pandas: powerful Python data analysis toolkit

Release 0.12.0

Wes McKinney & PyData Development Team

January 31, 2014

CONTENTS

PDF Version

Zipped HTML **Date**: January 31, 2014 **Version**: 0.12.0

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

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

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

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

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

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis / manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, `Series` (1-dimensional) and `DataFrame` (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, `DataFrame` provides everything that R's `data.frame` provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let *Series*, *DataFrame*, etc. automatically align the data for you in computations
- Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets
- Intuitive **merging** and **joining** data sets
- Flexible **reshaping** and pivoting of data sets
- **Hierarchical** labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
- **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

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

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

CHAPTER

ONE

WHAT'S NEW

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

1.1 v0.12.0 (July 24, 2013)

This is a major release from 0.11.0 and includes several new features and enhancements along with a large number of bug fixes.

Highlites include a consistent I/O API naming scheme, routines to read html, write multi-indexes to csv files, read & write STATA data files, read & write JSON format files, Python 3 support for `HDFStore`, filtering of groupby expressions via `filter`, and a revamped `replace` routine that accepts regular expressions.

1.1.1 API changes

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

  The corresponding `writer` functions are object methods that are accessed like `df.to_csv()`
  - `to_csv`
  - `to_excel`
  - `to_hdf`
  - `to_sql`
  - `to_json`
  - `to_html`

  - to_stata
  - to_clipboard

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

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

In [2]: p % 0

   first  second
0    NaN     NaN
1    NaN     NaN
2    NaN     NaN

In [3]: p % p

   first  second
0      0     NaN
1      0     NaN
2      0       0

In [4]: p / p

   first    second
0      1       inf
1      1       inf
2      1  1.000000

In [5]: p / 0

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

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

```
In [6]: df2 = DataFrame([{"val1": 1, "val2" : 20}, {"val1":1, "val2": 19},
   ...:                 {"val1":1, "val2": 27}, {"val1":1, "val2": 12}])
   ...:

In [7]: def func(dataf):
   ...:     return dataf["val2"]  - dataf["val2"].mean()
   ...:

# squeezing the result frame to a series (because we have unique groups)
In [8]: df2.groupby("val1", squeeze=True).apply(func)

0    0.5
1   -0.5
2    7.5
3   -7.5
Name: 1, dtype: float64

# no squeezing (the default, and behavior in 0.10.1)
```

```
In [9]: df2.groupby("val1").apply(func)

        0    1    2    3
val1
1     0.5 -0.5  7.5 -7.5
```

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

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

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

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

In [12]: mask

A     True
B    False
C     True
D    False
E     True
Name: a, dtype: bool

# this is what you should use
In [13]: df.loc[mask]

   a
A  0
C  2
E  4

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

   a
A  0
C  2
E  4
```

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

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

- Implement __nonzero__ for NDFrame objects (GH3691, GH3696)
- IO api
  - added top-level function read_excel to replace the following, The original API is deprecated and will be removed in a future version

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

    With

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

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

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

- DataFrame.to_html and DataFrame.to_latex now accept a path for their first argument (GH3702)
- Do not allow astypes on datetime64[ns] except to object, and timedelta64[ns] to object/int (GH3425)
- The behavior of datetime64 dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a TypeError when perfomed on a Series and return an *empty* Series when performed on a DataFrame similar to performing these operations on, for example, a DataFrame of slice objects:
  - sum, prod, mean, std, var, skew, kurt, corr, and cov
- read_html now defaults to None when reading, and falls back on bs4 + html5lib when lxml fails to parse. a list of parsers to try until success is also valid
- The internal pandas class hierarchy has changed (slightly). The previous PandasObject now is called PandasContainer and a new PandasObject has become the baseclass for PandasContainer as well as Index, Categorical, GroupBy, SparseList, and SparseArray (+ their base classes). Currently, PandasObject provides string methods (from StringMixin). (GH4090, GH4092)
- New StringMixin that, given a __unicode__ method, gets python 2 and python 3 compatible string methods (__str__, __bytes__, and __repr__). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)

1.1.2 I/O Enhancements

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

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

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

  In [16]: print df
     a  b
  0  0  a
  1  1  b
  2  2  c
  ```

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

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

In [19]: print df == alist[0]
      a     b
0  True  True
1  True  True
2  True  True
```

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

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

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

- Added module for reading and writing Stata files: `pandas.io.stata` (GH1512) accessable via `read_stata` top-level function for reading, and `to_stata` DataFrame method for writing, *See the docs*
- Added module for reading and writing json format files: `pandas.io.json` accessable via `read_json` top-level function for reading, and `to_json` DataFrame method for writing, *See the docs* various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
- `MultiIndex` column support for reading and writing csv format files
  - The `header` option in `read_csv` now accepts a list of the rows from which to read the index.
  - The option, `tupleize_cols` can now be specified in both `to_csv` and `read_csv`, to provide compatiblity for the pre 0.12 behavior of writing and reading `MultIndex` columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a `MultiIndex` column.

    Note: The default behavior in 0.12 remains unchanged from prior versions, but starting with 0.13, the default *to* write and read `MultiIndex` columns will be in the new format. (GH3571, GH1651, GH3141)
  - If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False)`, then any `names` on the columns index will be *lost*.

```
In [20]: from pandas.util.testing import makeCustomDataframe as mkdf

In [21]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)

In [22]: df.to_csv('mi.csv',tupleize_cols=False)

In [23]: print open('mi.csv').read()
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2

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

C0              C_l0_g0 C_l0_g1 C_l0_g2
```

```
C1              C_l1_g0 C_l1_g1 C_l1_g2
C2              C_l2_g0 C_l2_g1 C_l2_g2
C3              C_l3_g0 C_l3_g1 C_l3_g2
R0      R1
R_l0_g0 R_l1_g0    R0C0    R0C1    R0C2
R_l0_g1 R_l1_g1    R1C0    R1C1    R1C2
R_l0_g2 R_l1_g2    R2C0    R2C1    R2C2
R_l0_g3 R_l1_g3    R3C0    R3C1    R3C2
R_l0_g4 R_l1_g4    R4C0    R4C1    R4C2
```

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

```
In [25]: path = 'store_iterator.h5'

In [26]: DataFrame(randn(10,2)).to_hdf(path,'df',table=True)

In [27]: for df in read_hdf(path,'df', chunksize=3):
   ....:    print df
   ....:
          0         1
0  1.129167  0.231299
1 -0.184695 -0.138561
2 -0.924325  0.232465
          0         1
3 -0.789552 -0.364308
4 -0.534541  0.822239
5 -0.443109 -2.119990
          0         1
6 -0.460149  1.813962
7 -1.053571  0.009412
8 -0.165966 -0.848662
          0         1
9 -0.495553 -0.176421
```

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

1.1.3 Other Enhancements

- `DataFrame.replace()` now allows regular expressions on contained `Series` with object dtype. See the examples section in the regular docs *Replacing via String Expression*

  For example you can do

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

In [29]: df.replace(regex=r'\s*\.\s*', value=np.nan)

     a  b
0    a  1
1    b  2
2  NaN  3
3  NaN  4
```

  to replace all occurrences of the string `'.'` with zero or more instances of surrounding whitespace with `NaN`.

Regular string replacement still works as expected. For example, you can do

```
In [30]: df.replace('.', np.nan)

     a  b
0    a  1
1    b  2
2  NaN  3
3  NaN  4
```

to replace all occurrences of the string `'.'` with `NaN`.

- `pd.melt()` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame.
- `pd.set_option()` now allows N option, value pairs (GH3667).

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

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

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

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

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

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

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

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

  In [37]: sf.groupby(sf).filter(lambda x: x.sum() > 2)

  3    3
  4    3
  5    3
  dtype: int64
  ```

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

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

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

  In [39]: dff.groupby('B').filter(lambda x: len(x) > 2)

     A  B
  2  2  b
  3  3  b
  4  4  b
  5  5  b
  ```

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

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

    A    B
0 NaN  NaN
1 NaN  NaN
2   2    b
3   3    b
4   4    b
5   5    b
6 NaN  NaN
7 NaN  NaN
```

- Series and DataFrame hist methods now take a `figsize` argument (GH3834)
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations (GH3877)
- Timestamp.min and Timestamp.max now represent valid Timestamp instances instead of the default datetime.min and datetime.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

1.1.4 Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. (GH2301)

Note: This uses the `numpy.busdaycalendar` API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.

```
In [41]: from pandas.tseries.offsets import CustomBusinessDay

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

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

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

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

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

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

In [48]: print Series(dts.weekday, dts).map(Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
dtype: object
```

1.1.5 Bug Fixes

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

```
In [49]: strs = 'go', 'bow', 'joe', 'slow'

In [50]: ds = Series(strs)

In [51]: for s in ds.str:
   ....:     print s
   ....:
0    g
1    b
2    j
3    s
dtype: object
0    o
1    o
2    o
3    l
dtype: object
0    NaN
1      w
2      e
3      o
dtype: object
0    NaN
1    NaN
2    NaN
3      w
dtype: object

In [52]: s

0    NaN
1    NaN
2    NaN
3      w
dtype: object

In [53]: s.dropna().values.item() == 'w'
True
```

  The last element yielded by the iterator will be a `Series` containing the last element of the longest string in the `Series` with all other elements being `NaN`. Here since `'slow'` is the longest string and there are no other strings with the same length `'w'` is the only non-null string in the yielded `Series`.

- `HDFStore`
  - will retain index attributes (freq,tz,name) on recreation (GH3499)

  - will warn with a `AttributeConflictWarning` if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
  - support datelike columns with a timezone as data_columns (GH2852)
- Non-unique index support clarified (GH3468).
  - Fix assigning a new index to a duplicate index in a DataFrame would fail (GH3468)
  - Fix construction of a DataFrame with a duplicate index
  - ref_locs support to allow duplicative indices across dtypes, allows iget support to always find the index (even across dtypes) (GH2194)
  - applymap on a DataFrame with a non-unique index now works (removed warning) (GH2786), and fix (GH3230)
  - Fix to_csv to handle non-unique columns (GH3495)
  - Duplicate indexes with getitem will return items in the correct order (GH3455, GH3457) and handle missing elements like unique indices (GH3561)
  - Duplicate indexes with and empty DataFrame.from_records will return a correct frame (GH3562)
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed (GH3602)
  - Allow insert/delete to non-unique columns (GH3679)
  - Non-unique indexing with a slice via `loc` and friends fixed (GH3659)
  - Allow insert/delete to non-unique columns (GH3679)
  - Extend `reindex` to correctly deal with non-unique indices (GH3679)
  - `DataFrame.itertuples()` now works with frames with duplicate column names (GH3873)
  - Bug in non-unique indexing via `iloc` (GH4017); added `takeable` argument to `reindex` for location-based taking
  - Allow non-unique indexing in series via `.ix/.loc` and `__getitem__` (GH4246)
  - Fixed non-unique indexing memory allocation issue with `.ix/.loc` (GH4280)
- `DataFrame.from_records` did not accept empty recarrays (GH3682)
- `read_html` now correctly skips tests (GH3741)
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn't working (GH3907)
- Improved `network` test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. (GH3910, GH3914)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)

- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when regex=False (GH4115)
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` (GH4152)
- Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
- Fixed bug in plotting that wasn't raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
- Fixed bug where Index slices weren't carrying the name attribute (GH4226)
- Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
- Fixed bug where html5lib wasn't being properly skipped (GH4265)
- Fixed bug where get_data_famafrench wasn't using the correct file edges (GH4281)

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

1.2 v0.11.0 (April 22, 2013)

This is a major release from 0.10.1 and includes many new features and enhancements along with a large number of bug fixes. The methods of Selecting Data have had quite a number of additions, and Dtype support is now full-fledged. There are also a number of important API changes that long-time pandas users should pay close attention to.

There is a new section in the documentation, *10 Minutes to Pandas*, primarily geared to new users.

There is a new section in the documentation, *Cookbook*, a collection of useful recipes in pandas (and that we want contributions!).

There are several libraries that are now *Recommended Dependencies*

1.2.1 Selection Choices

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

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

  See more at *Selection by Label*
- `.iloc` is strictly integer position based (from `0` to `length-1` of the axis), will raise `IndexError` when the requested indicies are out of bounds. Allowed inputs are:
  - An integer e.g. `5`
  - A list or array of integers `[4, 3, 0]`

  - A slice object with ints `1:7`
  - A boolean array

  See more at *Selection by Position*

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

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

  See more at *Advanced Indexing*, *Advanced Hierarchical* and *Fallback Indexing*

1.2.2 Selection Deprecations

Starting in version 0.11.0, these methods *may* be deprecated in future versions.

- `irow`
- `icol`
- `iget_value`

See the section *Selection by Position* for substitutes.

1.2.3 Dtypes

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [1]: df1 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')

In [2]: df1

          A
0 -0.423595
1 -1.035433
2 -1.035375
3 -2.369079
4  0.524408
5 -0.871120
6  1.585433
7  0.039501

In [3]: df1.dtypes

A    float32
dtype: object

In [4]: df2 = DataFrame(dict( A = Series(randn(8),dtype='float16'),
   ...:                       B = Series(randn(8)),
   ...:                       C = Series(randn(8),dtype='uint8') ))
   ...:

In [5]: df2
```

```
          A         B  C
0  2.273438  1.799276  0
1 -1.118164 -0.968916  0
2  0.431396 -0.779465  0
3  0.554688 -2.000701  0
4 -1.333984 -1.866630  0
5 -0.332275 -1.101268  0
6 -0.485840  1.957478  0
7  1.725586  0.058889  0
```

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

```
A    float16
B    float64
C      uint8
dtype: object
```

```
# here you get some upcasting
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```

```
In [8]: df3
```

```
          A         B  C
0  1.849842  1.799276  0
1 -2.153597 -0.968916  0
2 -0.603978 -0.779465  0
3 -1.814392 -2.000701  0
4 -0.809577 -1.866630  0
5 -1.203395 -1.101268  0
6  1.099593  1.957478  0
7  1.765087  0.058889  0
```

```
In [9]: df3.dtypes
```

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

1.2.4 Dtype Conversion

This is lower-common-denomicator upcasting, meaning you get the dtype which can accomodate all of the types

```
In [10]: df3.values.dtype
dtype('float64')
```

Conversion

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

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

Mixed Conversion

```
In [12]: df3['D'] = '1.'

