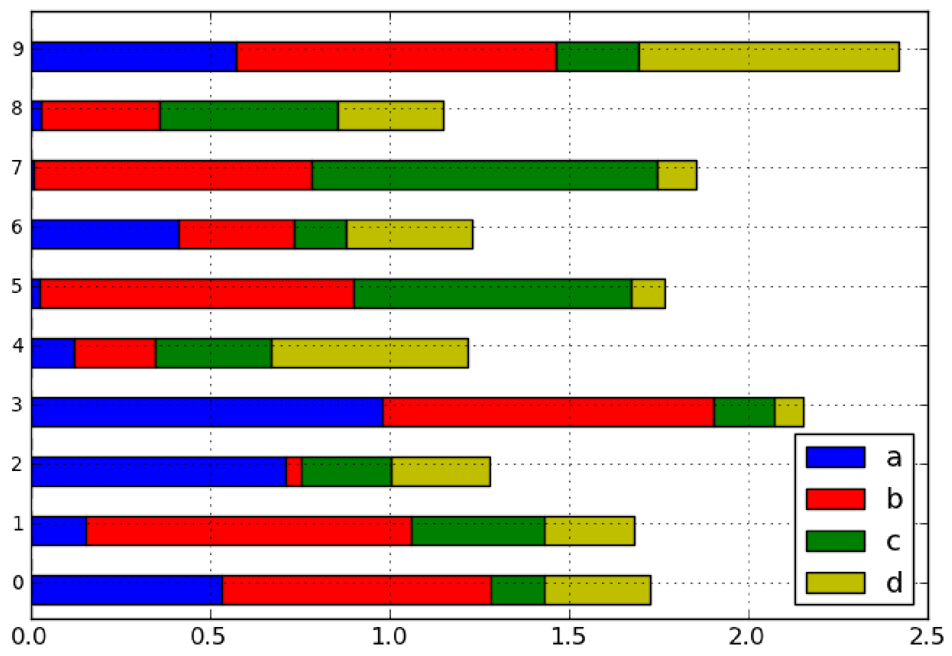
To produce a stacked bar plot, pass `stacked=True`:

```
In [908]: df2.plot(kind='bar', stacked=True);
```



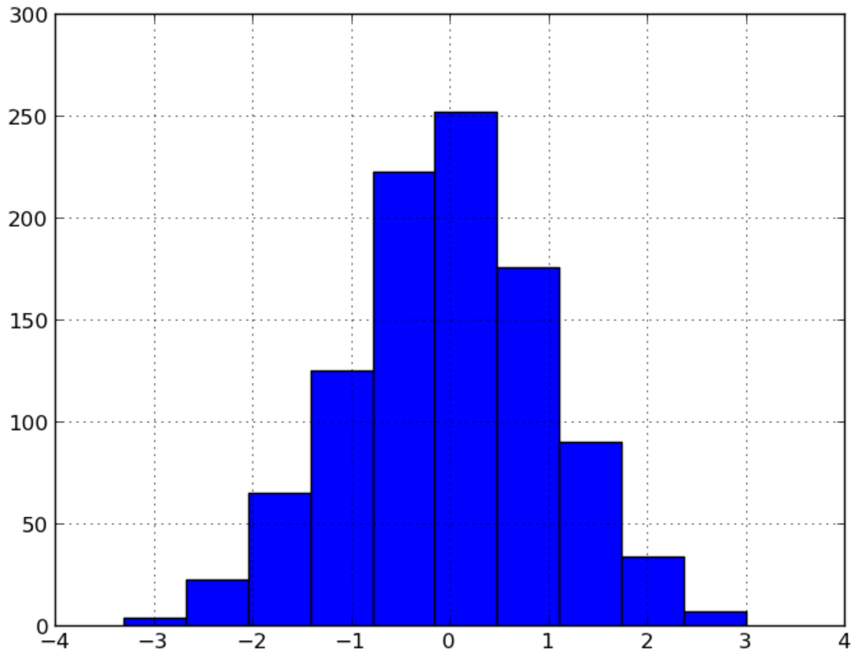
To get horizontal bar plots, pass `kind='barh'`:

```
In [908]: df2.plot(kind='barh', stacked=True);
```



## 14.2.2 Histograms

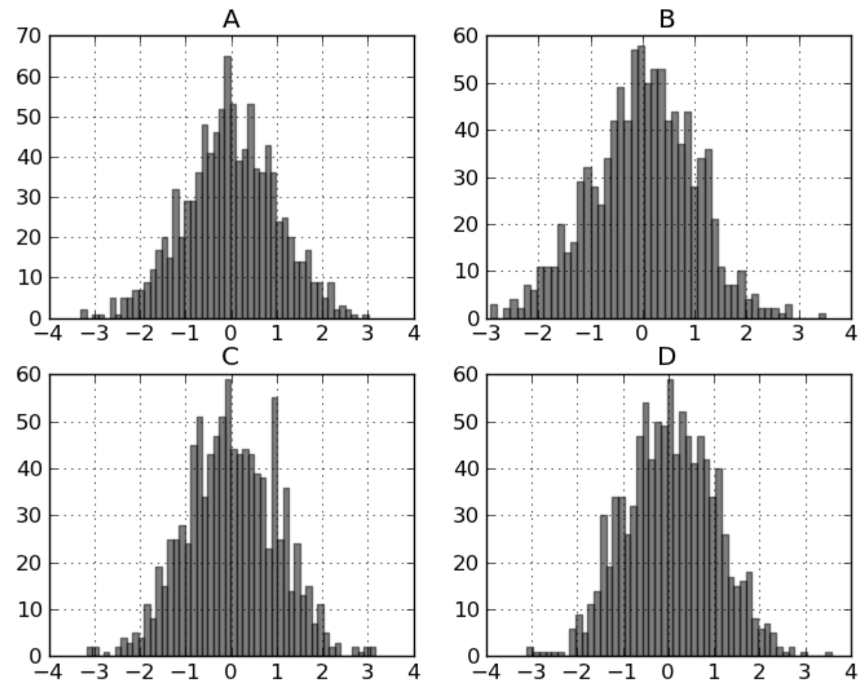
```
In [908]: plt.figure();
In [908]: df['A'].diff().hist()
Out[908]: <matplotlib.axes.AxesSubplot at 0x11ba46910>
```



For a DataFrame, `hist` plots the histograms of the columns on multiple subplots:

```
In [909]: plt.figure()
Out[909]: <matplotlib.figure.Figure at 0x11c05bb50>
```

```
In [910]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[910]:
array([[Axes(0.125,0.536364;0.352273x0.363636),
        Axes(0.547727,0.536364;0.352273x0.363636)],
       [Axes(0.125,0.1;0.352273x0.363636),
        Axes(0.547727,0.1;0.352273x0.363636)]], dtype=object)
```

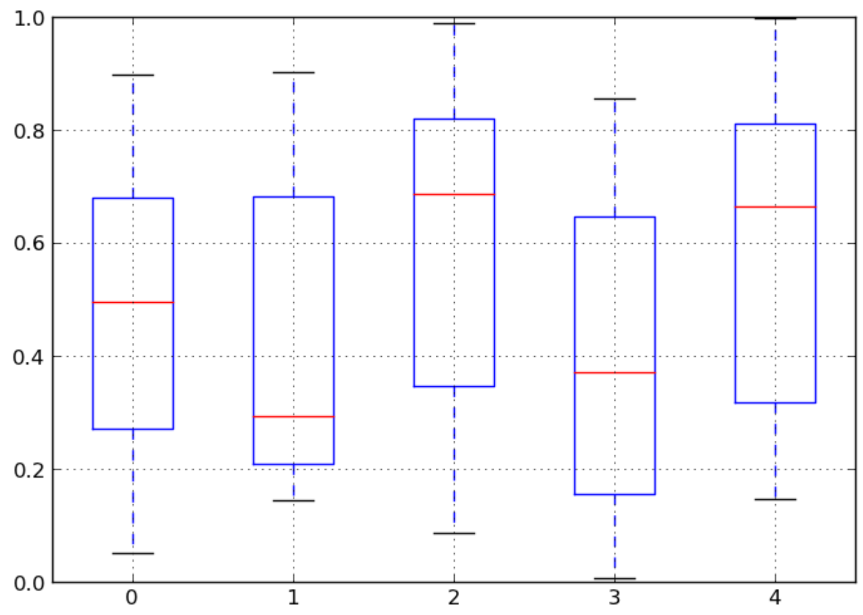


## 14.2.3 Box-Plotting

DataFrame has a `boxplot` method which allows you to visualize the distribution of values within each column.

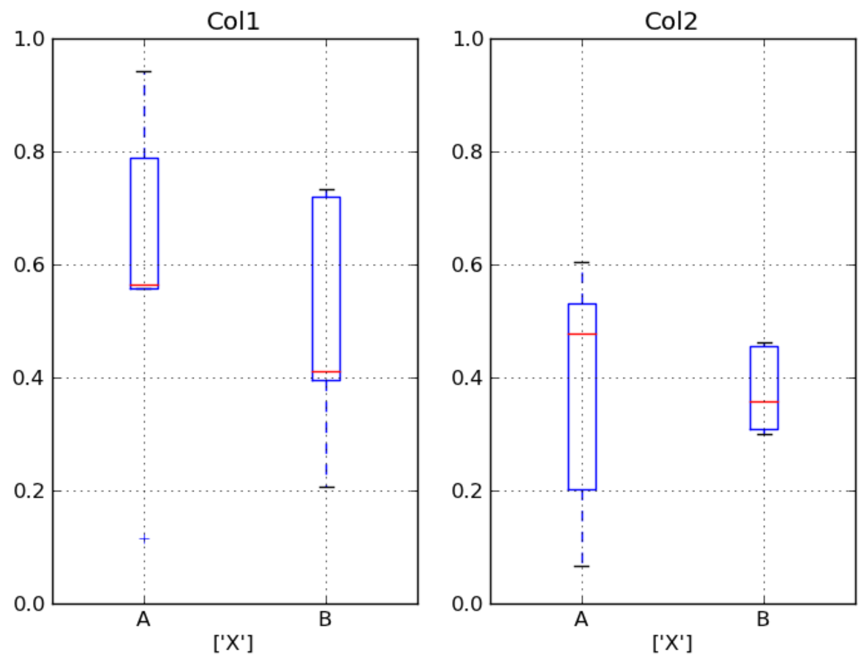
For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

```
In [911]: df = DataFrame(np.random.rand(10,5))

In [912]: plt.figure();
In [912]: bp = df.boxplot()
```



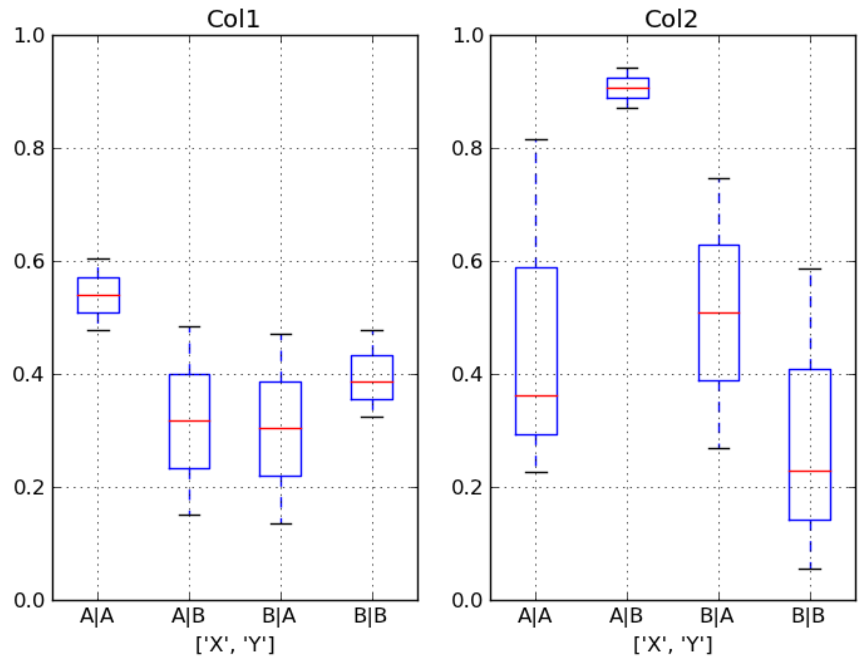
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [913]: df = DataFrame(np.random.rand(10,2), columns=['Col1', 'Col2'] )

In [914]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])

In [915]: plt.figure();
In [915]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

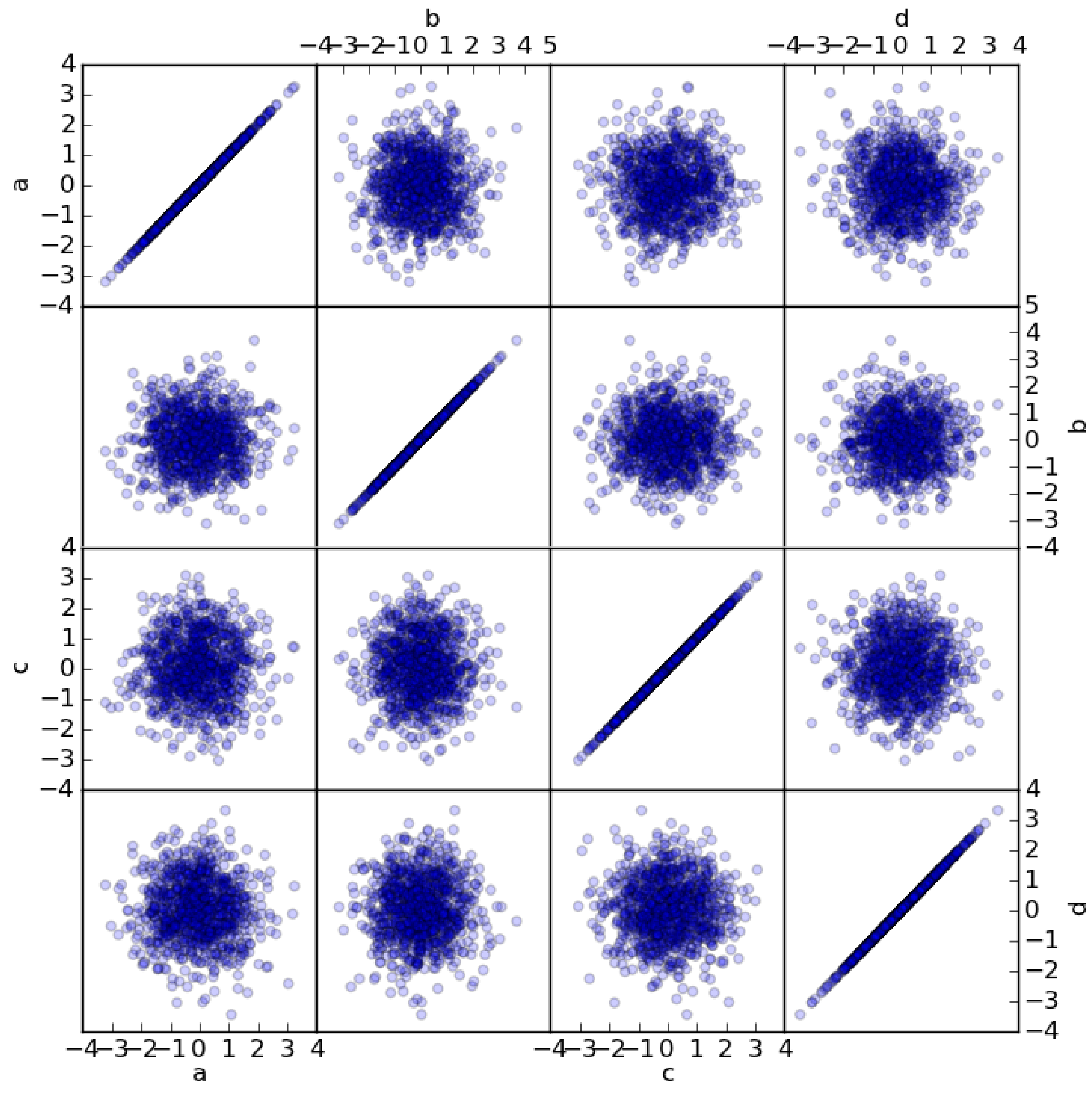
```
In [916]: df = DataFrame(np.random.rand(10,3), columns=['Col1', 'Col2', 'Col3'])

In [917]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])
In [918]: df['Y'] = Series(['A','B','A','B','A','B','A','B','A','B'])

In [919]: plt.figure();
In [919]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```



## 14.2.4 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 [920]: from pandas.tools.plotting import scatter_matrix

In [921]: df = DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [922]: scatter_matrix(df, alpha=0.2, figsize=(8, 8))
Out[922]:
array([[Axes(0.125,0.7;0.19375x0.2), Axes(0.31875,0.7;0.19375x0.2),
        Axes(0.5125,0.7;0.19375x0.2), Axes(0.70625,0.7;0.19375x0.2)],
       [Axes(0.125,0.5;0.19375x0.2), Axes(0.31875,0.5;0.19375x0.2),
        Axes(0.5125,0.5;0.19375x0.2), Axes(0.70625,0.5;0.19375x0.2)],
       [Axes(0.125,0.3;0.19375x0.2), Axes(0.31875,0.3;0.19375x0.2),
        Axes(0.5125,0.3;0.19375x0.2), Axes(0.70625,0.3;0.19375x0.2)],
       [Axes(0.125,0.1;0.19375x0.2), Axes(0.31875,0.1;0.19375x0.2),
        Axes(0.5125,0.1;0.19375x0.2), Axes(0.70625,0.1;0.19375x0.2)]], dtype=object)
```


a
b
c
d

CHAPTER

FIFTEEN

# IO TOOLS (TEXT, CSV, HDF5, ...)

## 15.1 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 described in the next section. 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 = read_clipboard(sep='\s*')
```

```
In [594]: clipdf
Out[594]:
   A  B  C
x  1  4  p
y  2  5  q
z  3  6  r
```

## 15.2 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. They can take a number of arguments:

- `path_or_buffer`: Either a string path to a file, 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 arbitrary whitespace.
- `header`: row number to use as the column names, and the start of the data. Defaults to 0 (first row); specify None if there is no header row.
- `names`: List of column names to use. If passed, header will be implicitly set to None.
- `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, or list of column numbers, 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.
- `parse_dates`: If True, attempt to parse the index column as dates. 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.
- `na_values`: optional list of strings to recognize as NaN (missing values), in addition to a default set.
- `nrows`: Number of rows to read out of the file. Useful to only read a small portion of a large file
- `chunksize`: An number of rows to be used to "chunk" a file into pieces. Will cause an `TextParser` object to be returned. More on this below in the section on *iterating and chunking*
- `iterator`: If True, return a `TextParser` to enable reading a file into memory piece by piece
- `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 if the contents are non-ascii
- `verbose` : show number of NA values inserted in non-numeric columns

Consider a typical CSV file containing, in this case, some time series data:

```
In [595]: 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 [596]: read_csv('foo.csv')
Out[596]:
       date  A  B  C
0  20090101  a  1  2
1  20090102  b  3  4
2  20090103  c  4  5
```

In the case of indexed data, you can pass the column number (or a list of column numbers, for a hierarchical index) you wish to use as the index. If the index values are dates and you want them to be converted to `datetime` objects, pass `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [597]: df = read_csv('foo.csv', index_col=0, parse_dates=True)

In [598]: df
Out[598]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5

# These are python datetime objects
In [599]: df.index
Out[599]: Index([2009-01-01 00:00:00, 2009-01-02 00:00:00, 2009-01-03 00:00:00], dtype=object)
```

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.

## 15.2.1 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 [from, to[
- `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 [600]: 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 [601]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [602]: df = read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [603]: df
Out[603]:
               X.2         X.3       X.4
X.1
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
#Widths are a list of integers
In [604]: widths = [6, 14, 13, 10]

In [605]: df = read_fwf('bar.csv', widths=widths, header=None)

In [606]: df
Out[606]:
      X.1         X.2         X.3       X.4
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

The parser will take care of extra white spaces around the columns so it's ok to have extra separation between the columns in the file.

### 15.2.2 Files with an "implicit" index column

Consider a file with one less entry in the header than the number of data column:

```
In [607]: 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 [608]: read_csv('foo.csv')
Out[608]:
          A  B  C
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Note that the dates weren't automatically parsed. In that case you would need to do as before:

```
In [609]: df = read_csv('foo.csv', parse_dates=True)

In [610]: df.index
Out[610]: Index([2009-01-01 00:00:00, 2009-01-02 00:00:00, 2009-01-03 00:00:00], dtype=object)
```

### 15.2.3 Reading DataFrame objects with `MultiIndex`

Suppose you have data indexed by two columns:

```
In [611]: print open('data/mindex_ex.csv').read()
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

The `index_col` argument to `read_csv` and `read_table` can take a list of column numbers to turn multiple columns into a `MultiIndex`:

```
In [612]: df = read_csv("data/mindex_ex.csv", index_col=[0,1])

In [613]: df
```

```
Out[613]:
                 zit   xit
year indiv
1977 A      1.20  0.60
     B      1.50  0.50
     C      1.70  0.80
1978 A      0.20  0.06
     B      0.70  0.20
     C      0.80  0.30
     D      0.90  0.50
     E      1.40  0.90
1979 C      0.20  0.15
     D      0.14  0.05
     E      0.50  0.15
     F      1.20  0.50
     G      3.40  1.90
     H      5.40  2.70
     I      6.40  1.20

In [614]: df.ix[1978]
Out[614]:
       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
```

### 15.2.4 Automatically "sniffing" the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files. YMMV, as pandas uses the Sniffer class of the csv module.

```
In [615]: print open('tmp2.sv').read()
year:indiv:zit:xit
1977:A:1.2:0.59999999999999998
1977:B:1.5:0.5
1977:C:1.7:0.80000000000000004
1978:A:0.20000000000000001:0.059999999999999998
1978:B:0.69999999999999996:0.20000000000000001
1978:C:0.80000000000000004:0.29999999999999999
1978:D:0.90000000000000002:0.5

In [616]: read_csv('tmp2.sv')
Out[616]:
                                year:indiv:zit:xit
0                  1977:A:1.2:0.59999999999999998
1                                 1977:B:1.5:0.5
2                  1977:C:1.7:0.80000000000000004
3  1978:A:0.20000000000000001:0.059999999999999998
4   1978:B:0.69999999999999996:0.20000000000000001
5   1978:C:0.80000000000000004:0.29999999999999999
6                  1978:D:0.90000000000000002:0.5
```

### 15.2.5 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 [617]: print open('tmp.sv').read()
year|indiv|zit|xit
1977|A|1.2|0.59999999999999998
1977|B|1.5|0.5
1977|C|1.7|0.80000000000000004
1978|A|0.20000000000000001|0.059999999999999998
1978|B|0.69999999999999996|0.20000000000000001
1978|C|0.80000000000000004|0.29999999999999999
1978|D|0.90000000000000002|0.5