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

In [14]: df3.convert_objects(convert_numeric=True).dtypes

A    float32
B    float64
C    float64
D    float64
E      int64
dtype: object

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

In [16]: df3['E'] = df3['E'].astype('int32')

In [17]: df3.dtypes

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

Forcing Date coercion (and setting `NaT` when not datelike)

```
In [18]: s = Series([datetime(2001,1,1,0,0), 'foo', 1.0, 1,
   ....:             Timestamp('20010104'), '20010105'],dtype='O')
   ....:

In [19]: s.convert_objects(convert_dates='coerce')

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

1.2.5 Dtype Gotchas

Platform Gotchas

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

The following will all result in `int64` dtypes

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

a    int64
dtype: object
```

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

a    int64
dtype: object

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

a    int64
dtype: object
```

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

Upcasting Gotchas

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

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

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

In [25]: dfi

   A  B  C  D  E
0  1  1  0  1  1
1 -2  0  0  1  1
2  0  0  0  1  1
3 -1 -2  0  1  1
4  0 -1  0  1  1
5 -1 -1  0  1  1
6  1  1  0  1  1
7  1  0  0  1  1

In [26]: dfi.dtypes

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

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

In [28]: casted

    A   B   C  D  E
0   1   1 NaN  1  1
1 NaN NaN NaN  1  1
2 NaN NaN NaN  1  1
3 NaN NaN NaN  1  1
4 NaN NaN NaN  1  1
5 NaN NaN NaN  1  1
6   1   1 NaN  1  1
7   1 NaN NaN  1  1

In [29]: casted.dtypes

A    float64
```

```
B    float64
C    float64
D      int64
E      int32
dtype: object
```

While float dtypes are unchanged.

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

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

In [32]: df4.dtypes

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

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

In [34]: casted

          A         B   C  D  E
0  1.849842  1.799276 NaN  1  1
1       NaN       NaN NaN  1  1
2       NaN       NaN NaN  1  1
3       NaN       NaN NaN  1  1
4       NaN       NaN NaN  1  1
5       NaN       NaN NaN  1  1
6  1.099593  1.957478 NaN  1  1
7  1.765087  0.058889 NaN  1  1

In [35]: casted.dtypes

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

1.2.6 Datetimes Conversion

Datetime64[ns] columns in a DataFrame (or a Series) allow the use of `np.nan` to indicate a nan value, in addition to the traditional `NaT`, or not-a-time. This allows convenient nan setting in a generic way. Furthermore `datetime64[ns]` columns are created by default, when passed datetimelike objects (*this change was introduced in 0.10.1*) (GH2809, GH2810)

```
In [36]: df = DataFrame(randn(6,2),date_range('20010102',periods=6),columns=['A','B'])

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

In [38]: df
```

```
                   A         B           timestamp
2001-01-02 -0.277446 -1.102896 2001-01-03 00:00:00
2001-01-03  0.100307 -1.602814 2001-01-03 00:00:00
2001-01-04  0.920139 -0.643870 2001-01-03 00:00:00
2001-01-05  0.060336 -0.434942 2001-01-03 00:00:00
2001-01-06 -0.494305  0.737973 2001-01-03 00:00:00
2001-01-07  0.451632  0.334124 2001-01-03 00:00:00

# datetime64[ns] out of the box
In [39]: df.get_dtype_counts()

datetime64[ns]    1
float64           2
dtype: int64

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

In [41]: df

                   A         B           timestamp
2001-01-02 -0.277446 -1.102896 2001-01-03 00:00:00
2001-01-03  0.100307 -1.602814 2001-01-03 00:00:00
2001-01-04       NaN -0.643870                 NaT
2001-01-05       NaN -0.434942                 NaT
2001-01-06 -0.494305  0.737973 2001-01-03 00:00:00
2001-01-07  0.451632  0.334124 2001-01-03 00:00:00
```

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

```
In [42]: import datetime

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

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

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

In [46]: s

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

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

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

In [49]: s

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

In [50]: s.dtype
```

```
dtype('O')
```

1.2.7 API changes

- Added to_series() method to indicies, to facilitate the creation of indexers (GH3275)
- HDFStore
  - added the method select_column to select a single column from a table as a Series.
  - deprecated the unique method, can be replicated by select_column(key,column).unique()
  - min_itemsize parameter to append will now automatically create data_columns for passed keys

1.2.8 Enhancements

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

```
In [51]: df = DataFrame(dict(A=range(5), B=range(5)))

In [52]: df.to_hdf('store.h5','table',append=True)

In [53]: read_hdf('store.h5', 'table', where = ['index>2'])

   A  B
3  3  3
4  4  4
```

  - provide dotted attribute access to get from stores, e.g. store.df == store['df']
  - new keywords iterator=boolean, and chunksize=number_in_a_chunk are provided to support iteration on select and select_as_multiple (GH3076)
- You can now select timestamps from an *unordered* timeseries similarly to an *ordered* timeseries (GH2437)
- You can now select with a string from a DataFrame with a datelike index, in a similar way to a Series (GH3070)

```
In [54]: idx = date_range("2001-10-1", periods=5, freq='M')

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

In [56]: ts['2001']

2001-10-31    0.745574
2001-11-30    0.203280
2001-12-31    0.951437
Freq: M, dtype: float64

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

In [58]: df['2001']
```

```
                   A
2001-10-31  0.745574
2001-11-30  0.203280
2001-12-31  0.951437
```

- `Squeeze` to possibly remove length 1 dimensions from an object.

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

In [60]: p

<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D

In [61]: p.reindex(items=['ItemA']).squeeze()

                   A         B         C         D
2001-01-02  1.231965 -1.334798 -0.032730 -1.181730
2001-01-03 -1.122736 -1.995631  0.297274  2.061559
2001-01-04  1.658786 -0.755777 -0.965026  0.122730
2001-01-05 -0.627326  0.118077  1.147422  0.323622

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

2001-01-02   -1.334798
2001-01-03   -1.995631
2001-01-04   -0.755777
2001-01-05    0.118077
Freq: D, Name: B, dtype: float64
```

- In `pd.io.data.Options`,
  - Fix bug when trying to fetch data for the current month when already past expiry.
  - Now using lxml to scrape html instead of BeautifulSoup (lxml was faster).
  - New instance variables for calls and puts are automatically created when a method that creates them is called. This works for current month where the instance variables are simply `calls` and `puts`. Also works for future expiry months and save the instance variable as `callsMMYY` or `putsMMYY`, where `MMYY` are, respectively, the month and year of the option's expiry.
  - `Options.get_near_stock_price` now allows the user to specify the month for which to get relevant options data.
  - `Options.get_forward_data` now has optional kwargs `near` and `above_below`. This allows the user to specify if they would like to only return forward looking data for options near the current stock price. This just obtains the data from Options.get_near_stock_price instead of Options.get_xxx_data() (GH2758).
- Cursor coordinate information is now displayed in time-series plots.
- added option *display.max_seq_items* to control the number of elements printed per sequence pprinting it. (GH2979)
- added option *display.chop_threshold* to control display of small numerical values. (GH2739)

- added option *display.max_info_rows* to prevent verbose_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- value_counts() now accepts a "normalize" argument, for normalized histograms. (GH2710).
- DataFrame.from_records now accepts not only dicts but any instance of the collections.Mapping ABC.
- added option *display.mpl_style* providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- to_html() now accepts an optional "escape" argument to control reserved HTML character escaping (enabled by default) and escapes `&`, in addition to < and >. (GH2919)

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

1.3 v0.10.1 (January 22, 2013)

This is a minor release from 0.10.0 and includes new features, enhancements, and bug fixes. In particular, there is substantial new HDFStore functionality contributed by Jeff Reback.

An undesired API breakage with functions taking the `inplace` option has been reverted and deprecation warnings added.

1.3.1 API changes

- Functions taking an `inplace` option return the calling object as before. A deprecation message has been added
- Groupby aggregations Max/Min no longer exclude non-numeric data (GH2700)
- Resampling an empty DataFrame now returns an empty DataFrame instead of raising an exception (GH2640)
- The file reader will now raise an exception when NA values are found in an explicitly specified integer column instead of converting the column to float (GH2631)
- DatetimeIndex.unique now returns a DatetimeIndex with the same name and
- timezone instead of an array (GH2563)

1.3.2 New features

- MySQL support for database (contribution from Dan Allan)

1.3.3 HDFStore

You may need to upgrade your existing data files. Please visit the **compatibility** section in the main docs.

You can designate (and index) certain columns that you want to be able to perform queries on a table, by passing a list to `data_columns`

```
In [1]: store = HDFStore('store.h5')

In [2]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
   ...:                columns=['A', 'B', 'C'])
   ...:
```

```
In [3]: df['string'] = 'foo'

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

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

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

In [7]: df

                   A         B         C string string2
2000-01-01  0.709012 -0.192540 -1.195765    foo    cool
2000-01-02 -1.020229 -0.538519  0.861494    foo    cool
2000-01-03 -1.627149 -0.431159  1.104739    foo    cool
2000-01-04 -0.682496  1.024105 -0.712047    foo    cool
2000-01-05 -0.296578  0.916893 -0.967695    NaN    cool
2000-01-06 -0.085360 -0.334353 -0.279334    NaN    cool
2000-01-07  1.169735 -0.878264 -1.212880    foo    cool
2000-01-08 -0.181581  1.156482 -1.768441    bar    cool

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

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

                   A         B         C string string2
2000-01-04 -0.682496  1.024105 -0.712047    foo    cool

# this is in-memory version of this type of selection
In [10]: df[(df.B > 0) & (df.string == 'foo')]

                   A         B         C string string2
2000-01-04 -0.682496  1.024105 -0.712047    foo    cool
```

Retrieving unique values in an indexable or data column.

```
In [11]: import warnings

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

You can now store `datetime64` in data columns

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

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

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

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

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

In [18]: df_mixed1

                   A         B         C string string2          datetime64
```

```
2000-01-01  0.709012 -0.192540 -1.195765    foo    cool 2001-01-02 00:00:00
2000-01-02 -1.020229 -0.538519  0.861494    foo    cool 2001-01-02 00:00:00
2000-01-03 -1.627149 -0.431159  1.104739    foo    cool 2001-01-02 00:00:00
2000-01-04       NaN       NaN -0.712047    foo    cool 2001-01-02 00:00:00
2000-01-05 -0.296578  0.916893 -0.967695    NaN    cool 2001-01-02 00:00:00
2000-01-06 -0.085360 -0.334353 -0.279334    NaN    cool 2001-01-02 00:00:00
2000-01-07  1.169735 -0.878264 -1.212880    foo    cool 2001-01-02 00:00:00
2000-01-08 -0.181581  1.156482 -1.768441    bar    cool 2001-01-02 00:00:00
```

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

```
datetime64[ns]    1
float64           3
object            2
dtype: int64
```

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

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

```
                   A         B
2000-01-01  0.709012 -0.192540
2000-01-02 -1.020229 -0.538519
2000-01-03 -1.627149 -0.431159
2000-01-04 -0.682496  1.024105
2000-01-05 -0.296578  0.916893
2000-01-06 -0.085360 -0.334353
2000-01-07  1.169735 -0.878264
2000-01-08 -0.181581  1.156482
```

`HDFStore` now serializes multi-index dataframes when appending tables.

```
In [21]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   ....:                            ['one', 'two', 'three']],
   ....:                    labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   ....:                            [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   ....:                    names=['foo', 'bar'])
   ....:

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

In [23]: df
```

```
                  A         B         C
foo bar
foo one     0.410679 -0.938918 -1.452154
    two     0.835328 -0.698888  0.766402
    three   0.536443 -0.147986  0.339040
bar one    -0.195183 -1.332316  1.684194
    two    -0.137506  2.138582  0.118417
baz two     0.517623  1.646523  2.036856
    three -0.557814 -1.319266 -0.116488
qux one     0.093791 -0.129510 -0.147917
    two     0.689216  1.257932 -0.698661
    three   0.424236 -0.593513 -0.257994
```

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

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

                  A         B         C
foo bar
foo one     0.410679 -0.938918 -1.452154
    two     0.835328 -0.698888  0.766402
    three   0.536443 -0.147986  0.339040
bar one    -0.195183 -1.332316  1.684194
    two    -0.137506  2.138582  0.118417
baz two     0.517623  1.646523  2.036856
    three -0.557814 -1.319266 -0.116488
qux one     0.093791 -0.129510 -0.147917
    two     0.689216  1.257932 -0.698661
    three   0.424236 -0.593513 -0.257994

# the levels are automatically included as data columns
In [26]: store.select('mi', Term('foo=bar'))

                A         B         C
foo bar
bar one -0.195183 -1.332316  1.684194
    two -0.137506  2.138582  0.118417
```

Multi-table creation via `append_to_multiple` and selection via `select_as_multiple` can create/select from multiple tables and return a combined result, by using `where` on a selector table.

```
In [27]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
   ....:                                columns=['A', 'B', 'C', 'D', 'E', 'F'])
   ....:

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

# you can also create the tables individually
In [29]: store.append_to_multiple({ 'df1_mt' : ['A','B'], 'df2_mt' : None }, df_mt, selector = 'df1_m

In [30]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                  frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,strin
/df1_mt              frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt              frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_mixed            frame_table  (typ->appendable,nrows->8,ncols->6,indexers->[index])
/mi                  frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->[ba

# indiviual tables were created
In [31]: store.select('df1_mt')

                   A         B
2000-01-01  0.286809 -0.260858
2000-01-02  0.166832  0.576719
2000-01-03 -1.956312 -0.404748
2000-01-04  0.705209  0.487465
2000-01-05 -0.708517 -0.164255
2000-01-06 -1.441858 -0.279006
2000-01-07 -0.696374  2.672481
```

```
2000-01-08 -0.937772 -0.628728

In [32]: store.select('df2_mt')

                   C         D         E         F  foo
2000-01-01  1.066971 -0.234118 -0.866424  0.058853  bar
2000-01-02  1.050428 -0.512589 -0.144609 -1.323606  bar
2000-01-03 -0.078827  1.586977  0.817924 -1.589809  bar
2000-01-04  0.432875  1.248644  1.200735  1.645992  bar
2000-01-05  0.647882 -2.053613 -0.129861 -0.312326  bar
2000-01-06  0.614345  0.582126  0.888399  0.582476  bar
2000-01-07  0.841381 -0.288439  0.772803 -1.566607  bar
2000-01-08  0.984129  1.625077 -0.600940 -0.463547  bar

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

                   A         B         C         D         E         F  foo
2000-01-02  0.166832  0.576719  1.050428 -0.512589 -0.144609 -1.323606  bar
2000-01-04  0.705209  0.487465  0.432875  1.248644  1.200735  1.645992  bar
```

Enhancements

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

Bug Fixes

- `HDFStore` tables can now store `float32` types correctly (cannot be mixed with `float64` however)
- Fixed Google Analytics prefix when specifying request segment (GH2713).
- Function to reset Google Analytics token store so users can recover from improperly setup client secrets (GH2687).
- Fixed groupby bug resulting in segfault when passing in MultiIndex (GH2706)

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

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

1.4 v0.10.0 (December 17, 2012)

This is a major release from 0.9.1 and includes many new features and enhancements along with a large number of bug fixes. There are also a number of important API changes that long-time pandas users should pay close attention to.

1.4.1 File parsing new features

The delimited file parsing engine (the guts of `read_csv` and `read_table`) has been rewritten from the ground up and now uses a fraction the amount of memory while parsing, while being 40% or more faster in most use cases (in some cases much faster).

There are also many new features:

- Much-improved Unicode handling via the `encoding` option.
- Column filtering (`usecols`)
- Dtype specification (`dtype` argument)
- Ability to specify strings to be recognized as True/False
- Ability to yield NumPy record arrays (`as_recarray`)
- High performance `delim_whitespace` option
- Decimal format (e.g. European format) specification
- Easier CSV dialect options: `escapechar`, `lineterminator`, `quotechar`, etc.
- More robust handling of many exceptional kinds of files observed in the wild

1.4.2 API changes

Deprecated DataFrame BINOP TimeSeries special case behavior

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

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

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

In [3]: df

                   0         1         2         3
2000-01-01 -1.981994  0.252830 -0.578883 -0.129520
2000-01-02 -1.334960 -1.065058  0.287832  0.271441
2000-01-03  0.447157 -0.286125  0.694543  0.249647
2000-01-04 -0.632059 -1.605213  0.021863  0.771202
2000-01-05 -0.790803  0.305671 -1.153575 -0.064685
2000-01-06 -0.518046 -0.763586  1.892381  0.820290

# deprecated now
In [4]: df - df[0]

            0         1         2         3
2000-01-01  0  2.234824  1.403111  1.852474
2000-01-02  0  0.269902  1.622792  1.606401
2000-01-03  0 -0.733283  0.247386 -0.197510
2000-01-04  0 -0.973154  0.653922  1.403261
2000-01-05  0  1.096474 -0.362772  0.726118
2000-01-06  0 -0.245540  2.410427  1.338336

# Change your code to
In [5]: df.sub(df[0], axis=0) # align on axis 0 (rows)

            0         1         2         3
2000-01-01  0  2.234824  1.403111  1.852474
2000-01-02  0  0.269902  1.622792  1.606401
2000-01-03  0 -0.733283  0.247386 -0.197510
2000-01-04  0 -0.973154  0.653922  1.403261
2000-01-05  0  1.096474 -0.362772  0.726118
2000-01-06  0 -0.245540  2.410427  1.338336
```

You will get a deprecation warning in the 0.10.x series, and the deprecated functionality will be removed in 0.11 or later.

Altered resample default behavior

The default time series `resample` binning behavior of daily `D` and *higher* frequencies has been changed to `closed='left', label='left'`. Lower nfrequencies are unaffected. The prior defaults were causing a great deal of confusion for users, especially resampling data to daily frequency (which labeled the aggregated group with the end of the interval: the next day).

Note:

```
In [6]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')

In [7]: series = Series(np.arange(len(dates)), index=dates)

In [8]: series

2000-01-01 00:00:00     0
2000-01-01 04:00:00     1
2000-01-01 08:00:00     2
2000-01-01 12:00:00     3
2000-01-01 16:00:00     4
2000-01-01 20:00:00     5
2000-01-02 00:00:00     6
2000-01-02 04:00:00     7
2000-01-02 08:00:00     8
2000-01-02 12:00:00     9
2000-01-02 16:00:00    10
2000-01-02 20:00:00    11
2000-01-03 00:00:00    12
2000-01-03 04:00:00    13
2000-01-03 08:00:00    14
2000-01-03 12:00:00    15
2000-01-03 16:00:00    16
2000-01-03 20:00:00    17
2000-01-04 00:00:00    18
2000-01-04 04:00:00    19
2000-01-04 08:00:00    20
2000-01-04 12:00:00    21
2000-01-04 16:00:00    22
2000-01-04 20:00:00    23
2000-01-05 00:00:00    24
Freq: 4H, dtype: int64

In [9]: series.resample('D', how='sum')

2000-01-01     15
2000-01-02     51
2000-01-03     87
2000-01-04    123
2000-01-05     24
Freq: D, dtype: int64

# old behavior
In [10]: series.resample('D', how='sum', closed='right', label='right')

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

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

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

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

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

In [13]: s.fillna(0)

0    1.500000
1         inf
2    3.400000
3        -inf
dtype: float64

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

In [15]: pd.isnull(s)

0    False
1     True
2    False
3     True
dtype: bool

In [16]: s.fillna(0)

0    1.5
1    0.0
2    3.4
3    0.0
dtype: float64

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

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

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

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

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

   0    1  2
0  a    b  c
1  1  Yes  2
```

```
2  3   No  4

In [21]: pd.read_csv(StringIO(data), header=None, prefix='X')

  X0   X1 X2
0  a    b  c
1  1  Yes  2
2  3   No  4
```

- Values like `'Yes'` and `'No'` are not interpreted as boolean by default, though this can be controlled by new `true_values` and `false_values` arguments:

```
In [22]: print data
a,b,c
1,Yes,2
3,No,4

In [23]: pd.read_csv(StringIO(data))

   a    b  c
0  1  Yes  2
1  3   No  4

In [24]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])

   a      b  c
0  1   True  2
1  3  False  4
```

- The file parsers will not recognize non-string values arising from a converter function as NA if passed in the `na_values` argument. It's better to do post-processing using the `replace` function instead.
- Calling `fillna` on Series or DataFrame with no arguments is no longer valid code. You must either specify a fill value or an interpolation method:

```
In [25]: s = Series([np.nan, 1., 2., np.nan, 4])

In [26]: s

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

In [27]: s.fillna(0)

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

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

0   NaN
1     1
```

```
2     2
3     2
4     4
dtype: float64
```

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

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

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

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

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

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

  In [32]: s

  0    0.973896
  1    0.857088
  2    0.671961
  3    0.887791
  4    0.742435
  dtype: float64

  In [33]: s.apply(f)

            x       x^2
  0  0.973896  0.948473
  1  0.857088  0.734600
  2  0.671961  0.451532
  3  0.887791  0.788172
  4  0.742435  0.551209
  ```

- New API functions for working with pandas options (GH2097):

  - `get_option` / `set_option` - get/set the value of an option. Partial names are accepted. - `reset_option` - reset one or more options to their default value. Partial names are accepted. - `describe_option` - print a description of one or more options. When called with no arguments. print all registered options.

  Note: `set_printoptions`/ `reset_printoptions` are now deprecated (but functioning), the print options now live under "display.XYZ". For example:

  ```
  In [34]: get_option("display.max_rows")
  60
  ```

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

1.4.3 New features

1.4.4 Wide DataFrame Printing

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

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

In [36]: wide_frame
```

```
          0         1         2         3         4         5         6   \
0  0.722039  1.643652  0.010659  0.271584  1.534387 -0.598612 -0.805310
1 -0.268853  1.463431  0.434453 -0.531871 -0.965296  0.777320 -0.468978
2  1.249648  0.430064  1.356398 -0.427470 -0.542395 -0.056195  0.721914
3  1.987657 -0.794123 -0.905081  0.683094  2.299532  0.182536 -1.051262
4 -0.618226 -0.887271 -1.079597 -0.804151 -1.367845  0.586248  0.902790
          7         8         9         10        11        12        13  \
0 -1.245497 -1.185110 -1.598589  0.964828 -0.578950 -1.059236  0.218315
1 -0.115498  0.377557  1.726457 -0.837538  1.152593  1.679167  0.038564
2 -0.959224 -1.806757  0.065992  1.028521 -1.687575 -0.027231  0.149906
3  1.211515  1.069502 -0.377559 -0.295551  0.337310  0.326721  1.100168
4  1.229379  0.378480  0.939430  0.062776 -0.148337 -1.127071  0.463483
          14        15
0  0.475444 -0.785670
1  2.154660  0.538700
2 -0.347414 -0.395058
3 -1.381624 -1.524766
4  0.472451 -0.063966
```

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

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

In [38]: wide_frame
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 16 columns):
0     5  non-null values
1     5  non-null values
2     5  non-null values
3     5  non-null values
4     5  non-null values
5     5  non-null values
6     5  non-null values
7     5  non-null values
8     5  non-null values
9     5  non-null values
10    5  non-null values
11    5  non-null values
12    5  non-null values
13    5  non-null values
14    5  non-null values
15    5  non-null values
dtypes: float64(16)
```