In [618]: table = read_table('tmp.sv', sep='|')

In [619]: table
Out[619]:
   year indiv  zit   xit
0  1977     A  1.2  0.60
1  1977     B  1.5  0.50
2  1977     C  1.7  0.80
3  1978     A  0.2  0.06
4  1978     B  0.7  0.20
5  1978     C  0.8  0.30
6  1978     D  0.9  0.50
```

By specifiying a `chunksize` to `read_csv` or `read_table`, the return value will be an iterable object of type `TextParser`:

```
In [620]: reader = read_table('tmp.sv', sep='|', chunksize=4)

In [621]: reader
Out[621]: <pandas.io.parsers.TextParser at 0x1138bef10>

In [622]: for chunk in reader:
   .....:     print chunk
   .....:
   year indiv  zit   xit
0  1977     A  1.2  0.60
1  1977     B  1.5  0.50
2  1977     C  1.7  0.80
3  1978     A  0.2  0.06
   year indiv  zit  xit
0  1978     B  0.7  0.2
1  1978     C  0.8  0.3
2  1978     D  0.9  0.5
```

Specifying `iterator=True` will also return the `TextParser` object:

```
In [623]: reader = read_table('tmp.sv', sep='|', iterator=True)

In [624]: reader.get_chunk(5)
Out[624]:
   year indiv  zit   xit
0  1977     A  1.2  0.60
1  1977     B  1.5  0.50
2  1977     C  1.7  0.80
```

```
3  1978     A  0.2  0.06
4  1978     B  0.7  0.20
```

### 15.2.6 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 `nanRep`: 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

### 15.2.7 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, number of spaces to write between columns
- `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.

### 15.2.8 Writing to HTML format

DataFrame object has 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.

## 15.3 Excel files

The `ExcelFile` class can read an Excel 2003 file using the `xlrd` Python module and use the same parsing code as the above to convert tabular data into a DataFrame. To use it, create the `ExcelFile` object:

```
xls = ExcelFile('path_to_file.xls')
```

Then use the `parse` instance method with a sheetname, then use the same additional arguments as the parsers above:

```
xls.parse('Sheet1', index_col=None, na_values=['NA'])
```

To read sheets from an Excel 2007 file, you can pass a filename with a `.xlsx` extension, in which case the `openpyxl` module will be used to read the file.

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 `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 use the ExcelWriter class, as in the following example:

```
writer = ExcelWriter('path_to_file.xlsx')
df1.to_excel(writer, sheet_name='sheet1')
df2.to_excel(writer, sheet_name='sheet2')
writer.save()
```

## 15.4 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas to the high performance HDF5 format using the excellent PyTables library.

```
In [625]: store = HDFStore('store.h5')

In [626]: 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 [627]: index = DateRange('1/1/2000', periods=8)

In [628]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [629]: df = DataFrame(randn(8, 3), index=index,
   .....:                columns=['A', 'B', 'C'])

In [630]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:            major_axis=DateRange('1/1/2000', periods=5),
   .....:            minor_axis=['A', 'B', 'C', 'D'])

In [631]: store['s'] = s

In [632]: store['df'] = df

In [633]: store['wp'] = wp

In [634]: store
Out[634]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
df      DataFrame
s       Series
wp      Panel
```

In a current or later Python session, you can retrieve stored objects:

```
In [635]: store['df']
Out[635]:
                   A         B         C
2000-01-03 -0.173215  0.119209 -1.044236
2000-01-04 -0.861849 -2.104569 -0.494929
2000-01-05  1.071804  0.721555 -0.706771
2000-01-06 -1.039575  0.271860 -0.424972
2000-01-07  0.567020  0.276232 -1.087401
2000-01-10 -0.673690  0.113648 -1.478427
2000-01-11  0.524988  0.404705  0.577046
2000-01-12 -1.715002 -1.039268 -0.370647
```

CHAPTER

SIXTEEN

# 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 [824]: ts = Series(randn(10))

In [825]: ts[2:-2] = np.nan

In [826]: sts = ts.to_sparse()

In [827]: sts
Out[827]:
0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
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 [828]: ts.fillna(0).to_sparse(fill_value=0)
Out[828]:
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
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 [829]: df = DataFrame(randn(10000, 4))

In [830]: df.ix[:9998] = np.nan

In [831]: sdf = df.to_sparse()

In [832]: sdf
Out[832]:
<class 'pandas.sparse.frame.SparseDataFrame'>
Int64Index: 10000 entries, 0 to 9999
Columns: 4 entries, 0 to 3
dtypes: float64(4)

In [833]: sdf.density
Out[833]: 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 [834]: sts.to_dense()
Out[834]:
0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
```

## 16.1 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 [835]: arr = np.random.randn(10)

In [836]: arr[2:5] = np.nan; arr[7:8] = np.nan

In [837]: sparr = SparseArray(arr)

In [838]: sparr
Out[838]:
SparseArray([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
                nan,  0.606 ,  1.3342])
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 [839]: sparr.to_dense()
Out[839]:
array([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
           nan,  0.606 ,  1.3342])
```

## 16.2 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 [840]: spl = SparseList()

In [841]: spl
Out[841]:
<pandas.sparse.list.SparseList object at 0x10a7e64d0>
```

The two important methods are `append` and `to_array`. `append` can accept scalar values or any 1-dimensional sequence:

```
In [842]: spl.append(np.array([1., nan, nan, 2., 3.]))

In [843]: spl.append(5)

In [844]: spl.append(sparr)

In [845]: spl
Out[845]:
<pandas.sparse.list.SparseList object at 0x10a7e64d0>
SparseArray([ 1., nan, nan,  2.,  3.])
IntIndex
Indices: array([0, 3, 4], dtype=int32)
SparseArray([ 5.])
IntIndex
Indices: array([0], dtype=int32)
SparseArray([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
           nan,  0.606 ,  1.3342])
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 [846]: spl.to_array()
Out[846]:
SparseArray([ 1.    ,     nan,     nan,  2.    ,  3.    ,  5.    , -1.9557,
       -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,     nan,
        0.606 ,  1.3342])
IntIndex
Indices: array([ 0,  3,  4,  5,  6,  7, 11, 12, 14, 15], dtype=int32)
```

## 16.3 SparseIndex objects

Two kinds of `SparseIndex` are implemented, `block` and `integer`. We recommend using `block` as it's more memory efficient. The `integer` format keeps an arrays of all of the locations where the data are not equal to the fill value. The `block` format tracks only the locations and sizes of blocks of data.

CHAPTER

SEVENTEEN

# CAVEATS AND GOTCHAS

## 17.1 NaN, Integer NA values and NA type promotions

### 17.1.1 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.

### 17.1.2 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 [347]: s = Series([1, 2, 3, 4, 5], index=list('abcde'))

In [348]: s
Out[348]:
a    1
b    2
c    3
d    4
e    5

In [349]: s.dtype
Out[349]: dtype('int64')

In [350]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [351]: s2
Out[351]:
a     1
b     2
```

```
c      3
f    NaN
u    NaN

In [352]: s2.dtype
Out[352]: 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.

### 17.1.3 `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.

### 17.1.4 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 langauge R. Part of the reason is the NumPy type hierarchy:

| Typeclass | Dtypes |
|---|---|
| `numpy.floating` | `float16, float32, float64, float128` |
| `numpy.integer` | `int8, int16, int32, int64` |
| `numpy.unsignedinteger` | `uint8, uint16, uint32, uint64` |
| `numpy.object_` | `object_` |
| `numpy.bool_` | `bool_` |
| `numpy.character` | `string_, unicode_` |

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.

## 17.2 Integer indexing

## 17.3 Label-based slicing conventions

### 17.3.1 Non-monotonic indexes require exact matches

### 17.3.2 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 the "successor" or next element after a particular label in an index. For example, consider the following Series:

```
In [353]: s = Series(randn(6), index=list('abcdef'))

In [354]: s
Out[354]:
a    0.483368
b    0.186405
c   -1.439567
d   -0.503782
e    0.890769
f   -0.777798
```

Suppose we wished to slice from `c` to `e`, using integers this would be

```
In [355]: s[2:5]
Out[355]:
c   -1.439567
d   -0.503782
e    0.890769
```

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 [356]: s.ix['c':'e']
Out[356]:
c   -1.439567
d   -0.503782
e    0.890769
```

This is most definitely a "practicality beats purity" sort of thing, but it is something to watch out for is you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

## 17.4 Miscellaneous indexing gotchas

### 17.4.1 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 [357]: df = DataFrame(randn(6, 4), columns=['one', 'two', 'three', 'four'],
   .....:                index=list('abcdef'))

In [358]: df
Out[358]:
        one       two     three      four
a -0.552820 -1.428744  0.597468 -0.491240
b  1.281674 -0.099685 -1.823043 -0.779213
c -0.949327  0.768043 -0.054860 -1.493561
d  0.106004 -0.903513 -0.719875  0.301945
e  1.112546 -0.542770 -2.695540  0.431284
f -0.431092  1.666631  0.716659 -0.919717

In [359]: df.ix[['b', 'c', 'e']]
Out[359]:
        one       two     three      four
b  1.281674 -0.099685 -1.823043 -0.779213
c -0.949327  0.768043 -0.054860 -1.493561
e  1.112546 -0.542770 -2.695540  0.431284
```

This is, of course, completely equivalent *in this case* to using th `reindex` method:

```
In [360]: df.reindex(['b', 'c', 'e'])
Out[360]:
        one       two     three      four
b  1.281674 -0.099685 -1.823043 -0.779213
c -0.949327  0.768043 -0.054860 -1.493561
e  1.112546 -0.542770 -2.695540  0.431284
```

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 [361]: df.ix[[1, 2, 4]]
Out[361]:
        one       two     three      four
b  1.281674 -0.099685 -1.823043 -0.779213
c -0.949327  0.768043 -0.054860 -1.493561
e  1.112546 -0.542770 -2.695540  0.431284
```

If you pass `[1, 2, 4]` to `reindex` you will get another thing entirely:

```
In [362]: df.reindex([1, 2, 4])
Out[362]:
   one  two  three  four
1  NaN  NaN    NaN   NaN
2  NaN  NaN    NaN   NaN
4  NaN  NaN    NaN   NaN
```

So it's important to remember that `reindex` is **strict label indexing only**. This can lead to some potentially surprising results in pathological cases where an index contains, say, both integers and strings:

```
In [363]: s = Series([1, 2, 3], index=['a', 0, 1])

In [364]: s
Out[364]:
a    1
0    2
1    3

In [365]: s.ix[[0, 1]]
```

```
Out[365]:
a    1
0    2

In [366]: s.reindex([0, 1])
Out[366]:
0    2
1    3
```

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.

CHAPTER

EIGHTEEN

# 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 the current interface is designed for the 2.2.x series, and we recommend to use over other series unless you are prepared to fix parts of the code. Released packages are available in PyPi, but should the latest code in the 2.2.x series be wanted it can be obtained with:

```
# 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.