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

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

In [40]: wide_frame

          0         1         2   \
0  0.722039  1.643652  0.010659
1 -0.268853  1.463431  0.434453
2  1.249648  0.430064  1.356398
3  1.987657 -0.794123 -0.905081
4 -0.618226 -0.887271 -1.079597
          3         4         5   \
0  0.271584  1.534387 -0.598612
1 -0.531871 -0.965296  0.777320
2 -0.427470 -0.542395 -0.056195
3  0.683094  2.299532  0.182536
4 -0.804151 -1.367845  0.586248
          6         7         8   \
0 -0.805310 -1.245497 -1.185110
1 -0.468978 -0.115498  0.377557
2  0.721914 -0.959224 -1.806757
3 -1.051262  1.211515  1.069502
4  0.902790  1.229379  0.378480
          9         10        11  \
0 -1.598589  0.964828 -0.578950
1  1.726457 -0.837538  1.152593
2  0.065992  1.028521 -1.687575
3 -0.377559 -0.295551  0.337310
4  0.939430  0.062776 -0.148337
          12        13        14  \
0 -1.059236  0.218315  0.475444
1  1.679167  0.038564  2.154660
2 -0.027231  0.149906 -0.347414
3  0.326721  1.100168 -1.381624
4 -1.127071  0.463483  0.472451
          15
0 -0.785670
1  0.538700
2 -0.395058
3 -1.524766
4 -0.063966
```

1.4.5 Updated PyTables Support

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

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

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

In [43]: df

                   A         B         C
2000-01-01 -0.579255  1.037877  0.646438
2000-01-02 -0.496395 -0.777511  0.705732
2000-01-03  1.184630  1.427407 -0.463865
```

```
2000-01-04  2.045486  0.025548 -0.003826
2000-01-05  1.312582  1.165116 -0.359024
2000-01-06 -0.442785  0.286406 -1.685139
2000-01-07  2.208326  0.317190  0.236846
2000-01-08  0.323316 -0.584380  0.545657

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

In [45]: df2 = df[4:]

In [46]: store.append('df', df1)

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

In [48]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])

# selecting the entire store
In [49]: store.select('df')

                   A         B         C
2000-01-01 -0.579255  1.037877  0.646438
2000-01-02 -0.496395 -0.777511  0.705732
2000-01-03  1.184630  1.427407 -0.463865
2000-01-04  2.045486  0.025548 -0.003826
2000-01-05  1.312582  1.165116 -0.359024
2000-01-06 -0.442785  0.286406 -1.685139
2000-01-07  2.208326  0.317190  0.236846
2000-01-08  0.323316 -0.584380  0.545657

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

In [51]: wp

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

# storing a panel
In [52]: store.append('wp',wp)

# selecting via A QUERY
In [53]: store.select('wp',
   ....:    [ Term('major_axis>20000102'), Term('minor_axis', '=', ['A','B']) ])
   ....:

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

```
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B

# removing data from tables
In [54]: store.remove('wp', [ 'major_axis', '>', wp.major_axis[3] ])
4

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

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-04 00:00:00
Minor_axis axis: A to D

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

In [57]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp            wide_table   (typ->appendable,nrows->16,ncols->2,indexers->[major_axis,minor_axis])
```

Enhancements

- added ability to hierarchical keys

```
In [58]: store.put('foo/bar/bah', df)

In [59]: store.append('food/orange', df)

In [60]: store.append('food/apple',  df)

In [61]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp                     wide_table   (typ->appendable,nrows->16,ncols->2,indexers->[major_ax
/food/apple             frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah            frame        (shape->[8,3])

# remove all nodes under this level
In [62]: store.remove('food')

In [63]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp                     wide_table   (typ->appendable,nrows->16,ncols->2,indexers->[major_ax
/foo/bar/bah            frame        (shape->[8,3])
```

- added mixed-dtype support!

```
In [64]: df['string'] = 'string'

In [65]: df['int']    = 1
```

```
In [66]: store.append('df',df)

In [67]: df1 = store.select('df')

In [68]: df1

                   A         B         C  string  int
2000-01-01 -0.579255  1.037877  0.646438  string    1
2000-01-02 -0.496395 -0.777511  0.705732  string    1
2000-01-03  1.184630  1.427407 -0.463865  string    1
2000-01-04  2.045486  0.025548 -0.003826  string    1
2000-01-05  1.312582  1.165116 -0.359024  string    1
2000-01-06 -0.442785  0.286406 -1.685139  string    1
2000-01-07  2.208326  0.317190  0.236846  string    1
2000-01-08  0.323316 -0.584380  0.545657  string    1

In [69]: df1.get_dtype_counts()

float64    3
int64      1
object     1
dtype: int64
```

- performance improvments on table writing
- support for arbitrarily indexed dimensions
- `SparseSeries` now has a `density` property (GH2384)
- enable `Series.str.strip/lstrip/rstrip` methods to take an input argument to strip arbitrary characters (GH2411)
- implement `value_vars` in `melt` to limit values to certain columns and add `melt` to pandas namespace (GH2412)

Bug Fixes

- added `Term` method of specifying where conditions (GH1996).
- `del store['df']` now call `store.remove('df')` for store deletion
- deleting of consecutive rows is much faster than before
- `min_itemsize` parameter can be specified in table creation to force a minimum size for indexing columns (the previous implementation would set the column size based on the first append)
- indexing support via `create_table_index` (requires PyTables >= 2.3) (GH698).
- appending on a store would fail if the table was not first created via `put`
- fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- minor change to select and remove: require a table ONLY if where is also provided (and not None)

Compatibility

0.10 of `HDFStore` is backwards compatible for reading tables created in a prior version of pandas, however, query terms using the prior (undocumented) methodology are unsupported. You must read in the entire file and write it out using the new format to take advantage of the updates.

1.4.6 N Dimensional Panels (Experimental)

Adding experimental support for Panel4D and factory functions to create n-dimensional named panels. *Docs* for NDim. Here is a taste of what to expect.

```
In [70]: p4d = Panel4D(randn(2, 2, 5, 4),
   ....:       labels=['Label1','Label2'],
   ....:       items=['Item1', 'Item2'],
   ....:       major_axis=date_range('1/1/2000', periods=5),
   ....:       minor_axis=['A', 'B', 'C', 'D'])
   ....:

In [71]: p4d

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

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

1.5 v0.9.1 (November 14, 2012)

This is a bugfix release from 0.9.0 and includes several new features and enhancements along with a large number of bug fixes. The new features include by-column sort order for DataFrame and Series, improved NA handling for the rank method, masking functions for DataFrame, and intraday time-series filtering for DataFrame.

1.5.1 New features

- *Series.sort*, *DataFrame.sort*, and *DataFrame.sort_index* can now be specified in a per-column manner to support multiple sort orders (GH928)

```
In [1]: df = DataFrame(np.random.randint(0, 2, (6, 3)), columns=['A', 'B', 'C'])

In [2]: df.sort(['A', 'B'], ascending=[1, 0])

   A  B  C
5  0  1  0
0  0  0  0
1  0  0  0
2  1  0  0
3  1  0  1
4  1  0  1
```

- *DataFrame.rank* now supports additional argument values for the *na_option* parameter so missing values can be assigned either the largest or the smallest rank (GH1508, GH2159)

```
In [3]: df = DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

In [4]: df.ix[2:4] = np.nan

In [5]: df.rank()
```

```
    A   B   C
0   1   2   1
1   2   1   3
2 NaN NaN NaN
3 NaN NaN NaN
4 NaN NaN NaN
5   3   3   2
```

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

```
   A  B  C
0  4  5  4
1  5  4  6
2  2  2  2
3  2  2  2
4  2  2  2
5  6  6  5
```

```
In [7]: df.rank(na_option='bottom')
```

```
   A  B  C
0  1  2  1
1  2  1  3
2  5  5  5
3  5  5  5
4  5  5  5
5  3  3  2
```

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

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

  ```
  In [8]: df = DataFrame(np.random.randn(5, 3), columns = ['A','B','C'])
  ```

  ```
  In [9]: df
  ```

  ```
            A         B         C
  0 -0.342297  0.771260 -0.390734
  1 -0.387514  0.370570 -0.846937
  2 -2.729405 -0.700188 -0.449835
  3  1.169848 -0.462677  0.718830
  4  0.651227 -0.119350 -0.945562
  ```

  ```
  In [10]: df[df['A'] > 0]
  ```

  ```
            A         B         C
  3  1.169848 -0.462677  0.718830
  4  0.651227 -0.119350 -0.945562
  ```

  If a DataFrame is sliced with a DataFrame based boolean condition (with the same size as the original DataFrame), then a DataFrame the same size (index and columns) as the original is returned, with elements that do not meet the boolean condition as *NaN*. This is accomplished via the new method *DataFrame.where*. In addition, *where* takes an optional *other* argument for replacement.

  ```
  In [11]: df[df>0]
  ```

```
          A        B        C
0       NaN  0.77126      NaN
1       NaN  0.37057      NaN
2       NaN      NaN      NaN
3  1.169848      NaN  0.71883
4  0.651227      NaN      NaN

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

          A        B        C
0       NaN  0.77126      NaN
1       NaN  0.37057      NaN
2       NaN      NaN      NaN
3  1.169848      NaN  0.71883
4  0.651227      NaN      NaN

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

          A         B         C
0  0.342297  0.771260  0.390734
1  0.387514  0.370570  0.846937
2  2.729405  0.700188  0.449835
3  1.169848  0.462677  0.718830
4  0.651227  0.119350  0.945562
```

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

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

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

In [16]: df2

          A         B         C
0 -0.342297  0.771260 -0.390734
1 -0.387514  3.000000 -0.846937
2 -2.729405 -0.700188 -0.449835
3  3.000000 -0.462677  3.000000
4  0.651227 -0.119350 -0.945562
```

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

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

          A        B        C
0       NaN  0.77126      NaN
1       NaN  0.37057      NaN
2       NaN      NaN      NaN
3  1.169848      NaN  0.71883
4  0.651227      NaN      NaN
```

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

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

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

```
....:

                   A         B         C
2000-01-03  0.980269  3.685731 -0.364217
2000-01-04  1.047916 -0.041232 -0.161812
2000-01-05  0.498581  0.731168 -0.537677
2000-01-06  1.120202  1.567621  0.003641
2000-01-07 -0.487094  0.571455 -1.611639
2000-01-10  0.836649  0.246462  0.588543
2000-01-11 -0.157161  1.340307  1.195778
```

- Added option to disable pandas-style tick locators and formatters using *series.plot(x_compat=True)* or *pandas.plot_params['x_compat'] = True* (GH2205)
- Existing TimeSeries methods *at_time* and *between_time* were added to DataFrame (GH2149)
- DataFrame.dot can now accept ndarrays (GH2042)
- DataFrame.drop now supports non-unique indexes (GH2101)
- Panel.shift now supports negative periods (GH2164)
- DataFrame now support unary ~ operator (GH2110)

1.5.2 API changes

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

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

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

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

2012-01   -1.876374
2012-02         NaN
2012-03         NaN
2012-04    1.079091
2012-05         NaN
2012-06         NaN
Freq: M, dtype: float64
```

- Period.end_time now returns the last nanosecond in the time interval (GH2124, GH2125, GH1764)

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

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

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

```
In [25]: data = 'A,B,C\n00001,001,5\n00002,002,6'

In [26]: from cStringIO import StringIO

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

       A  B  C
```

```
0  00001  1  5
1  00002  2  6
```

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

1.6 v0.9.0 (October 7, 2012)

This is a major release from 0.8.1 and includes several new features and enhancements along with a large number of bug fixes. New features include vectorized unicode encoding/decoding for *Series.str*, *to_latex* method to DataFrame, more flexible parsing of boolean values, and enabling the download of options data from Yahoo! Finance.

1.6.1 New features

- Add `encode` and `decode` for unicode handling to *vectorized string processing methods* in Series.str (GH1706)
- Add `DataFrame.to_latex` method (GH1735)
- Add convenient expanding window equivalents of all rolling_* ops (GH1785)
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance (GH1748, GH1739)
- More flexible parsing of boolean values (Yes, No, TRUE, FALSE, etc) (GH1691, GH1295)
- Add `level` parameter to `Series.reset_index`
- `TimeSeries.between_time` can now select times across midnight (GH1871)
- Series constructor can now handle generator as input (GH1679)
- `DataFrame.dropna` can now take multiple axes (tuple/list) as input (GH924)
- Enable `skip_footer` parameter in `ExcelFile.parse` (GH1843)

1.6.2 API changes

- The default column names when `header=None` and no columns names passed to functions like `read_csv` has changed to be more Pythonic and amenable to attribute access:

```
In [1]: from StringIO import StringIO

In [2]: data = '0,0,1\n1,1,0\n0,1,0'

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

In [4]: df

   0  1  2
0  0  0  1
1  1  1  0
2  0  1  0
```

- Creating a Series from another Series, passing an index, will cause reindexing to happen inside rather than treating the Series like an ndarray. Technically improper usages like `Series(df[col1], index=df[col2])` that worked before "by accident" (this was never intended) will lead to all NA Series in some cases. To be perfectly clear:

```
In [5]: s1 = Series([1, 2, 3])

In [6]: s1

0    1
1    2
2    3
dtype: int64

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

In [8]: s2

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

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

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

1.7 v0.8.1 (July 22, 2012)

This release includes a few new features, performance enhancements, and over 30 bug fixes from 0.8.0. New features include notably NA friendly string processing functionality and a series of new plot types and options.

1.7.1 New features

- Add *vectorized string processing methods* accessible via Series.str (GH620)
- Add option to disable adjustment in EWMA (GH1584)
- *Radviz plot* (GH1566)
- *Parallel coordinates plot*
- *Bootstrap plot*

- Per column styles and secondary y-axis plotting (GH1559)
- New datetime converters millisecond plotting (GH1599)
- Add option to disable "sparse" display of hierarchical indexes (GH1538)
- Series/DataFrame's `set_index` method can *append levels* to an existing Index/MultiIndex (GH1569, GH1577)

1.7.2 Performance improvements

- Improved implementation of rolling min and max (thanks to Bottleneck !)
- Add accelerated `'median'` GroupBy option (GH1358)
- Significantly improve the performance of parsing ISO8601-format date strings with `DatetimeIndex` or `to_datetime` (GH1571)
- Improve the performance of GroupBy on single-key aggregations and use with Categorical types
- Significant datetime parsing performance improvments

1.8 v0.8.0 (June 29, 2012)

This is a major release from 0.7.3 and includes extensive work on the time series handling and processing infrastructure as well as a great deal of new functionality throughout the library. It includes over 700 commits from more than 20 distinct authors. Most pandas 0.7.3 and earlier users should not experience any issues upgrading, but due to the migration to the NumPy datetime64 dtype, there may be a number of bugs and incompatibilities lurking. Lingering incompatibilities will be fixed ASAP in a 0.8.1 release if necessary. See the *full release notes* or issue tracker on GitHub for a complete list.

1.8.1 Support for non-unique indexes

All objects can now work with non-unique indexes. Data alignment / join operations work according to SQL join semantics (including, if application, index duplication in many-to-many joins)

1.8.2 NumPy datetime64 dtype and 1.6 dependency

Time series data are now represented using NumPy's datetime64 dtype; thus, pandas 0.8.0 now requires at least NumPy 1.6. It has been tested and verified to work with the development version (1.7+) of NumPy as well which includes some significant user-facing API changes. NumPy 1.6 also has a number of bugs having to do with nanosecond resolution data, so I recommend that you steer clear of NumPy 1.6's datetime64 API functions (though limited as they are) and only interact with this data using the interface that pandas provides.

See the end of the 0.8.0 section for a "porting" guide listing potential issues for users migrating legacy codebases from pandas 0.7 or earlier to 0.8.0.

Bug fixes to the 0.7.x series for legacy NumPy < 1.6 users will be provided as they arise. There will be no more further development in 0.7.x beyond bug fixes.

1.8.3 Time series changes and improvements

Note: With this release, legacy scikits.timeseries users should be able to port their code to use pandas.

Note: See *documentation* for overview of pandas timeseries API.

- New datetime64 representation **speeds up join operations and data alignment**, **reduces memory usage**, and improve serialization / deserialization performance significantly over datetime.datetime
- High performance and flexible **resample** method for converting from high-to-low and low-to-high frequency. Supports interpolation, user-defined aggregation functions, and control over how the intervals and result labeling are defined. A suite of high performance Cython/C-based resampling functions (including Open-High-Low-Close) have also been implemented.
- Revamp of *frequency aliases* and support for **frequency shortcuts** like '15min', or '1h30min'
- New *DatetimeIndex class* supports both fixed frequency and irregular time series. Replaces now deprecated DateRange class
- New `PeriodIndex` and `Period` classes for representing *time spans* and performing **calendar logic**, including the *12 fiscal quarterly frequencies* <*timeseries.quarterly*>. This is a partial port of, and a substantial enhancement to, elements of the scikits.timeseries codebase. Support for conversion between PeriodIndex and DatetimeIndex
- New Timestamp data type subclasses *datetime.datetime*, providing the same interface while enabling working with nanosecond-resolution data. Also provides *easy time zone conversions*.
- Enhanced support for *time zones*. Add *tz_convert* and `tz_lcoalize` methods to TimeSeries and DataFrame. All timestamps are stored as UTC; Timestamps from DatetimeIndex objects with time zone set will be localized to localtime. Time zone conversions are therefore essentially free. User needs to know very little about pytz library now; only time zone names as as strings are required. Time zone-aware timestamps are equal if and only if their UTC timestamps match. Operations between time zone-aware time series with different time zones will result in a UTC-indexed time series.
- Time series **string indexing conveniences** / shortcuts: slice years, year and month, and index values with strings
- Enhanced time series **plotting**; adaptation of scikits.timeseries matplotlib-based plotting code
- New `date_range`, `bdate_range`, and `period_range` *factory functions*
- Robust **frequency inference** function *infer_freq* and `inferred_freq` property of DatetimeIndex, with option to infer frequency on construction of DatetimeIndex
- to_datetime function efficiently **parses array of strings** to DatetimeIndex. DatetimeIndex will parse array or list of strings to datetime64
- **Optimized** support for datetime64-dtype data in Series and DataFrame columns
- New NaT (Not-a-Time) type to represent **NA** in timestamp arrays
- Optimize Series.asof for looking up **"as of" values** for arrays of timestamps
- Milli, Micro, Nano date offset objects
- Can index time series with datetime.time objects to select all data at particular **time of day** (`TimeSeries.at_time`) or **between two times** (`TimeSeries.between_time`)
- Add *tshift* method for leading/lagging using the frequency (if any) of the index, as opposed to a naive lead/lag using shift

1.8.4 Other new features

- New *cut* and `qcut` functions (like R's cut function) for computing a categorical variable from a continuous variable by binning values either into value-based (`cut`) or quantile-based (`qcut`) bins
- Rename `Factor` to `Categorical` and add a number of usability features
- Add *limit* argument to fillna/reindex
- More flexible multiple function application in GroupBy, and can pass list (name, function) tuples to get result in particular order with given names
- Add flexible *replace* method for efficiently substituting values
- Enhanced *read_csv/read_table* for reading time series data and converting multiple columns to dates
- Add *comments* option to parser functions: read_csv, etc.
- Add :ref‘dayfirst <io.dayfirst>‘ option to parser functions for parsing international DD/MM/YYYY dates
- Allow the user to specify the CSV reader *dialect* to control quoting etc.
- Handling *thousands* separators in read_csv to improve integer parsing.
- Enable unstacking of multiple levels in one shot. Alleviate `pivot_table` bugs (empty columns being introduced)
- Move to klib-based hash tables for indexing; better performance and less memory usage than Python's dict
- Add first, last, min, max, and prod optimized GroupBy functions
- New *ordered_merge* function
- Add flexible *comparison* instance methods eq, ne, lt, gt, etc. to DataFrame, Series
- Improve *scatter_matrix* plotting function and add histogram or kernel density estimates to diagonal
- Add *'kde'* plot option for density plots
- Support for converting DataFrame to R data.frame through rpy2
- Improved support for complex numbers in Series and DataFrame
- Add *pct_change* method to all data structures
- Add max_colwidth configuration option for DataFrame console output
- *Interpolate* Series values using index values
- Can select multiple columns from GroupBy
- Add *update* methods to Series/DataFrame for updating values in place
- Add `any` and `all` method to DataFrame

1.8.5 New plotting methods

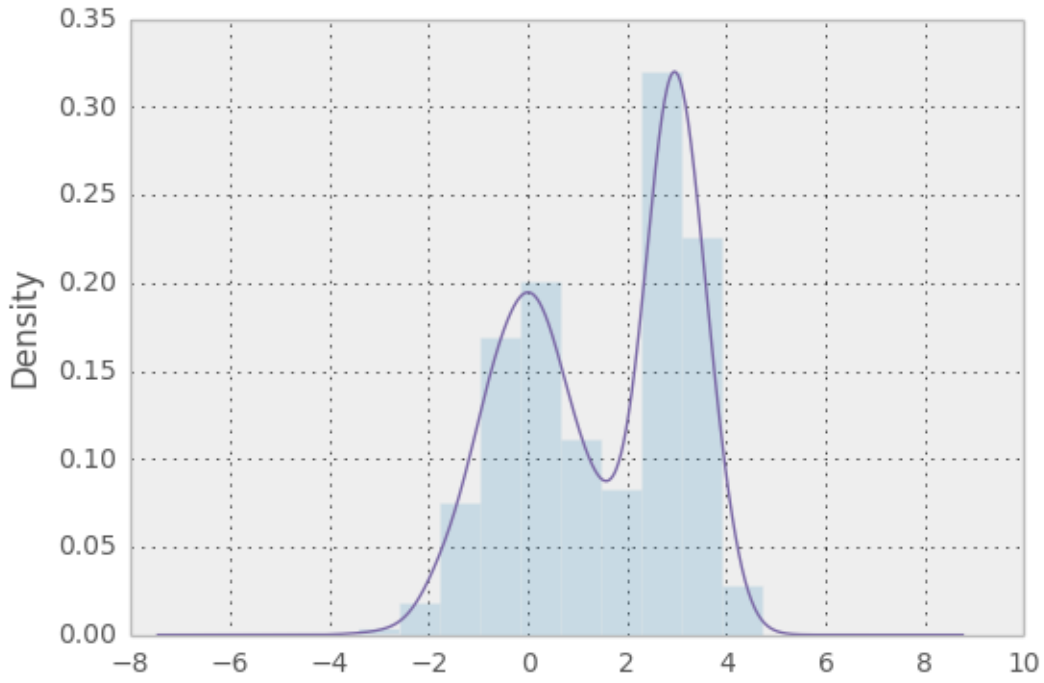
`Series.plot` now supports a `secondary_y` option:

```
In [1]: plt.figure()
<matplotlib.figure.Figure at 0x7fec350>

In [2]: fx['FR'].plot(style='g')
<matplotlib.axes.AxesSubplot at 0x7fec7d0>
```

```
In [3]: fx['IT'].plot(style='k--', secondary_y=True)
<matplotlib.axes.AxesSubplot at 0x8030b90>
```

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

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

In [5]: plt.figure()
<matplotlib.figure.Figure at 0x7fec3d0>

In [6]: s.hist(normed=True, alpha=0.2)
<matplotlib.axes.AxesSubplot at 0x5e6fe10>

In [7]: s.plot(kind='kde')
<matplotlib.axes.AxesSubplot at 0x5e6fe10>
```



See *the plotting page* for much more.

1.8.6 Other API changes

- Deprecation of `offset`, `time_rule`, and `timeRule` arguments names in time series functions. Warnings will be printed until pandas 0.9 or 1.0.

1.8.7 Potential porting issues for pandas <= 0.7.3 users

The major change that may affect you in pandas 0.8.0 is that time series indexes use NumPy's `datetime64` data type instead of `dtype=object` arrays of Python's built-in `datetime.datetime` objects. `DateRange` has been replaced by `DatetimeIndex` but otherwise behaved identically. But, if you have code that converts `DateRange` or `Index` objects that used to contain `datetime.datetime` values to plain NumPy arrays, you may have bugs lurking with code using scalar values because you are handing control over to NumPy:

```
In [8]: import datetime

In [9]: rng = date_range('1/1/2000', periods=10)

In [10]: rng[5]
Timestamp('2000-01-06 00:00:00', tz=None)

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

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

In [13]: scalar_val = rng_asarray[5]

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

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

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

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

In [16]: stamp_array
Index([2000-01-01 00:00:00, 2000-01-02 00:00:00, 2000-01-03 00:00:00, 2000-01-04 00:00:00, 2000-01-05

In [17]: stamp_array[5]
Timestamp('2000-01-06 00:00:00', tz=None)
```

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

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

In [19]: dt_array

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

In [20]: dt_array[5]
datetime.datetime(2000, 1, 6, 0, 0)
```

matplotlib knows how to handle `datetime.datetime` but not Timestamp objects. While I recommend that you plot time series using `TimeSeries.plot`, you can either use `to_pydatetime` or register a converter for the Timestamp type. See matplotlib documentation for more on this.

Warning: There are bugs in the user-facing API with the nanosecond datetime64 unit in NumPy 1.6. In particular, the string version of the array shows garbage values, and conversion to `dtype=object` is similarly broken.

```
In [21]: rng = date_range('1/1/2000', periods=10)

In [22]: rng

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

In [23]: np.asarray(rng)

array(['2000-01-01T02:00:00.000000000+0200',
       '2000-01-02T02:00:00.000000000+0200',
       '2000-01-03T02:00:00.000000000+0200',
       '2000-01-04T02:00:00.000000000+0200',
       '2000-01-05T02:00:00.000000000+0200',
       '2000-01-06T02:00:00.000000000+0200',
       '2000-01-07T02:00:00.000000000+0200',
       '2000-01-08T02:00:00.000000000+0200',
       '2000-01-09T02:00:00.000000000+0200',
       '2000-01-10T02:00:00.000000000+0200'], dtype='datetime64[ns]')

In [24]: converted = np.asarray(rng, dtype=object)

In [25]: converted[5]
947116800000000000L
```

Trust me: don't panic. If you are using NumPy 1.6 and restrict your interaction with `datetime64` values to pandas's API you will be just fine. There is nothing wrong with the data-type (a 64-bit integer internally); all of the important data processing happens in pandas and is heavily tested. I strongly recommend that you **do not work directly with datetime64 arrays in NumPy 1.6** and only use the pandas API.