## 18.1 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 [773]: import pandas.rpy.common as com

In [774]: infert = com.load_data('infert')

In [775]: infert.head()
Out[775]:
  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
```

## 18.2 Calling R functions with pandas objects

## 18.3 High-level interface to R estimators

CHAPTER

NINETEEN

# RELATED PYTHON LIBRARIES

## 19.1 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

## 19.2 scikits.statsmodels

The main statistics and econometrics library for Python. pandas has become a dependency of this library.

## 19.3 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.

CHAPTER

TWENTY

# 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 it's many 3rd party libraries as they relate to pandas. In offering 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 or harder to use (you may have to be the judge of this given side-by-side code comparisons)

As I do not have an encyclopedic knowledge of R packages, feel free to suggest additional CRAN packages to add to this list. This is also here to offer a big of a translation guide for users of these R packages.

## 20.1 data.frame

## 20.2 zoo

## 20.3 xts

## 20.4 plyr

## 20.5 reshape / reshape2

CHAPTER

# TWENTYONE

# API REFERENCE

## 21.1 General functions

### 21.1.1 Data manipulations

| | |
|---|---|
| pivot_table(data[, values, rows, cols, ...]) | Create a spreadsheet-style pivot table as a DataFrame. The levels in the |

#### pandas.tools.pivot.pivot_table

pandas.tools.pivot.**pivot_table**(*data*, *values=None*, *rows=None*, *cols=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*)

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)

**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
```

| | |
|---|---|
| merge(left, right[, how, on, left_on, ...]) | Merge DataFrame objects by performing a database-style join operation by |
| concat(objs[, axis, join, join_axes, ...]) | Concatenate pandas objects along a particular axis with optional set logic along the other a |

## 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=True*, *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.

**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 True

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 True

Use the index from the right DataFrame as the join key. Same caveats as left_index

**sort** : boolean, default True

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.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 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.

**Returns** **concatenated** : type of objects

#### Notes

The keys, levels, and names arguments are all optional

### 21.1.2 Pickling

| | |
|---|---|
| `load`(path) | Load pickled pandas object (or any other pickled object) from the specified |
| `save`(obj, path) | Pickle (serialize) object to input file path |

#### pandas.core.common.load

`pandas.core.common.`**`load`**(*path*)

Load pickled pandas object (or any other pickled object) from the specified file path

**Parameters** **path** : string

File path

**Returns** **unpickled** : type of object stored in file

### pandas.core.common.save

`pandas.core.common.`**`save`**(*obj*, *path*)

Pickle (serialize) object to input file path

**Parameters** **obj** : any object

**path** : string

File path

## 21.1.3 File IO

| | |
|---|---|
| `read_table`(filepath_or_buffer[, sep, ...]) | Read general delimited file into DataFrame |
| `read_csv`(filepath_or_buffer[, sep, header, ...]) | Read CSV (comma-separated) file into DataFrame |
| `ExcelFile.parse`(sheetname[, header, ...]) | Read Excel table into DataFrame |

### pandas.io.parsers.read_table

`pandas.io.parsers.`**`read_table`**(*filepath_or_buffer*, *sep='\t'*, *header=0*, *index_col=None*, *names=None*, *skiprows=None*, *na_values=None*, *parse_dates=False*, *date_parser=None*, *nrows=None*, *iterator=False*, *chunksize=None*, *skip_footer=0*, *converters=None*, *verbose=False*, *delimiter=None*, *encoding=None*)

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://, 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

**header** : int, default 0

Row to use for the column labels of the parsed DataFrame

**skiprows** : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int)

**index_col** : int or sequence, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used.

**names** : array-like

List of column names

**na_values** : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

**parse_dates** : boolean, default False

Attempt to parse dates in the index column(s)

**date_parser** : function

Function to use for converting dates to strings. Defaults to dateutil.parser

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**iterator** : boolean, default False

Return TextParser object

**chunksize** : int, default None

Return TextParser object for iteration

**skip_footer** : 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

**encoding** : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

**Returns** **result** : DataFrame or TextParser

## pandas.io.parsers.read_csv

`pandas.io.parsers.`**`read_csv`**(*filepath_or_buffer*, *sep=', '*, *header=0*, *index_col=None*, *names=None*, *skiprows=None*, *na_values=None*, *parse_dates=False*, *date_parser=None*, *nrows=None*, *iterator=False*, *chunksize=None*, *skip_footer=0*, *converters=None*, *verbose=False*, *delimiter=None*, *encoding=None*)

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://, 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

**header** : int, default 0

Row to use for the column labels of the parsed DataFrame

**skiprows** : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int)

**index_col** : int or sequence, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used.

**names** : array-like

List of column names

**na_values** : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

**parse_dates** : boolean, default False

Attempt to parse dates in the index column(s)

**date_parser** : function

Function to use for converting dates to strings. Defaults to dateutil.parser

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**iterator** : boolean, default False

Return TextParser object

**chunksize** : int, default None

Return TextParser object for iteration

**skip_footer** : 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

**encoding** : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

**Returns** **result** : DataFrame or TextParser

### pandas.io.parsers.ExcelFile.parse

`ExcelFile.`**`parse`**(*sheetname*, *header=0*, *skiprows=None*, *index_col=None*, *parse_dates=False*, *date_parser=None*, *na_values=None*, *chunksize=None*)

Read Excel table into DataFrame

**Parameters** **sheetname** : string

Name of Excel sheet

**header** : int, default 0

Row to use for the column labels of the parsed DataFrame

**skiprows** : list-like

Row numbers 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

**na_values** : list-like, default None

List of additional strings to recognize as NA/NaN

**Returns** **parsed** : DataFrame

## 21.1.4 HDFStore: PyTables (HDF5)

| | |
|---|---|
| `HDFStore.put`(key, value[, table, append, ...]) | Store object in HDFStore |
| `HDFStore.get`(key) | Retrieve pandas object stored in file |

### pandas.io.pytables.HDFStore.put

`HDFStore.`**`put`**(*key*, *value*, *table=False*, *append=False*, *compression=None*)

Store object in HDFStore

**Parameters** **key** : object

**value** : {Series, DataFrame, Panel}

**table** : boolean, default False

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 data structures, append the input data to the existing table

**compression** : {None, ‘blosc’, ‘lzo’, ‘zlib’}, default None

Use a compression algorithm to compress the data If None, the compression settings specified in the ctor will be used.

### 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

## 21.1.5 Standard moving window functions

| | |
|---|---|
| `rolling_count`(arg, window[, time_rule]) | Rolling count of number of non-NaN observations inside provided window. |
| `rolling_sum`(arg, window[, min_periods, ...]) | Moving sum |

Continued on next page

Table 21.6 – continued from previous page

| | |
|---|---|
| `rolling_mean`(arg, window[, min_periods, ...]) | Moving mean |
| `rolling_median`(arg, window[, min_periods, ...]) | O(N log(window)) implementation using skip list |
| `rolling_var`(arg, window[, min_periods, ...]) | Unbiased moving variance |
| `rolling_std`(arg, window[, min_periods, ...]) | Unbiased moving standard deviation |
| `rolling_corr`(arg1, arg2, window[, ...]) | Moving sample correlation |
| `rolling_cov`(arg1, arg2, window[, ...]) | Unbiased moving covariance |
| `rolling_skew`(arg, window[, min_periods, ...]) | Unbiased moving skewness |
| `rolling_kurt`(arg, window[, min_periods, ...]) | Unbiased moving kurtosis |
| `rolling_apply`(arg, window, func[, ...]) | Generic moving function application |
| `rolling_quantile`(arg, window, quantile[, ...]) | Moving quantile |

## pandas.stats.moments.rolling_count

`pandas.stats.moments.`**`rolling_count`**(*arg*, *window*, *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

**Returns** **rolling_count** : type of caller

## pandas.stats.moments.rolling_sum

`pandas.stats.moments.`**`rolling_sum`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

## pandas.stats.moments.rolling_mean

`pandas.stats.moments.`**`rolling_mean`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_median

`pandas.stats.moments.`**`rolling_median`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_var

`pandas.stats.moments.`**`rolling_var`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_std

`pandas.stats.moments.`**`rolling_std`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_corr

`pandas.stats.moments.rolling_corr`(*arg1*, *arg2*, *window*, *min_periods=None*, *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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule 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.rolling_cov

`pandas.stats.moments.rolling_cov`(*arg1*, *arg2*, *window*, *min_periods=None*, *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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule 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.rolling_skew

`pandas.stats.moments.rolling_skew`(*arg*, *window*, *min_periods=None*, *time_rule=None*)
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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_kurt

`pandas.stats.moments.`**`rolling_kurt`**(*arg*, *window*, *min_periods=None*, *time_rule=None*)

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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule 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*, *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

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### pandas.stats.moments.rolling_quantile

`pandas.stats.moments.`**`rolling_quantile`**(*arg*, *window*, *quantile*, *min_periods=None*, *time_rule=None*)

Moving quantile

**Parameters** **arg** : Series, DataFrame

**window** : Number of observations used for calculating statistic

**quantile** : 0 <= quantile <= 1

**min_periods** : int

Minimum number of observations in window required to have a value

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default=None

Name of time rule to conform to before computing statistic

**Returns** **y** : type of input argument

### 21.1.6 Exponentially-weighted moving window functions

| | |
|---|---|
| `ewma`(arg[, com, span, min_periods, time_rule]) | Exponentially-weighted moving average |
| `ewmstd`(arg[, com, span, min_periods, bias, ...]) | Exponentially-weighted moving std |
| `ewmvar`(arg[, com, span, min_periods, bias, ...]) | Exponentially-weighted moving variance |
| `ewmcorr`(arg1, arg2[, com, span, ...]) | Exponentially-weighted moving correlation |
| `ewmcov`(arg1, arg2[, com, span, min_periods, ...]) | Exponentially-weighted moving covariance |

#### pandas.stats.moments.ewma

`pandas.stats.moments.`**`ewma`**(*arg*, *com=None*, *span=None*, *min_periods=0*, *time_rule=None*)

Exponentially-weighted moving average

**Parameters** **arg** : Series, DataFrame

**com** : float. optional

Center of mass: alpha = com / (1 + com),

**span** : float, optional

Specify decay in terms of span, alpha = 2 / (span + 1)

**min_periods** : int, default 0

Number of observations in sample to require (only affects beginning)

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default None

Name of time rule to conform to before computing statistic

**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 $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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*, *min_periods=0*, *bias=False*, *time_rule=None*)

Exponentially-weighted moving std

**Parameters** **arg** : Series, DataFrame

**com** : float. optional

Center of mass: alpha = com / (1 + com),

**span** : float, optional

Specify decay in terms of span, alpha = 2 / (span + 1)

**min_periods** : int, default 0

Number of observations in sample to require (only affects beginning)

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default None

Name of time rule to conform to before computing statistic

**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 $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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*, *min_periods=0*, *bias=False*, *time_rule=None*)

Exponentially-weighted moving variance

**Parameters** **arg** : Series, DataFrame

**com** : float. optional

Center of mass: alpha = com / (1 + com),

**span** : float, optional

Specify decay in terms of span, alpha = 2 / (span + 1)

**min_periods** : int, default 0

Number of observations in sample to require (only affects beginning)

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default None

Name of time rule to conform to before computing statistic

**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 $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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*, *min_periods=0*, *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 = com / (1 + com),

**span** : float, optional

Specify decay in terms of span, alpha = 2 / (span + 1)

**min_periods** : int, default 0

Number of observations in sample to require (only affects beginning)

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default None

Name of time rule to conform to before computing statistic

**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 $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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*, *min_periods=0*, *bias=False*, *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 = com / (1 + com),

**span** : float, optional

Specify decay in terms of span, alpha = 2 / (span + 1)

**min_periods** : int, default 0

Number of observations in sample to require (only affects beginning)

**time_rule** : {None, 'WEEKDAY', 'EOM', 'W@MON', ...}, default None

Name of time rule to conform to before computing statistic

**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 $\alpha = 1 - 2/(s+1) = c/(1+c)$

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.