Support for non-unique indexes: In the latter case, you may have code inside a `try:... catch:` block that failed due to the index not being unique. In many cases it will no longer fail (some method like `append` still check for uniqueness unless disabled). However, all is not lost: you can inspect `index.is_unique` and raise an exception explicitly if it is `False` or go to a different code branch.

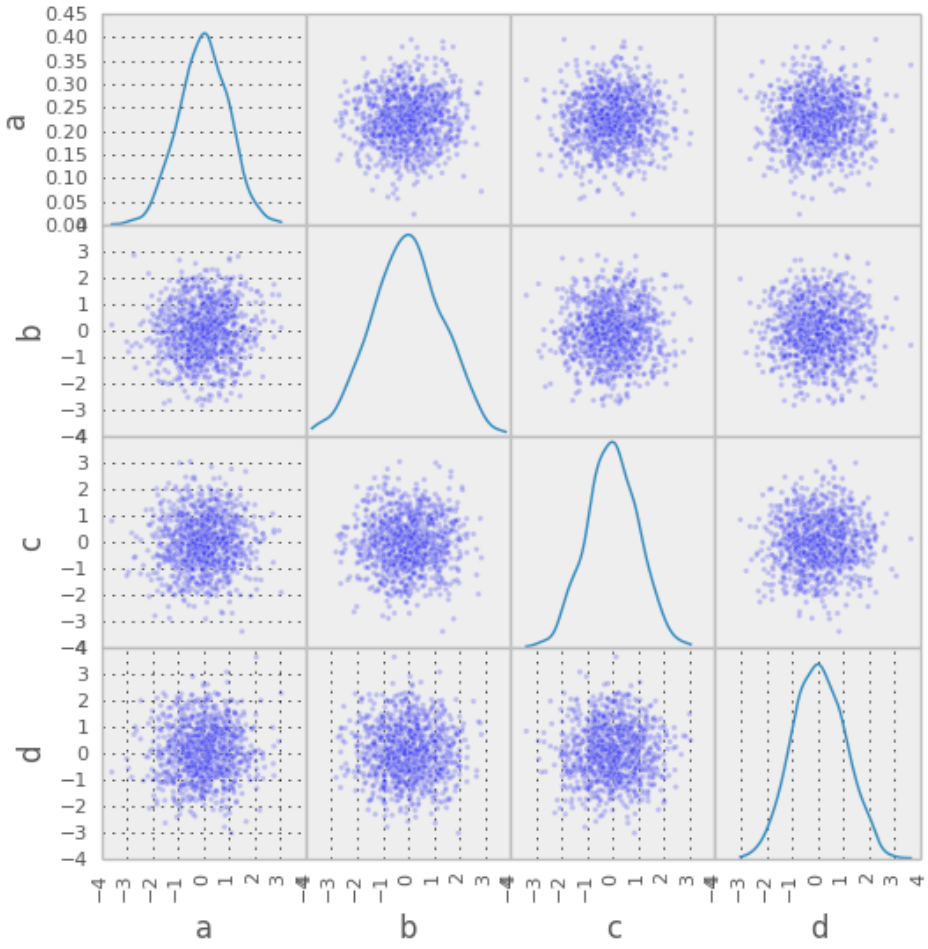
1.9 v.0.7.3 (April 12, 2012)

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.

1.9.1 New features

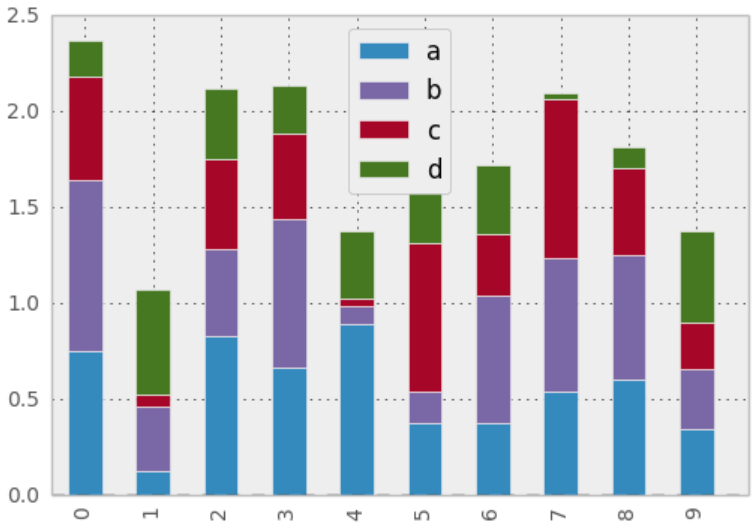
- New *fixed width file reader*, `read_fwf`
- New *scatter_matrix* function for making a scatter plot matrix

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

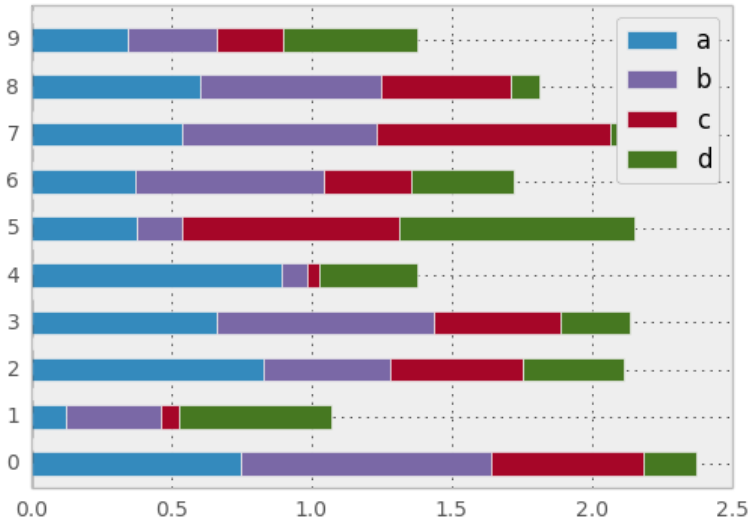


- Add `stacked` argument to Series and DataFrame's `plot` method for *stacked bar plots*.

```
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

1.9.2 NA Boolean Comparison API Change

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

```
In [1]: series = Series(['Steve', np.nan, 'Joe'])

In [2]: series == 'Steve'

0     True
1    False
2    False
dtype: bool

In [3]: series != 'Steve'

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

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

```
In [4]: mask = series == 'Steve'

In [5]: series[mask & series.notnull()]

0    Steve
dtype: object
```

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

1.9.3 Other API Changes

When calling `apply` on a grouped Series, the return value will also be a Series, to be more consistent with the `groupby` behavior with DataFrame:

```
In [1]: df = DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
   ...:                        'foo', 'bar', 'foo', 'foo'],
   ...:                 'B' : ['one', 'one', 'two', 'three',
   ...:                        'two', 'two', 'one', 'three'],
   ...:                 'C' : np.random.randn(8), 'D' : np.random.randn(8)})
   ...:

In [2]: df

     A      B         C         D
0  foo    one -0.426632  0.075555
1  bar    one  0.359373 -0.554339
2  foo    two -0.904623 -0.081867
3  bar  three  0.602661  0.656628
4  foo    two -0.463909  0.646616
5  bar    two  0.401666 -0.361991
6  foo    one -0.453960 -0.121668
7  foo  three -0.003605 -1.152130

In [3]: grouped = df.groupby('A')['C']

In [4]: grouped.describe()

A
bar  count    3.000000
     mean     0.454566
     std      0.129985
     min      0.359373
     25%      0.380519
     50%      0.401666
     75%      0.502163
     max      0.602661
foo  count    5.000000
     mean    -0.450546
     std      0.318867
     min     -0.904623
     25%     -0.463909
     50%     -0.453960
     75%     -0.426632
     max     -0.003605
dtype: float64

In [5]: grouped.apply(lambda x: x.order()[-2:]) # top 2 values

A
bar  5    0.401666
     3    0.602661
foo  0   -0.426632
     7   -0.003605
dtype: float64
```

1.10 v.0.7.2 (March 16, 2012)

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

1.10.1 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)

1.10.2 Performance improvements

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

1.11 v.0.7.1 (February 29, 2012)

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

1.11.1 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

1.11.2 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)

1.12 v.0.7.0 (February 9, 2012)

1.12.1 New features

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

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

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

```
                0          1          2          3
count  10.000000  10.000000  10.000000  10.000000
mean   -0.446115   0.405112   0.528573  -0.038971
std     0.602461   0.720553   0.708179   1.023853
min    -1.676067  -0.872725  -0.490528  -2.001988
25%    -0.549039  -0.097344   0.152407  -0.020831
50%    -0.274871   0.620660   0.337195   0.117498
75%    -0.092780   0.828148   1.068922   0.465303
max     0.180880   1.246181   1.601909   1.226653
```

- *Add* `reorder_levels` method to Series and DataFrame (GH534)
- *Add* dict-like `get` function to DataFrame and Panel (GH521)
- *Add* `DataFrame.iterrows` method for efficiently iterating through the rows of a DataFrame
- *Add* `DataFrame.to_panel` with code adapted from `LongPanel.to_long`
- *Add* `reindex_axis` method added to DataFrame
- *Add* `level` option to binary arithmetic functions on `DataFrame` and `Series`
- *Add* `level` option to the `reindex` and `align` methods on Series and DataFrame for broadcasting values across a level (GH542, GH552, others)
- *Add* attribute-based item access to `Panel` and add IPython completion (GH563)
- *Add* `logy` option to `Series.plot` for log-scaling on the Y axis
- *Add* `index` and `header` options to `DataFrame.to_string`
- *Can* pass multiple DataFrames to `DataFrame.join` to join on index (GH115)
- *Can* pass multiple Panels to `Panel.join` (GH115)
- *Added* `justify` argument to `DataFrame.to_string` to allow different alignment of column headers
- *Add* `sort` option to GroupBy to allow disabling sorting of the group keys for potential speedups (GH595)

- *Can* pass MaskedArray to Series constructor (GH563)
- *Add* Panel item access via attributes and IPython completion (GH554)
- Implement `DataFrame.lookup`, fancy-indexing analogue for retrieving values given a sequence of row and column labels (GH338)
- Can pass a *list of functions* to aggregate with groupby on a DataFrame, yielding an aggregated result with hierarchical columns (GH166)
- Can call `cummin` and `cummax` on Series and DataFrame to get cumulative minimum and maximum, respectively (GH647)
- `value_range` added as utility function to get min and max of a dataframe (GH288)
- Added `encoding` argument to `read_csv`, `read_table`, `to_csv` and `from_csv` for non-ascii text (GH717)
- *Added* `abs` method to pandas objects
- *Added* `crosstab` function for easily computing frequency tables
- *Added* `isin` method to index objects
- *Added* `level` argument to `xs` method of DataFrame.