## 21.2 Series

### 21.2.1 Attributes and underlying data

**Axes**

- **index**: axis labels

| | |
|---|---|
| `Series.values` | Return Series as ndarray |
| `Series.dtype` | Data-type of the array's elements. |
| `Series.isnull`(obj) | Replacement for numpy.isnan / -numpy.isfinite which is suitable for use on object arrays. |
| `Series.notnull`(obj) | Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays. |

#### pandas.Series.values

`Series.`**`values`**

Return Series as ndarray

**Returns** **arr** : numpy.ndarray

#### pandas.Series.dtype

`Series.`**`dtype`**

Data-type of the array's elements.

**Parameters** **None** :

**Returns** **d** : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

### pandas.Series.isnull

Series.**isnull**(*obj*)

Replacement for numpy.isnan / -numpy.isfinite which is suitable for use on object arrays.

**Parameters** **arr: ndarray or object value** :

**Returns** **boolean ndarray or boolean** :

### pandas.Series.notnull

Series.**notnull**(*obj*)

Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

**Parameters** **arr: ndarray or object value** :

**Returns** **boolean ndarray or boolean** :

## 21.2.2 Conversion / Constructors

| | |
|---|---|
| Series.__init__([data, index, dtype, name, copy]) | One-dimensional ndarray with axis labels (including time series). |
| Series.astype(dtype) | See numpy.ndarray.astype |
| Series.copy() | Return new Series with copy of underlying values |

### pandas.Series.__init__

Series.**__init__**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*)

One-dimensional ndarray with axis labels (including time series). Labels must be unique and can 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

**copy** : boolean, default False

### pandas.Series.astype

`Series.`**`astype`**`(`*dtype*`)`

See numpy.ndarray.astype

### pandas.Series.copy

`Series.`**`copy`**`()`

Return new Series with copy of underlying values

**Returns** **cp** : Series

## 21.2.3 Indexing, iteration

| | |
|---|---|
| `Series.get`(label[, default]) | Returns value occupying requested label, default to specified missing value if not present. |
| `Series.ix` | |
| `Series.__iter__`() | |
| `Series.iteritems`([index]) | 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.ix

`Series.`**`ix`**

### pandas.Series.__iter__

`Series.`**`__iter__`**`()`

#### pandas.Series.iteritems

`Series.`**`iteritems`**`(`*index=True*`)`
Lazily iterate over (index, value) tuples

### 21.2.4 Binary operator functions

| | |
|---|---|
| `Series.add`(other[, level, fill_value]) | Binary operator add with support to substitute a fill_value for missing data |
| `Series.div`(other[, level, fill_value]) | Binary operator divide with support to substitute a fill_value for missing data |
| `Series.mul`(other[, level, fill_value]) | Binary operator multiply with support to substitute a fill_value for missing data |
| `Series.sub`(other[, level, fill_value]) | Binary operator subtract 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 |

#### pandas.Series.add

`Series.`**`add`**`(`*other*, *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 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*`)`
Binary operator divide 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*`)`
Binary operator multiply 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*)

Binary operator subtract 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

Continued on next page

Table 21.12 – continued from previous page

## 21.2.5 Function application, GroupBy

| | |
|---|---|
| `Series.apply`(func) | Invoke function on values of Series. Can be ufunc or Python function |
| `Series.map`(arg) | Map values of Series using input correspondence (which can be |
| `Series.groupby`([by, axis, level, as_index, ...]) | Group series using mapper (dict or key function, apply given function |

### pandas.Series.apply

`Series.apply(func)`

Invoke function on values of Series. Can be ufunc or Python function expecting only single values

**Parameters** **func** : function

**Returns** **y** : Series

**See Also:**

**`Series.map`** For element-wise operations

### pandas.Series.map

`Series.map(arg)`

Map values of Series using input correspondence (which can be a dict, Series, or function)

**Parameters** **arg** : function, dict, or Series

**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)`

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

**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()

## 21.2.6 Computations / Descriptive Stats

| | |
|---|---|
| `Series.autocorr`() | Lag-1 autocorrelation |
| `Series.clip`([lower, upper, out]) | Trim values at input threshold(s) |
| `Series.clip_lower`(threshold) | Return copy of series with values below given value truncated |
| `Series.clip_upper`(threshold) | Return copy of series with values above given value truncated |
| `Series.corr`(other[, method]) | Compute correlation two Series, excluding missing values |
| `Series.count`([level]) | Return number of non-NA/null observations in the Series |
| `Series.cumprod`([axis, dtype, out, skipna]) | Cumulative product of values. |
| `Series.cumsum`([axis, dtype, out, skipna]) | Cumulative sum of values. |
| `Series.describe`([percentile_width]) | Generate various summary statistics of Series, excluding NaN |
| `Series.diff`([periods]) | 1st discrete difference of object |
| `Series.max`([axis, out, skipna, level]) | Return maximum of values |
| `Series.mean`([axis, dtype, out, skipna, level]) | Return mean of values |
| `Series.median`([skipna, level]) | Return median of values |
| `Series.min`([axis, out, skipna, level]) | Return minimum of values |
| `Series.prod`([axis, dtype, out, skipna, level]) | Return product of values |
| `Series.quantile`([q]) | Return value at the given quantile, a la scoreatpercentile in |
| `Series.skew`([skipna, level]) | Return unbiased skewness of values |
| `Series.std`([axis, dtype, out, ddof, skipna, ...]) | Return standard deviation of values |
| `Series.sum`([axis, dtype, out, skipna, level]) | Return sum of values |

Continued on next page

Table 21.13 – continued from previous page

| | |
|---|---|
| `Series.var`([axis, dtype, out, ddof, skipna, ...]) | Return variance of values |
| `Series.value_counts`() | Returns Series containing counts of unique values. The resulting Series |

### pandas.Series.autocorr

`Series.autocorr()`

Lag-1 autocorrelation

**Returns** **autocorr** : float

### 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 series with values below given value truncated

**Returns** **clipped** : Series

**See Also:**

`clip`

### pandas.Series.clip_upper

`Series.clip_upper`(*threshold*)

Return copy of series with values above given value truncated

**Returns** **clipped** : Series

**See Also:**

`clip`

### pandas.Series.corr

`Series.corr`(*other*, *method='pearson'*)

Compute correlation two Series, excluding missing values

**Parameters** **other** : Series

**method** : {'pearson', 'kendall', 'spearman'}

pearson : standard correlation coefficient kendall : Kendall Tau correlation coefficient spearman : Spearman rank correlation

**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.cumprod

`Series.cumprod(axis=0, dtype=None, out=None, skipna=True)`
Cumulative product of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**Returns** **cumprod** : Series

### pandas.Series.cumsum

`Series.cumsum(axis=0, dtype=None, out=None, skipna=True)`
Cumulative sum of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**Returns** **cumsum** : Series

### 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.max

`Series.max`(*axis=None*, *out=None*, *skipna=True*, *level=None*)

Return maximum of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **max** : float (or Series if level specified)

### pandas.Series.mean

`Series.mean`(*axis=0*, *dtype=None*, *out=None*, *skipna=True*, *level=None*)

Return mean of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Extra parameters are to preserve ndarrayinterface.** :

**Returns** **mean** : float (or Series if level specified)

### pandas.Series.median

`Series.median`(*skipna=True*, *level=None*)

Return median of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **median** : float (or Series if level specified)

### pandas.Series.min

`Series.min`(*axis=None*, *out=None*, *skipna=True*, *level=None*)

Return minimum of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **min** : float (or Series if level specified)

### pandas.Series.prod

`Series.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *skipna=True*, *level=None*)

Return product of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **product** : float (or Series if level specified)

### pandas.Series.quantile

`Series.`**`quantile`**(*q=0.5*)

Return value at the given quantile, a la scoreatpercentile in scipy.stats

**Parameters** **q** : quantile

0 <= q <= 1

**Returns** **quantile** : float

### pandas.Series.skew

`Series.`**`skew`**(*skipna=True*, *level=None*)

Return unbiased skewness of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **skew** : float (or Series if level specified)

### pandas.Series.std

`Series.`**`std`**(*axis=None*, *dtype=None*, *out=None*, *ddof=1*, *skipna=True*, *level=None*)

Return standard deviation of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **stdev** : float (or Series if level specified)

### pandas.Series.sum

`Series.`**`sum`**(*axis=0*, *dtype=None*, *out=None*, *skipna=True*, *level=None*)

Return sum of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Extra parameters are to preserve ndarrayinterface.** :

**Returns** **sum** : float (or Series if level specified)

### pandas.Series.var

`Series.`**`var`**(*axis=None*, *dtype=None*, *out=None*, *ddof=1*, *skipna=True*, *level=None*)

Return variance of values NA/null values are excluded

**Parameters** **skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns** **var** : float (or Series if level specified)

### pandas.Series.value_counts

`Series.`**`value_counts`**()

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

**Returns** **counts** : Series

## 21.2.7 Reindexing / Selection / Label manipulation

| | |
|---|---|
| `Series.align`(other[, join, level, copy, ...]) | Align two Series object with the specified join method |
| `Series.drop`(labels[, axis, level]) | Return new object with labels in requested axis removed |
| `Series.reindex`([index, method, level, ...]) | Conform Series to new index with optional filling logic, placing |
| `Series.reindex_like`(other[, method]) | Reindex Series to match index of another Series, optionally with |
| `Series.rename`(mapper) | Alter Series index using dict or function |
| `Series.select`(crit[, axis]) | Return data corresponding to axis labels matching criteria |

Continued on next page

**Table 21.14 – continued from previous page**

| | |
|---|---|
| `Series.take`(indices[, axis]) | Analogous to ndarray.take, return Series corresponding to requested |
| `Series.truncate`([before, after, copy]) | Function truncate a sorted DataFrame / Series before and/or after |

### pandas.Series.align

`Series.`**`align`**(*other*, *join='outer'*, *level=None*, *copy=True*, *fill_value=None*, *method=None*)

Align two Series object with the specified join method

**Parameters other** : Series

**join** : {'outer', 'inner', 'left', 'right'}, default 'outer'

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always return new objects. If copy=False and no reindexing is required, the same object will be returned (for better performance)

**fill_value** : object, default None

**method** : str, default 'pad'

**Returns (left, right)** : (Series, Series)

Aligned Series

### pandas.Series.drop

`Series.`**`drop`**(*labels*, *axis=0*, *level=None*)

Return new object with labels in requested axis removed

**Parameters labels** : array-like

**axis** : int

**level** : int or name, default None

For MultiIndex

**Returns dropped** : type of caller

### pandas.Series.reindex

`Series.`**`reindex`**(*index=None*, *method=None*, *level=None*, *fill_value=nan*, *copy=True*)

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 or Index

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**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 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

**Returns** **reindexed** : Series

## pandas.Series.reindex_like

`Series.`**`reindex_like`**`(`*other*, *method=None*`)`

Reindex Series to match index of another Series, optionally with filling logic

**Parameters** **other** : Series

**method** : string or None

See Series.reindex docstring

**Returns** **reindexed** : Series

### Notes

Like calling s.reindex(other.index, method=...)

## pandas.Series.rename

`Series.`**`rename`**`(`*mapper*`)`

Alter Series index using dict or function

**Parameters** **mapper** : dict-like or function

Transformation to apply to each index

**Returns** **renamed** : Series (new object)

### Notes

Function / dict values must be unique (1-to-1)

### Examples

```
>>> x
foo 1
bar 2
baz 3

>>> x.rename(str.upper)
FOO 1
BAR 2
BAZ 3
```

```
>>> x.rename({'foo' : 'a', 'bar' : 'b', 'baz' : 'c'})
a 1
b 2
c 3
```

### pandas.Series.select

`Series.select(crit, axis=0)`
Return data corresponding to axis labels matching criteria

**Parameters** **crit** : function

To be called on each index (label). Should return True or False

**axis** : int

**Returns** **selection** : type of caller

### pandas.Series.take

`Series.take(indices, axis=0)`
Analogous to ndarray.take, return Series corresponding to requested indices

**Parameters** **indices** : list / array of ints

**Returns** **taken** : Series

### pandas.Series.truncate

`Series.truncate(before=None, after=None, copy=True)`
Function truncate a sorted DataFrame / Series before and/or after some particular dates.

**Parameters** **before** : date

Truncate before date

**after** : date

Truncate after date

**Returns** **truncated** : type of caller

## 21.2.8 Missing data handling

| | |
|---|---|
| `Series.dropna`() | Return Series without null values |
| `Series.fillna`([value, method, inplace]) | Fill NA/NaN values using the specified method |
| `Series.interpolate`([method]) | Interpolate missing values (after the first valid value) |

### pandas.Series.dropna

`Series.dropna()`
Return Series without null values

**Returns** **valid** : Series

### pandas.Series.fillna

`Series.fillna(value=None, method='pad', inplace=False)`

Fill NA/NaN values using the specified method

**Parameters** **value** : any kind (should be same type as array)

Value to use to fill holes (e.g. 0)

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'

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

**inplace** : boolean, default False

If True, fill the Series in place. Note: this will modify any other views on this Series, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True

**Returns** **filled** : Series

**See Also:**

`reindex`, `asfreq`

### pandas.Series.interpolate

`Series.interpolate(method='linear')`

Interpolate missing values (after the first valid value)

**Parameters** **method** : {'linear', 'time'}

Interpolation method. Time interpolation works on daily and higher resolution data to interpolate given length of interval

**Returns** **interpolated** : Series

## 21.2.9 Reshaping, sorting

| | |
|---|---|
| `Series.argsort`([axis, kind, order]) | Overrides ndarray.argsort. |
| `Series.order`([na_last, ascending, kind]) | Sorts Series object, by value, maintaining index-value link |
| `Series.sort`([axis, kind, order]) | Sort values and index labels by value, in place. |
| `Series.sort_index`([ascending]) | Sort object by labels (along an axis) |
| `Series.sortlevel`([level, ascending]) | Sort Series with MultiIndex by chosen level. Data will be |
| `Series.unstack`([level]) | 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

### 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 algorith

**Returns** **y** : Series

### pandas.Series.sort

`Series.sort`(*axis=0*, *kind='quicksort'*, *order=None*)

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

### pandas.Series.sort_index

`Series.sort_index`(*ascending=True*)

Sort object by labels (along an axis)

**Parameters** **ascending** : boolean, default True

Sort ascending vs. descending

**Returns** **sorted_obj** : Series

### 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.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.
```

## 21.2.10 Combining / joining / merging

| | |
|---|---|
| Series.append(to_append) | Concatenate two or more Series. The indexes must not overlap |

### pandas.Series.append

Series.**append**(*to_append*)

Concatenate two or more Series. The indexes must not overlap

**Parameters** **to_append** : Series or list/tuple of Series

**Returns** **appended** : Series

## 21.2.11 Time series-related

| | |
|---|---|
| Series.asfreq(freq[, method]) | Convert this TimeSeries to the provided frequency using DateOffset |
| Series.asof(date) | Return last good (non-NaN) value in TimeSeries if value is NaN for |
| Series.shift(periods[, offset]) | Shift the index of the Series by desired number of periods with an |
| Series.first_valid_index() | Return label for first non-NA/null value |
| Series.last_valid_index() | Return label for last non-NA/null value |

Continued on next page

**Table 21.18 – continued from previous page**

| |
|---|
| `Series.weekday` |

### pandas.Series.asfreq

`Series.asfreq`(*freq*, *method=None*)

Convert this TimeSeries to the provided frequency using DateOffset object or time rule. Optionally provide fill method to pad/backfill missing values.

**Parameters** **offset** : DateOffset object, or string in {'WEEKDAY', 'EOM'}

DateOffset object or subclass (e.g. monthEnd)

**method** : {'backfill', 'pad', None}

Method to use for filling holes in new index

**Returns** **converted** : TimeSeries

### pandas.Series.asof

`Series.asof`(*date*)

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** **date** : datetime or similar value

**Returns** **value or NaN** :

#### Notes

Dates are assumed to be sorted

### pandas.Series.shift

`Series.shift`(*periods*, *offset=None*, ***kwds*)

Shift the index of the Series by desired number of periods with an optional time offset

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**offset** : DateOffset, timedelta, or time rule string, optional

Increment to use from datetools module or time rule (e.g. 'EOM')

**Returns** **shifted** : Series

### 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`**

## 21.2.12 Plotting

| | |
|---|---|
| `Series.hist`([ax, grid, xlabelsize, xrot, ...]) | Draw histogram of the input series using matplotlib |
| `Series.plot`(series[, label, kind, ...]) | Plot the input series with the index on the x-axis using matplotlib |

### pandas.Series.hist

`Series.`**`hist`**`(`*ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, ***kwds*`)`

Draw histogram of the input series using matplotlib

**Parameters** **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

**kwds** : keywords

To be passed to the actual plotting function

#### Notes

See matplotlib documentation online for more on this

### pandas.Series.plot

`Series.`**`plot`**`(`*series*, *label=None*, *kind='line'*, *use_index=True*, *rot=None*, *xticks=None*, *yticks=None*, *xlim=None*, *ylim=None*, *ax=None*, *style=None*, *grid=True*, *logy=False*, ***kwds*`)`

Plot the input series with the index on the x-axis using matplotlib

**Parameters** **label** : label argument to provide to plot

**kind** : {'line', 'bar'}

**rot** : int, default 30

Rotation for tick labels

**use_index** : boolean, default True

Plot index as axis tick labels

**ax** : matplotlib axis object

If not passed, uses gca()

**style** : string, default matplotlib default

matplotlib line style to use

**ax** : matplotlib axis object

If not passed, uses gca()

**kind** : {'line', 'bar', 'barh'}

bar : vertical bar plot barh : horizontal bar plot

**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

**kwds** : keywords

Options to pass to matplotlib plotting method

### Notes

See matplotlib documentation online for more on this subject

## 21.2.13 Serialization / IO / Conversion

| | |
|---|---|
| `Series.from_csv`(path[, sep, parse_dates, ...]) | Read delimited file into Series |
| `Series.load`(path) | |
| `Series.save`(path) | |
| `Series.to_csv`(path[, index, sep, na_rep, ...]) | Write Series to a comma-separated values (csv) file |
| `Series.to_dict`() | Convert Series to {label -> value} dict |

Continued on next page

Table 21.20 – continued from previous page

| | |
|---|---|
| `Series.to_sparse`([kind, fill_value]) | Convert Series to SparseSeries |

### 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

**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.load

**classmethod** `Series.`**`load`**(*path*)

### pandas.Series.save

`Series.`**`save`**(*path*)

### pandas.Series.to_csv

`Series.`**`to_csv`**(*path*, *index=True*, *sep=', '*, *na_rep=''*, *header=False*, *index_label=None*, *mode='w'*, *nanRep=None*, *encoding=None*)

Write Series to a comma-separated values (csv) file

**Parameters** **path** : string

File path

**nanRep** : string, default ''

Missing data rep'n

**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

### pandas.Series.to_dict

`Series.`**`to_dict`**`()`

Convert Series to {label -> value} dict

**Returns** **value_dict** : dict

### 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

## 21.3 DataFrame

### 21.3.1 Attributes and underlying data

**Axes**

- **index**: row labels
- **columns**: column labels

| | |
|---|---|
| `DataFrame.as_matrix`([columns]) | Convert the frame to its Numpy-array matrix representation. Columns |
| `DataFrame.dtypes` | |
| `DataFrame.get_dtype_counts`() | |
| `DataFrame.values` | Convert the frame to its Numpy-array matrix representation. Columns |
| `DataFrame.axes` | |
| `DataFrame.ndim` | |
| `DataFrame.shape` | |

### 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.

**Parameters** **columns** : array-like

Specific column order

**Returns** **values** : ndarray

If the DataFrame 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`**`()`

### pandas.DataFrame.values

`DataFrame.`**`values`**

Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

**Parameters** **columns** : array-like

Specific column order

**Returns** **values** : ndarray

If the DataFrame is heterogeneous and contains booleans or objects, the result will be of dtype=object

### pandas.DataFrame.axes

`DataFrame.`**`axes`**

### pandas.DataFrame.ndim

`DataFrame.`**`ndim`**

### pandas.DataFrame.shape

`DataFrame.`**`shape`**

## 21.3.2 Conversion / Constructors

| | |
|---|---|
| DataFrame.__init__([data, index, columns, ...]) | Two-dimensional size-mutable, potentially heterogeneous tabular data structu |
| DataFrame.astype(dtype) | Cast object to input numpy.dtype |
| DataFrame.copy([deep]) | Make a copy of this object |

## pandas.DataFrame.__init__

DataFrame.**__init__**(*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

Will default to np.arange(n) if not column labels 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

read_csv

### 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'])
```

## pandas.DataFrame.astype

DataFrame.**astype**(*dtype*)

Cast object to input numpy.dtype

**Parameters** **dtype** : numpy.dtype or Python type

**Returns** **casted** : type of caller

### 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

## 21.3.3 Indexing, iteration

| | |
|---|---|
| `DataFrame.ix` | |
| `DataFrame.insert`(loc, column, value) | Insert column into DataFrame at specified location. Raises Exception if |
| `DataFrame.__iter__`() | Iterate over columns of the frame. |
| `DataFrame.iteritems`() | Iterator over (column, series) pairs |
| `DataFrame.pop`(item) | Return column and drop from frame. |
| `DataFrame.xs`(key[, axis, level, copy]) | Returns a cross-section (row or column) from the DataFrame as a Series |

### pandas.DataFrame.ix

`DataFrame.ix`

### pandas.DataFrame.insert

`DataFrame.insert(loc, column, value)`

Insert column into DataFrame at specified location. Raises Exception if column is already contained in the DataFrame

**Parameters** **loc** : int

Must have 0 <= loc <= len(columns)

**column** : object

**value** : int, Series, or array-like

### pandas.DataFrame.__iter__

`DataFrame.__iter__()`

Iterate over columns of the frame.

### pandas.DataFrame.iteritems

`DataFrame.iteritems()`

Iterator over (column, series) pairs

### pandas.DataFrame.pop

`DataFrame.pop(item)`

Return column and drop from frame. Raise KeyError if not found.