1.12.2 API Changes to integer indexing

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

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

In [4]: s

0    -0.112337
2     0.813895
4    -0.541691
6     1.493949
8    -0.841607
10    0.694848
12    1.475483
14   -0.104269
16   -0.545897
18    0.020178
dtype: float64

In [5]: s[0]
-0.11233661695640235

In [6]: s[2]
0.81389491723208873

In [7]: s[4]
-0.54169091737261921
```

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`

1.12.3 API tweaks regarding label-based slicing

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

```
In [8]: s = Series(randn(6), index=list('gmkaec'))

In [9]: s

g   -1.306170
m    0.387885
k   -1.094556
a   -1.379918
e    1.574699
c    0.234904
dtype: float64
```

Then this is OK:

```
In [10]: s.ix['k':'e']

k   -1.094556
a   -1.379918
e    1.574699
dtype: float64
```

But this is not:

```
In [12]: s.ix['b':'h']
KeyError 'b'
```

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

```
In [11]: s2 = s.sort_index()

In [12]: s2

a   -1.379918
c    0.234904
e    1.574699
g   -1.306170
k   -1.094556
m    0.387885
dtype: float64

In [13]: s2.ix['b':'h']

c    0.234904
e    1.574699
g   -1.306170
dtype: float64
```

1.12.4 Changes to Series [] operator

As as notational convenience, you can pass a sequence of labels or a label slice to a Series when getting and setting values via [] (i.e. the __getitem__ and __setitem__ methods). The behavior will be the same as passing similar input to ix **except in the case of integer indexing**:

```
In [14]: s = Series(randn(6), index=list('acegkm'))

In [15]: s

a   -1.337087
c   -1.001423
e   -2.094396
g    1.378495
k    1.723121
m    1.345118
dtype: float64

In [16]: s[['m', 'a', 'c', 'e']]

m    1.345118
a   -1.337087
c   -1.001423
e   -2.094396
dtype: float64

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

c   -1.001423
e   -2.094396
g    1.378495
k    1.723121
dtype: float64

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

```
c   -1.001423
e   -2.094396
g    1.378495
k    1.723121
dtype: float64
```

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

```
In [19]: s = Series(randn(6), index=range(0, 12, 2))

In [20]: s[[4, 0, 2]]

4    0.675077
0   -0.940475
2    0.007452
dtype: float64

In [21]: s[1:5]

2    0.007452
4    0.675077
6    0.491628
8    0.962361
dtype: float64
```

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

1.12.5 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)

1.12.6 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)

1.13 v.0.6.1 (December 13, 2011)

1.13.1 New features

- Can *append single rows* (as Series) to a DataFrame
- Add Spearman and Kendall rank *correlation* options to Series.corr and DataFrame.corr (GH428)
- *Added* `get_value` and `set_value` methods to Series, DataFrame, and Panel for very low-overhead access (>2x faster in many cases) to scalar elements (GH437, GH438). `set_value` is capable of producing an enlarged object.
- Add PyQt table widget to sandbox (GH435)
- DataFrame.align can *accept Series arguments* and an *axis option* (GH461)
- Implement new *SparseArray* and *SparseList* data structures. SparseSeries now derives from SparseArray (GH463)
- *Better console printing options* (GH453)
- Implement fast *data ranking* for Series and DataFrame, fast versions of scipy.stats.rankdata (GH428)
- Implement *DataFrame.from_items* alternate constructor (GH444)
- DataFrame.convert_objects method for *inferring better dtypes* for object columns (GH302)
- Add *rolling_corr_pairwise* function for computing Panel of correlation matrices (GH189)
- Add *margins* option to *pivot_table* for computing subgroup aggregates (GH114)
- Add `Series.from_csv` function (GH482)
- *Can pass* DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH #462)
- MultiIndex.get_level_values can *accept the level name*

1.13.2 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)

1.14 v.0.6.0 (November 25, 2011)

1.14.1 New Features

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

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

1.14.2 Performance Enhancements

- VBENCH Cythonized `cache_readonly`, resulting in substantial micro-performance enhancements throughout the codebase (GH361)
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than *np.apply_along_axis* (GH309)
- VBENCH Improved performance of `MultiIndex.from_tuples`
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations
- VBENCH + DOCUMENT Add `raw` option to `DataFrame.apply` for getting better performance when
- VBENCH Faster cythonized count by level in Series and DataFrame (GH341)
- VBENCH? Significant GroupBy performance enhancement with multiple keys with many "empty" combinations
- VBENCH New Cython vectorized function `map_infer` speeds up `Series.apply` and `Series.map` significantly when passed elementwise Python function, motivated by (GH355)
- VBENCH Significantly improved performance of `Series.order`, which also makes np.unique called on a Series faster (GH327)
- VBENCH Vastly improved performance of GroupBy on axes with a MultiIndex (GH299)

1.15 v.0.5.0 (October 24, 2011)

1.15.1 New Features

- *Added* `DataFrame.align` method with standard join options
- *Added* `parse_dates` option to `read_csv` and `read_table` methods to optionally try to parse dates in the index columns
- *Added* `nrows`, `chunksize`, and `iterator` arguments to `read_csv` and `read_table`. The last two return a new `TextParser` class capable of lazily iterating through chunks of a flat file (GH242)
- *Added* ability to join on multiple columns in `DataFrame.join` (GH214)
- Added private `_get_duplicates` function to `Index` for identifying duplicate values more easily (ENH5c)
- *Added* column attribute access to DataFrame.
- *Added* Python tab completion hook for DataFrame columns. (GH233, GH230)
- *Implemented* `Series.describe` for Series containing objects (GH241)
- *Added* inner join option to `DataFrame.join` when joining on key(s) (GH248)
- *Implemented* selecting DataFrame columns by passing a list to `__getitem__` (GH253)

- *Implemented* & and | to intersect / union Index objects, respectively (GH261)
- *Added* `pivot_table` convenience function to pandas namespace (GH234)
- *Implemented* `Panel.rename_axis` function (GH243)
- DataFrame will show index level names in console output (GH334)
- *Implemented* `Panel.take`
- *Added* `set_eng_float_format` for alternate DataFrame floating point string formatting (ENH61)
- *Added* convenience `set_index` function for creating a DataFrame index from its existing columns
- *Implemented* `groupby` hierarchical index level name (GH223)
- *Added* support for different delimiters in `DataFrame.to_csv` (GH244)
- TODO: DOCS ABOUT TAKE METHODS

1.15.2 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

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

1.16.1 New Features

- Added Python 3 support using 2to3 (GH200)
- *Added* `name` attribute to `Series`, now prints as part of `Series.__repr__`
- *Added* instance methods `isnull` and `notnull` to Series (GH209, GH203)
- *Added* `Series.align` method for aligning two series with choice of join method (ENH56)
- *Added* method `get_level_values` to `MultiIndex` (GH188)
- *Set* values in mixed-type `DataFrame` objects via `.ix` indexing attribute (GH135)
- Added new `DataFrame` *methods* `get_dtype_counts` and property `dtypes` (ENHdc)
- Added *ignore_index* option to `DataFrame.append` to stack DataFrames (ENH1b)
- `read_csv` tries to *sniff* delimiters using `csv.Sniffer` (GH146)

- `read_csv` can *read* multiple columns into a `MultiIndex`; DataFrame's `to_csv` method writes out a corresponding `MultiIndex` (GH151)
- `DataFrame.rename` has a new `copy` parameter to *rename* a DataFrame in place (ENHed)
- *Enable* unstacking by name (GH142)
- *Enable* `sortlevel` to work by level (GH141)

1.16.2 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 2.6 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

2.2.1 All platforms

Stable installers available on PyPI

Preliminary builds and installers on the Pandas download page .

2.2.2 Overview

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

2.3 Dependencies

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

2.4 Recommended Dependencies

- numexpr: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups.
- bottleneck: for accelerating certain types of `nan` evaluations. `bottleneck` uses specialized cython routines to achieve large speedups.

Note: You are highly encouraged to install these libraries, as they provide large speedups, especially if working with large data sets.

2.5 Optional Dependencies

- Cython: Only necessary to build development version. Version 0.17.1 or higher.

- SciPy: miscellaneous statistical functions
- PyTables: necessary for HDF5-based storage
- matplotlib: for plotting
- **statsmodels**
  - Needed for parts of `pandas.stats`
- **openpyxl, xlrd/xlwt**
  - openpyxl version 1.6.1 or higher
  - Needed for Excel I/O
- boto: necessary for Amazon S3 access.
- One of the following combinations of libraries is needed to use the top-level `read_html()` function:
  - BeautifulSoup4 and html5lib (Any recent version of html5lib is okay.)
  - BeautifulSoup4 and lxml
  - BeautifulSoup4 and html5lib and lxml
  - Only lxml, although see *HTML reading gotchas* for reasons as to why you should probably **not** take this approach.

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

Note:

- if you're on a system with `apt-get` you can do

  ```
  sudo apt-get build-dep python-lxml
  ```

  to get the necessary dependencies for installation of lxml. This will prevent further headaches down the line.

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

2.6 Installing from source

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

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

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

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

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

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

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

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

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

2.7 Running the test suite

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

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

OK (SKIP=2)
```

CHAPTER

THREE

FREQUENTLY ASKED QUESTIONS (FAQ)

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

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

As of 0.12, these are the relevant options, all under the *display* namespace, (e.g. display.width, etc'):

- notebook_repr_html: if True, IPython frontends with HTML support will display dataframes as HTML tables when possible.
- expand_repr (default True): when the frame width cannot fit within the screen, the output will be broken into multiple pages to accomedate. This applies to textual (as opposed to HTML) display only.
- max_columns: max dataframe columns to display. a wider frame will trigger a summary view, unless *expand_repr* is True and HTML output is disabled.
- max_rows: max dataframe rows display. a longer frame will trigger a summary view.
- width: width of display screen in characters, used to determine the width of lines when expand_repr is active, Setting this to None will trigger auto-detection of terminal width, this only works for proper terminals, not IPython frontends such as ipnb. width is ignored in IPython notebook, since the browser provides horizontal scrolling.

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

3.2 Adding Features to your Pandas Installation

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

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

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

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

In [2]: def just_foo_cols(self):
   ...:     """Get a list of column names containing the string 'foo'
   ...:     """
   ...:     return [x for x in self.columns if 'foo' in x]
   ...:

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

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

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

In [6]: del pd.DataFrame.just_foo_cols # you can also remove the new method
```

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

3.3 Migrating from scikits.timeseries to pandas >= 0.8.0

Starting with pandas 0.8.0, users of scikits.timeseries should have all of the features that they need to migrate their code to use pandas. Portions of the scikits.timeseries codebase for implementing calendar logic and timespan frequency conversions (but **not** resampling, that has all been implemented from scratch from the ground up) have been ported to the pandas codebase.

The scikits.timeseries notions of `Date` and `DateArray` are responsible for implementing calendar logic:

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

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

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

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

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

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

```
In [7]: pnow('D')  # scikits.timeseries.now()
Period('2014-01-31', 'D')

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

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

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

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

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

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

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

In [15]: rng

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

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

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

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

3.3.1 PeriodIndex / DateArray properties and functions

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

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

3.3.2 Frequency conversion

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

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

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

```
In [19]: data

timeseries([ 0.4691 -0.2829 -1.5091 -1.1356  1.2121 -0.1732  0.1192 -1.0442 -0.8618
 -2.1046 -0.4949  1.0718  0.7216 -0.7068 -1.0396  0.2719 -0.425   0.567
  0.2762 -1.0874 -0.6737  0.1136 -1.4784  0.525   0.4047  0.577  -1.715
 -1.0393 -0.3706 -1.1579 -1.3443  0.8449  1.0758 -0.109   1.6436 -1.4694
  0.357  -0.6746 -1.7769 -0.9689 -1.2945  0.4137  0.2767 -0.472  -0.014
 -0.3625 -0.0062 -0.9231  0.8957  0.8052],
   dates = [Jan-2014 ... Feb-2018],
   freq  = M)

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

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

Here is the equivalent pandas code:

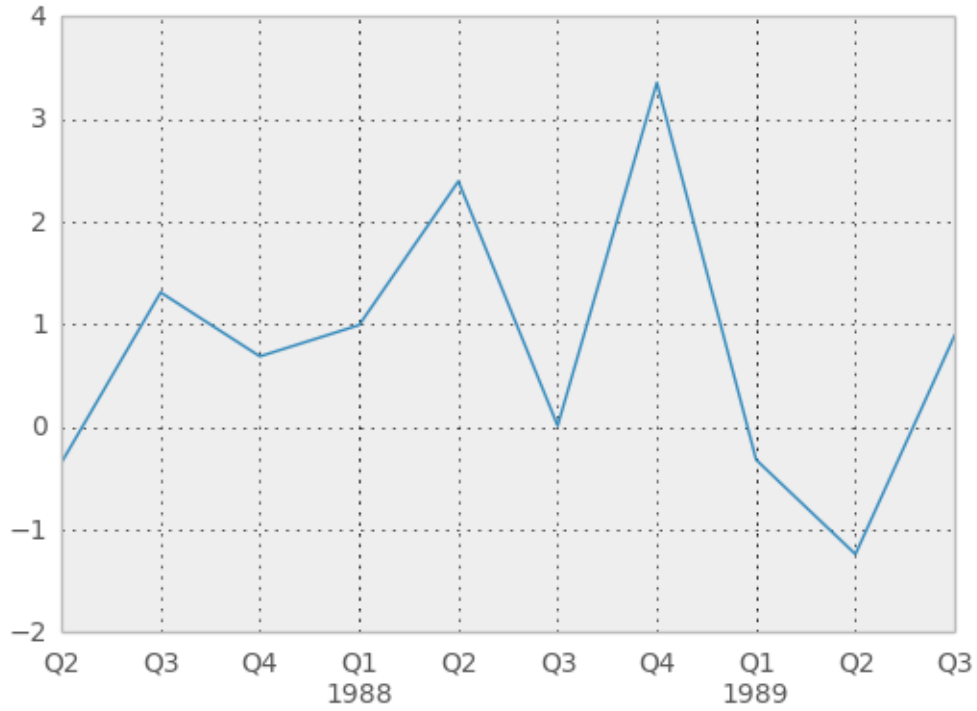
```
In [21]: rng = period_range('Jan-2000', periods=50, freq='M')

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

In [23]: data

2000-01   -1.206412
2000-02    2.565646
2000-03    1.431256
2000-04    1.340309
2000-05   -1.170299
2000-06   -0.226169
2000-07    0.410835
2000-08    0.813850
2000-09    0.132003
2000-10   -0.827317
2000-11   -0.076467
2000-12   -1.187678
2001-01    1.130127
2001-02   -1.436737
2001-03   -1.413681
2001-04    1.607920
2001-05    1.024180
2001-06    0.569605
2001-07    0.875906
2001-08   -2.211372
2001-09    0.974466
2001-10   -2.006747
2001-11   -0.410001
2001-12   -0.078638
2002-01    0.545952
2002-02   -1.219217
2002-03   -1.226825
2002-04    0.769804
2002-05   -1.281247
2002-06   -0.727707
2002-07   -0.121306
2002-08   -0.097883
```

```
2002-09    0.695775
2002-10    0.341734
2002-11    0.959726
2002-12   -1.110336
2003-01   -0.619976
2003-02    0.149748
2003-03   -0.732339
2003-04    0.687738
2003-05    0.176444
2003-06    0.403310
2003-07   -0.154951
2003-08    0.301624
2003-09   -2.179861
2003-10   -1.369849
2003-11   -0.954208
2003-12    1.462696
2004-01   -1.743161
2004-02   -0.826591
Freq: M, dtype: float64

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

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

3.3.3 Plotting

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

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

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

In [27]: plt.figure(); data.plot()
<matplotlib.axes.AxesSubplot at 0x6142d50>
```



3.3.4 Converting to and from period format

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

3.3.5 Treatment of missing data

Unlike scikits.timeseries, pandas data structures are not based on NumPy's `MaskedArray` object. Missing data is represented as `NaN` in numerical arrays and either as `None` or `NaN` in non-numerical arrays. Implementing a version of pandas's data structures that use MaskedArray is possible but would require the involvement of a dedicated maintainer. Active pandas developers are not interested in this.