**Returns** **column** : Series

### pandas.DataFrame.xs

`DataFrame.xs(key, axis=0, level=None, copy=True)`

Returns a cross-section (row or column) from the DataFrame as a Series object. Defaults to returning a row (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

**copy** : boolean, default True

Whether to make a copy of the data

**Returns** **xs** : Series

## 21.3.4 Binary operator functions

| | |
|---|---|
| `DataFrame.add`(other[, axis, level, fill_value]) | Binary operator add with support to substitute a fill_value for missing data in |
| `DataFrame.div`(other[, axis, level, fill_value]) | Binary operator divide with support to substitute a fill_value for missing data in |
| `DataFrame.mul`(other[, axis, level, fill_value]) | Binary operator multiply with support to substitute a fill_value for missing data |
| `DataFrame.sub`(other[, axis, level, fill_value]) | Binary operator subtract with support to substitute a fill_value for missing data i |
| `DataFrame.radd`(other[, axis, level, fill_value]) | Binary operator radd with support to substitute a fill_value for missing data in |
| `DataFrame.rdiv`(other[, axis, level, fill_value]) | Binary operator rdivide with support to substitute a fill_value for missing data i |
| `DataFrame.rmul`(other[, axis, level, fill_value]) | Binary operator rmultiply with support to substitute a fill_value for missing data |
| `DataFrame.rsub`(other[, axis, level, fill_value]) | Binary operator rsubtract with support to substitute a fill_value for missing data |
| `DataFrame.combine`(other, func[, fill_value]) | Add two DataFrame objects and do not propagate NaN values, so if for a |
| `DataFrame.combineAdd`(other) | Add two DataFrame objects and do not propagate |
| `DataFrame.combine_first`(other) | Combine two DataFrame objects and default to non-null values in frame |
| `DataFrame.combineMult`(other) | 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.div

`DataFrame.`**`div`**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Binary operator divide 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 multiply 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 subtract 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.rdiv

`DataFrame.`**`rdiv`**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Binary operator rdivide 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 rmultiply 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 rsubtract 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.combine

`DataFrame.`**`combine`**`(`*other*, *func*, *fill_value=None*`)`

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

**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 will be the union of the two indexes

**Parameters** **other** : DataFrame

**Returns** **combined** : DataFrame

### Examples

```
>>> a.combine_first(b)
    a's values prioritized, use values from b to fill holes
```

#### 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** :

### 21.3.5 Function application, GroupBy

| | |
|---|---|
| `DataFrame.apply`(func[, axis, broadcast, ...]) | Applies function along input axis of DataFrame. Objects passed to |
| `DataFrame.applymap`(func) | Apply a function to a DataFrame that is intended to operate |
| `DataFrame.groupby`([by, axis, level, ...]) | Group series using mapper (dict or key function, apply given function |

#### pandas.DataFrame.apply

`DataFrame.`**`apply`**(*func*, *axis=0*, *broadcast=False*, *raw=False*, *args=()*, ***kwds*)

Applies function along input axis of DataFrame. Objects passed to functions are Series objects having index either the DataFrame's index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates

**Parameters** **func** : function

Function to apply to each column

**axis** : {0, 1}

0 : apply function to each column 1 : apply function to each row

**broadcast** : bool, default False

For aggregation functions, return object of same size with values propagated

**raw** : boolean, default False

If False, convert each row or column into a Series. If raw=True the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance

**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

##### Notes

To apply a function elementwise, use applymap

##### 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

### pandas.DataFrame.groupby

`DataFrame.`**`groupby`**(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*)

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

**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()

### 21.3.6 Computations / Descriptive Stats

| | |
|---|---|
| `DataFrame.clip`([upper, lower]) | Trim values at input threshold(s) |
| `DataFrame.clip_lower`(threshold) | Trim values below threshold |
| `DataFrame.clip_upper`(threshold) | Trim values above threshold |
| `DataFrame.corr`([method]) | Compute pairwise correlation of columns, excluding NA/null values |
| `DataFrame.corrwith`(other[, axis, drop]) | Compute pairwise correlation between rows or columns of two DataFrame |
| `DataFrame.count`([axis, level, numeric_only]) | Return Series with number of non-NA/null observations over requested |
| `DataFrame.cumprod`([axis, skipna]) | Return cumulative product over requested axis as DataFrame |
| `DataFrame.cumsum`([axis, skipna]) | Return DataFrame of cumulative sums over requested axis. |
| `DataFrame.describe`([percentile_width]) | Generate various summary statistics of each column, excluding |
| `DataFrame.diff`([periods]) | 1st discrete difference of object |
| `DataFrame.mad`([axis, skipna, level]) | Return mean absolute deviation over requested axis. |
| `DataFrame.max`([axis, skipna, level]) | Return maximum over requested axis. |
| `DataFrame.mean`([axis, skipna, level]) | Return mean over requested axis. |
| `DataFrame.median`([axis, skipna, level]) | Return median over requested axis. |
| `DataFrame.min`([axis, skipna, level]) | Return minimum over requested axis. |
| `DataFrame.prod`([axis, skipna, level]) | Return product over requested axis. |
| `DataFrame.quantile`([q, axis]) | Return values at the given quantile over requested axis, a la |
| `DataFrame.skew`([axis, skipna, level]) | Return unbiased skewness over requested axis. |
| `DataFrame.sum`([axis, numeric_only, skipna, ...]) | Return sum over requested axis. |
| `DataFrame.std`([axis, skipna, level, ddof]) | Return standard deviation over requested axis. |
| `DataFrame.var`([axis, skipna, level, ddof]) | Return variance over requested axis. |

### pandas.DataFrame.clip

`DataFrame.`**`clip`**(*upper=None*, *lower=None*)

Trim values at input threshold(s)

**Parameters** **lower** : float, default None

**upper** : float, default None

**Returns** **clipped** : DataFrame

### pandas.DataFrame.clip_lower

`DataFrame.`**`clip_lower`**(*threshold*)

Trim values below threshold

**Returns** **clipped** : DataFrame

### pandas.DataFrame.clip_upper

`DataFrame.`**`clip_upper`**(*threshold*)

Trim values above threshold

**Returns** **clipped** : DataFrame

### pandas.DataFrame.corr

`DataFrame.`**`corr`**(*method='pearson'*)

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

**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.cumprod

`DataFrame.`**`cumprod`**(*axis=None*, *skipna=True*)

Return cumulative product over requested axis as DataFrame

**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** **y** : DataFrame

### pandas.DataFrame.cumsum

`DataFrame.`**`cumsum`**`(`*axis=None*, *skipna=True*`)`

Return DataFrame of cumulative sums over requested axis.

**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** **y** : DataFrame

### 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.mad

`DataFrame.`**`mad`**`(`*axis=0*, *skipna=True*, *level=None*`)`

Return mean absolute deviation 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **mad** : Series (or DataFrame if level specified)

### pandas.DataFrame.max

`DataFrame.`**`max`**(*axis=0*, *skipna=True*, *level=None*)

Return 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 NA

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **max** : Series (or DataFrame if level specified)

### pandas.DataFrame.mean

`DataFrame.`**`mean`**(*axis=0*, *skipna=True*, *level=None*)

Return mean 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **mean** : Series (or DataFrame if level specified)

### pandas.DataFrame.median

`DataFrame.`**`median`**(*axis=0*, *skipna=True*, *level=None*)

Return median 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **median** : Series (or DataFrame if level specified)

### pandas.DataFrame.min

`DataFrame.min(axis=0, skipna=True, level=None)`
Return 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **min** : Series (or DataFrame if level specified)

### pandas.DataFrame.prod

`DataFrame.prod(axis=0, skipna=True, level=None)`
Return product over requested axis. NA/null values are treated as 1

**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

**Returns** **product** : Series (or DataFrame if level specified)

### pandas.DataFrame.quantile

`DataFrame.quantile(q=0.5, axis=0)`
Return values at the given quantile over requested axis, a la scoreatpercentile in scipy.stats

**Parameters** **q** : quantile, default 0.5 (50% quantile)

0 <= q <= 1

**axis** : {0, 1}

0 for row-wise, 1 for column-wise

**Returns** **quantiles** : Series

### pandas.DataFrame.skew

`DataFrame.skew(axis=0, skipna=True, level=None)`
Return unbiased skewness 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **skew** : Series (or DataFrame if level specified)

### pandas.DataFrame.sum

`DataFrame.`**`sum`**(*axis=0*, *numeric_only=None*, *skipna=True*, *level=None*)

Return sum 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

**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** : Series (or DataFrame if level specified)

### pandas.DataFrame.std

`DataFrame.`**`std`**(*axis=0*, *skipna=True*, *level=None*, *ddof=1*)

Return standard deviation 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **std** : Series (or DataFrame if level specified)

### pandas.DataFrame.var

`DataFrame.`**`var`**(*axis=0*, *skipna=True*, *level=None*, *ddof=1*)

Return variance 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

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** **var** : Series (or DataFrame if level specified)

## 21.3.7 Reindexing / Selection / Label manipulation

| | |
|---|---|
| `DataFrame.add_prefix`(prefix) | Concatenate prefix string with panel items names. |
| `DataFrame.add_suffix`(suffix) | Concatenate suffix string with panel items names |
| `DataFrame.align`(other[, join, axis, level, ...]) | Align two DataFrame object on their index and columns with the |
| `DataFrame.drop`(labels[, axis, level]) | Return new object with labels in requested axis removed |
| `DataFrame.filter`([items, like, regex]) | Restrict frame's columns to set of items or wildcard |
| `DataFrame.reindex`([index, columns, method, ...]) | Conform DataFrame to new index with optional filling logic, placing |
| `DataFrame.reindex_like`(other[, method, copy]) | Reindex DataFrame to match indices of another DataFrame, optionally |
| `DataFrame.rename`([index, columns, copy]) | Alter index and / or columns using input function or functions. |
| `DataFrame.select`(crit[, axis]) | Return data corresponding to axis labels matching criteria |
| `DataFrame.take`(indices[, axis]) | Analogous to ndarray.take, return DataFrame corresponding to requested |
| `DataFrame.truncate`([before, after, copy]) | Function truncate a sorted DataFrame / Series before and/or after |
| `DataFrame.head`([n]) | Returns first n rows of DataFrame |
| `DataFrame.tail`([n]) | Returns last n rows of DataFrame |

### 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=nan*, *method=None*)

Align two DataFrame object on their index and columns with the specified join method for each axis Index

**Parameters** **other** : DataFrame or Series

**join** : {'outer', 'inner', 'left', 'right'}, default 'outer'

**axis** : {0, 1, None}, 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

**Returns** **(left, right)** : (DataFrame, type of other)

Aligned objects

### pandas.DataFrame.drop

`DataFrame.`**`drop`**(*labels*, *axis=0*, *level=None*)

Return new object with labels in requested axis removed

**Parameters** **labels** : array-like

**axis** : int

**level** : int or name, default None

For MultiIndex

**Returns** **dropped** : type of caller

### pandas.DataFrame.filter

`DataFrame.`**`filter`**(*items=None*, *like=None*, *regex=None*)

Restrict frame's columns to set of items or wildcard

**Parameters** **items** : list-like

List of columns to restrict to (must not all be present)

**like** : string

Keep columns where "arg in col == True"

**regex** : string (regular expression)

Keep columns with re.search(regex, col) == True

**Returns DataFrame with filtered columns** :

### Notes

Arguments are mutually exclusive, but this is not checked for

## pandas.DataFrame.reindex

`DataFrame.`**`reindex`**`(`*index=None*, *columns=None*, *method=None*, *level=None*, *fill_value=nan*, *copy=True*`)`

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** : array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**columns** : array-like, optional

Same usage as index argument

**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

**Returns** **reindexed** : same type as calling instance

### Examples

```
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

## pandas.DataFrame.reindex_like

`DataFrame.`**`reindex_like`**`(`*other*, *method=None*, *copy=True*`)`

Reindex DataFrame to match indices of another DataFrame, optionally with filling logic

**Parameters** **other** : DataFrame

**method** : string or None

**copy** : boolean, default True

**Returns** **reindexed** : DataFrame

### Notes

**Like calling s.reindex(index=other.index, columns=other.columns,** method=...)

## pandas.DataFrame.rename

`DataFrame.`**`rename`**`(`*index=None*, *columns=None*, *copy=True*`)`

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** **index** : dict-like or function, optional

Transformation to apply to index values

**columns** : dict-like or function, optional

Transformation to apply to column values

**copy** : boolean, default True

Also copy underlying data

**Returns** **renamed** : DataFrame (new object)

**See Also:**

`Series.rename`

## 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.take

`DataFrame.`**`take`**`(`*indices*, *axis=0*`)`

Analogous to ndarray.take, return DataFrame corresponding to requested indices along an axis

**Parameters** **indices** : list / array of ints

**axis** : {0, 1}

**Returns** **taken** : DataFrame

## pandas.DataFrame.truncate

`DataFrame.`**`truncate`**`(`*before=None*, *after=None*, *copy=True*`)`

Function truncate a sorted DataFrame / Series before and/or after some particular dates.

**Parameters** **before** : date

Truncate before date

**after** : date

Truncate after date

**Returns** **truncated** : type of caller

### pandas.DataFrame.head

`DataFrame.`**`head`**`(`*n=5*`)`

Returns first n rows of DataFrame

### pandas.DataFrame.tail

`DataFrame.`**`tail`**`(`*n=5*`)`

Returns last n rows of DataFrame

## 21.3.8 Missing data handling

| | |
|---|---|
| `DataFrame.dropna`([axis, how, thresh, subset]) | Return object with labels on given axis omitted where alternately any |
| `DataFrame.fillna`([value, method, axis, inplace]) | Fill NA/NaN values using the specified method |

### pandas.DataFrame.dropna

`DataFrame.`**`dropna`**`(`*axis=0*, *how='any'*, *thresh=None*, *subset=None*`)`

Return object with labels on given axis omitted where alternately any or all of the data are missing

**Parameters** **axis** : {0, 1}

**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

**Returns** **dropped** : DataFrame

### pandas.DataFrame.fillna

`DataFrame.`**`fillna`**`(`*value=None*, *method='pad'*, *axis=0*, *inplace=False*`)`

Fill NA/NaN values using the specified method

**Parameters** **method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'

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 or dict

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)

**axis** : {0, 1}, default 0

0: fill column-by-column 1: fill row-by-row

**inplace** : boolean, default False

If True, fill the DataFrame in place. Note: this will modify any other views on this DataFrame, like if you took a no-copy slice of an existing DataFrame, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True

**Returns** **filled** : DataFrame

**See Also:**

`reindex`, `asfreq`

### 21.3.9 Reshaping, sorting, transposing

| | |
|---|---|
| `DataFrame.sort_index`([axis, by, ascending]) | Sort DataFrame either by labels (along either axis) or by the values in |
| `DataFrame.delevel`(*args, **kwargs) | |
| `DataFrame.pivot`([index, columns, values]) | Reshape data (produce a "pivot" table) based on column values. |
| `DataFrame.sortlevel`([level, axis, ascending]) | Sort multilevel index by chosen axis and primary level. |
| `DataFrame.swaplevel`(i, j[, axis]) | Swap levels i and j in a MultiIndex on a particular axis |
| `DataFrame.stack`([level, dropna]) | Pivot a level of the (possibly hierarchical) column labels, returning a |
| `DataFrame.unstack`([level]) | Pivot a level of the (necessarily hierarchical) index labels, returning |
| `DataFrame.T` | Returns a DataFrame with the rows/columns switched. If the DataFrame is |
| `DataFrame.transpose`() | Returns a DataFrame with the rows/columns switched. If the DataFrame is |

#### pandas.DataFrame.sort_index

`DataFrame.`**`sort_index`**(*axis=0*, *by=None*, *ascending=True*)

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, default True

Sort ascending vs. descending

**Returns** **sorted** : DataFrame

#### 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.sortlevel

`DataFrame.`**`sortlevel`**(*level=0*, *axis=0*, *ascending=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

**Parameters** **level** : int

**axis** : {0, 1}

**ascending** : bool, default True

**Returns** **sorted** : DataFrame

## pandas.DataFrame.swaplevel

`DataFrame.`**`swaplevel`**`(`*i, j, axis=0*`)`

Swap levels i and j in a MultiIndex on a particular axis

**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 last level

Level(s) of index to unstack, can pass level name

**Returns** **unstacked** : DataFrame or Series

### 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.

>>> df = s.unstack(level=0)
>>> df
   one  two
a  1.   2.
b  3.   4.

>>> df.unstack()
one  a  1.
     b  3.
two  a  2.
     b  4.
```

### pandas.DataFrame.T

`DataFrame.T`
Returns a DataFrame with the rows/columns switched. If the DataFrame is homogeneously-typed, the data is not copied

### pandas.DataFrame.transpose

`DataFrame.transpose()`
Returns a DataFrame with the rows/columns switched. If the DataFrame is homogeneously-typed, the data is not copied

## 21.3.10 Combining / joining / merging

| | |
|---|---|
| `DataFrame.join`(other[, on, how, lsuffix, ...]) | Join columns with other DataFrame either on index or on a key |
| `DataFrame.merge`(right[, how, on, left_on, ...]) | Merge DataFrame objects by performing a database-style join operation by |
| `DataFrame.append`(other[, ignore_index, ...]) | Append columns of other to end of this frame's columns and index, returning a r |

### 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=True*, *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.

**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 True

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 True

Use the index from the right DataFrame as the join key. Same caveats as left_index

**sort** : boolean, default True

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.append

`DataFrame.append(other, ignore_index=False, verify_integrity=True)`

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

**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

### 21.3.11 Time series-related

| | |
|---|---|
| `DataFrame.asfreq`(freq[, method]) | Convert all TimeSeries inside to specified frequency using DateOffset |
| `DataFrame.shift`(periods[, offset]) | Shift the index of the DataFrame by desired number of periods with an |
| `DataFrame.first_valid_index`() | Return label for first non-NA/null value |
| `DataFrame.last_valid_index`() | Return label for last non-NA/null value |

#### pandas.DataFrame.asfreq

`DataFrame.`**`asfreq`**(*freq*, *method=None*)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters** **offset** : DateOffset object, or string in {'WEEKDAY', 'EOM'}

DateOffset object or subclass (e.g. monthEnd)

**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

**Returns** **converted** : DataFrame

#### pandas.DataFrame.shift

`DataFrame.`**`shift`**(*periods*, *offset=None*, ***kwds*)

Shift the index of the DataFrame by desired number of periods with an optional time offset

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**offset** : DateOffset, timedelta, or time rule string, optional

Increment to use from datetools module or time rule (e.g. 'EOM')

**Returns** **shifted** : DataFrame

#### 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

### 21.3.12 Plotting

| | |
|---|---|
| `DataFrame.hist`(data[, grid, xlabelsize, ...]) | Draw Histogram the DataFrame's series using matplotlib / pylab. |
| `DataFrame.plot`([frame, subplots, sharex, ...]) | Make line or bar plot of DataFrame's series with the index on the x-axis |

#### pandas.DataFrame.hist

`DataFrame.`**`hist`** (*data*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, ***kwds*)

Draw Histogram the DataFrame's series using matplotlib / pylab.

**Parameters** **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

**kwds** : other plotting keyword arguments

To be passed to hist function

#### pandas.DataFrame.plot

`DataFrame.`**`plot`** (*frame=None*, *subplots=False*, *sharex=True*, *sharey=False*, *use_index=True*, *figsize=None*, *grid=True*, *legend=True*, *rot=None*, *ax=None*, *title=None*, *xlim=None*, *ylim=None*, *logy=False*, *xticks=None*, *yticks=None*, *kind='line'*, *sort_columns=True*, *fontsize=None*, ***kwds*)

Make line or bar plot of DataFrame's series with the index on the x-axis using matplotlib / pylab.

**Parameters** **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 True** :

Sort column names to determine plot ordering

**title** : string

Title to use for the plot

**grid** : boolean, default True

Axis grid lines

**legend** : boolean, default True

Place legend on axis subplots

**ax** : matplotlib axis object, default None

**kind** : {'line', 'bar', 'barh'}

bar : vertical bar plot barh : horizontal bar plot

**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

**kwds** : keywords

Options to pass to matplotlib plotting method

**Returns** **ax_or_axes** : matplotlib.AxesSubplot or list of them

## 21.3.13 Serialization / IO / Conversion

| | |
|---|---|
| `DataFrame.from_csv`(path[, header, sep, ...]) | Read delimited file into DataFrame |
| `DataFrame.from_records`(data[, index, ...]) | Convert structured or record ndarray to DataFrame |
| `DataFrame.to_csv`(path_or_buf[, sep, na_rep, ...]) | Write DataFrame to a comma-separated values (csv) file |
| `DataFrame.to_excel`(excel_writer[, ...]) | Write DataFrame to a excel sheet |
| `DataFrame.to_dict`() | Convert DataFrame to nested dictionary |
| `DataFrame.to_records`([index]) | Convert DataFrame to record array. Index will be put in the |
| `DataFrame.to_sparse`([fill_value, kind]) | Convert to SparseDataFrame |
| `DataFrame.to_string`([buf, columns, ...]) | Render a DataFrame to a console-friendly tabular output. |
| `DataFrame.save`(path) | |
| `DataFrame.load`(path) | |
| `DataFrame.info`([verbose, buf]) | Concise summary of a DataFrame, used in __repr__ when very large. |

## pandas.DataFrame.from_csv

**classmethod** `DataFrame.from_csv`(*path*, *header=0*, *sep=', '*, *index_col=0*, *parse_dates=True*, *encoding=None*)

Read delimited file into DataFrame

**Parameters** **path** : string

**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

**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_records

**classmethod** `DataFrame.from_records`(*data*, *index=None*, *exclude=None*, *columns=None*, *names=None*, *coerce_float=False*)

Convert structured or record ndarray to DataFrame

**Parameters** **data** : ndarray (structured dtype), list of tuples, 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, replacing any found in passed data

**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.to_csv

`DataFrame.`**`to_csv`**(*path_or_buf*, *sep=', '*, *na_rep=''*, *cols=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *nanRep=None*, *encoding=None*)

Write DataFrame to a comma-separated values (csv) file

**Parameters** **path_or_buf** : string or file handle / StringIO

File path

**na_rep** : string, default ''

Missing data representation

**cols** : sequence, optional

Columns to write

**header** : boolean, default True

Write out 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.

**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

## pandas.DataFrame.to_excel

`DataFrame.`**`to_excel`**(*excel_writer*, *sheet_name='sheet1'*, *na_rep=''*, *cols=None*, *header=True*, *index=True*, *index_label=None*)

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 rep'n

**cols** : sequence, optional

Columns to write

**header** : boolean, default True

Write out 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.

#### 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_dict

`DataFrame.`**`to_dict`**`()`

Convert DataFrame to nested dictionary

**Returns** **result** : dict like {column -> {index -> value}}

### pandas.DataFrame.to_records

`DataFrame.`**`to_records`**`(`*index=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

**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=True*, *nanRep=None*, *index_names=True*, *justify=None*, *force_unicode=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 width of each columns

**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

**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 configuration in pandas.core.common, 'left' 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

**Returns** **formatted** : string (or unicode, depending on data and options)

### pandas.DataFrame.save

`DataFrame.`**`save`**`(`*path*`)`

### pandas.DataFrame.load

**classmethod** `DataFrame.`**`load`**`(`*path*`)`

#### pandas.DataFrame.info

`DataFrame.info`(*verbose=True*, *buf=None*)

Concise summary of a DataFrame, used in __repr__ when very large.

**Parameters** **verbose** : boolean, default True

If False, don't print column count summary

**buf** : writable buffer, defaults to sys.stdout

## 21.4 Panel

### 21.4.1 Computations / Descriptive Stats

# PYTHON MODULE INDEX

# PYTHON MODULE INDEX

# INDEX