3.3.6 Resampling with timestamps and periods

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

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

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

In [30]: data[:10]

2000-01-01   -0.487602
2000-01-02   -0.082240
2000-01-03   -2.182937
2000-01-04    0.380396
2000-01-05    0.084844
2000-01-06    0.432390
2000-01-07    1.519970
```

```
2000-01-08   -0.493662
2000-01-09    0.600178
2000-01-10    0.274230
Freq: D, dtype: float64

In [31]: data.index

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

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

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

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

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

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

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

In [36]: resampled.index

<class 'pandas.tseries.index.DatetimeIndex'>
[2000-12-31 00:00:00, ..., 2004-12-31 00:00:00]
Length: 5, Freq: A-DEC, Timezone: None
```

3.4 Byte-Ordering Issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. To deal with this issue you should convert the underlying NumPy array to the native system byte order *before* passing it to Series/DataFrame/Panel constructors using something similar to the following:

```
In [37]: x = np.array(range(10), '>i4') # big endian

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

In [39]: s = Series(newx)
```

See the NumPy documentation on byte order for more details.

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 (date_range) and custom date offsets enabling the implementation of customized frequencies
- Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects from the fast and efficient PyTables/HDF5 format.
- Memory-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

```
=======
License
=======

pandas is distributed under a 3-clause ("Simplified" or "New") BSD
license. Parts of NumPy, SciPy, numpydoc, bottleneck, which all have
BSD-compatible licenses, are included. Their licenses follow the pandas
license.

pandas license
==============

Copyright (c) 2011-2012, Lambda Foundry, Inc. and PyData Development Team
All rights reserved.

Copyright (c) 2008-2011 AQR Capital Management, LLC
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
       notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
       copyright notice, this list of conditions and the following
       disclaimer in the documentation and/or other materials provided
       with the distribution.

    * Neither the name of the copyright holder nor the names of any
       contributors may be used to endorse or promote products derived
       from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

About the Copyright Holders
===========================

AQR Capital Management began pandas development in 2008. Development was
led by Wes McKinney. AQR released the source under this license in 2009.
Wes is now an employee of Lambda Foundry, and remains the pandas project
lead.

The PyData Development Team is the collection of developers of the PyData
project. This includes all of the PyData sub-projects, including pandas. The
core team that coordinates development on GitHub can be found here:
http://github.com/pydata.
```

```
Full credits for pandas contributors can be found in the documentation.

Our Copyright Policy
====================

PyData uses a shared copyright model. Each contributor maintains copyright
over their contributions to PyData. However, it is important to note that
these contributions are typically only changes to the repositories. Thus,
the PyData source code, in its entirety, is not the copyright of any single
person or institution. Instead, it is the collective copyright of the
entire PyData Development Team. If individual contributors want to maintain
a record of what changes/contributions they have specific copyright on,
they should indicate their copyright in the commit message of the change
when they commit the change to one of the PyData repositories.

With this in mind, the following banner should be used in any source code
file to indicate the copyright and license terms:

#-----------------------------------------------------------------------------
# Copyright (c) 2012, PyData Development Team
# All rights reserved.
#
# Distributed under the terms of the BSD Simplified License.
#
# The full license is in the LICENSE file, distributed with this software.
#-----------------------------------------------------------------------------

Other licenses can be found in the LICENSES directory.
```

CHAPTER

FIVE

10 MINUTES TO PANDAS

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*

Customarily, we import as follows

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

In [2]: import numpy as np
```

5.1 Object Creation

See the *Data Structure Intro section*

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

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

In [4]: s

0     1
1     3
2     5
3   NaN
4     6
5     8
dtype: float64
```

Creating a `DataFrame` by passing a numpy array, with a datetime index and labeled columns.

```
In [5]: dates = pd.date_range('20130101',periods=6)

In [6]: dates

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

In [7]: df = pd.DataFrame(np.random.randn(6,4),index=dates,columns=list('ABCD'))

In [8]: df

                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
```

```
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Creating a `DataFrame` by passing a dict of objects that can be converted to series-like.

```
In [9]: df2 = pd.DataFrame({ 'A' : 1.,
   ...:                      'B' : pd.Timestamp('20130102'),
   ...:                      'C' : pd.Series(1,index=range(4),dtype='float32'),
   ...:                      'D' : np.array([3] * 4,dtype='int32'),
   ...:                      'E' : 'foo' })
   ...:

In [10]: df2
```

```
   A                   B  C  D    E
0  1 2013-01-02 00:00:00  1  3  foo
1  1 2013-01-02 00:00:00  1  3  foo
2  1 2013-01-02 00:00:00  1  3  foo
3  1 2013-01-02 00:00:00  1  3  foo
```

Having specific *dtypes*

```
In [11]: df2.dtypes
```

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

5.2 Viewing Data

See the *Basics section*

See the top & bottom rows of the frame

```
In [12]: df.head()
```

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

```
In [13]: df.tail(3)
```

```
                   A         B         C         D
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Display the index,columns, and the underlying numpy data

```
In [14]: df.index
```

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

```
In [15]: df.columns
Index([u'A', u'B', u'C', u'D'], dtype=object)
```

```
In [16]: df.values
```

```
array([[ 0.4691, -0.2829, -1.5091, -1.1356],
       [ 1.2121, -0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949,  1.0718],
       [ 0.7216, -0.7068, -1.0396,  0.2719],
       [-0.425 ,  0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784,  0.525 ]])
```

Describe shows a quick statistic summary of your data

```
In [17]: df.describe()
```

```
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.073711 -0.431125 -0.687758 -0.233103
std    0.843157  0.922818  0.779887  0.973118
min   -0.861849 -2.104569 -1.509059 -1.135632
25%   -0.611510 -0.600794 -1.368714 -1.076610
50%    0.022070 -0.228039 -0.767252 -0.386188
75%    0.658444  0.041933 -0.034326  0.461706
max    1.212112  0.567020  0.276232  1.071804
```

Transposing your data

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

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

Sorting by an axis

```
In [19]: df.sort_index(axis=1, ascending=False)
```

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

Sorting by values

```
In [20]: df.sort(columns='B')
```

```
                   A         B         C         D
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
```

```
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

5.3 Selection

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

See the *Indexing section* and below.

5.3.1 Getting

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

```
In [21]: df['A']
```

```
2013-01-01    0.469112
2013-01-02    1.212112
2013-01-03   -0.861849
2013-01-04    0.721555
2013-01-05   -0.424972
2013-01-06   -0.673690
Freq: D, Name: A, dtype: float64
```

Selecting via `[]`, which slices the rows.

```
In [22]: df[0:3]
```

```
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
```

```
In [23]: df['20130102':'20130104']
```

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

5.3.2 Selection by Label

See more in *Selection by Label*

For getting a cross section using a label

```
In [24]: df.loc[dates[0]]
```

```
A    0.469112
```

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

Selecting on a multi-axis by label

```
In [25]: df.loc[:,['A','B']]
```

```
                   A         B
2013-01-01  0.469112 -0.282863
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
2013-01-06 -0.673690  0.113648
```

Showing label slicing, both endpoints are *included*

```
In [26]: df.loc['20130102':'20130104',['A','B']]
```

```
                   A         B
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
```

Reduction in the dimensions of the returned object

```
In [27]: df.loc['20130102',['A','B']]
```

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

For getting a scalar value

```
In [28]: df.loc[dates[0],'A']
0.46911229990718628
```

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

```
In [29]: df.at[dates[0],'A']
0.46911229990718628
```

5.3.3 Selection by Position

See more in *Selection by Position*

Select via the position of the passed integers

```
In [30]: df.iloc[3]
```

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

By integer slices, acting similar to numpy/python

```
In [31]: df.iloc[3:5,0:2]

                   A         B
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
```

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

```
In [32]: df.iloc[[1,2,4],[0,2]]

                   A         C
2013-01-02  1.212112  0.119209
2013-01-03 -0.861849 -0.494929
2013-01-05 -0.424972  0.276232
```

For slicing rows explicitly

```
In [33]: df.iloc[1:3,:]

                   A         B         C         D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
```

For slicing columns explicitly

```
In [34]: df.iloc[:,1:3]

                   B         C
2013-01-01 -0.282863 -1.509059
2013-01-02 -0.173215  0.119209
2013-01-03 -2.104569 -0.494929
2013-01-04 -0.706771 -1.039575
2013-01-05  0.567020  0.276232
2013-01-06  0.113648 -1.478427
```

For getting a value explicity

```
In [35]: df.iloc[1,1]
-0.17321464905330858
```

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

```
In [36]: df.iat[1,1]
-0.17321464905330858
```

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

```
# these are allowed in python/numpy.
In [37]: x = list('abcdef')

In [38]: x[4:10]
['e', 'f']

In [39]: x[8:10]
[]
```

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

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

5.3.4 Boolean Indexing

Using a single column's values to select data.

```
In [40]: df[df.A > 0]
```

```
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

A `where` operation for getting.

```
In [41]: df[df > 0]
```

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

5.3.5 Setting

Setting a new column automatically aligns the data by the indexes

```
In [42]: s1 = pd.Series([1,2,3,4,5,6],index=date_range('20130102',periods=6))

In [43]: s1

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

In [44]: df['F'] = s1
```

Setting values by label

```
In [45]: df.at[dates[0],'A'] = 0
```

Setting values by position

```
In [46]: df.iat[0,1] = 0
```

Setting by assigning with a numpy array

```
In [47]: df.loc[:,'D'] = np.array([5] * len(df))
```

The result of the prior setting operations

```
In [48]: df

                   A         B         C  D   F
2013-01-01  0.000000  0.000000 -1.509059  5 NaN
```

```
2013-01-02  1.212112 -0.173215  0.119209  5   1
2013-01-03 -0.861849 -2.104569 -0.494929  5   2
2013-01-04  0.721555 -0.706771 -1.039575  5   3
2013-01-05 -0.424972  0.567020  0.276232  5   4
2013-01-06 -0.673690  0.113648 -1.478427  5   5
```

A `where` operation with setting.

```
In [49]: df2 = df.copy()

In [50]: df2[df2 > 0] = -df2

In [51]: df2
```

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

5.4 Missing Data

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

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

```
In [52]: df1 = df.reindex(index=dates[0:4],columns=list(df.columns) + ['E'])

In [53]: df1.loc[dates[0]:dates[1],'E'] = 1

In [54]: df1
```

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

To drop any rows that have missing data.

```
In [55]: df1.dropna(how='any')
```

```
                   A         B         C  D  F  E
2013-01-02  1.212112 -0.173215  0.119209  5  1  1
```

Filling missing data

```
In [56]: df1.fillna(value=5)
```

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

To get the boolean mask where values are `nan`

```
In [57]: pd.isnull(df1)
```

```
                A      B      C      D      F      E
2013-01-01  False  False  False  False   True  False
2013-01-02  False  False  False  False  False  False
2013-01-03  False  False  False  False  False   True
2013-01-04  False  False  False  False  False   True
```

5.5 Operations

See the *Basic section on Binary Ops*

5.5.1 Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic

```
In [58]: df.mean()
```

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

Same operation on the other axis

```
In [59]: df.mean(1)
```

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

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

```
In [60]: s = pd.Series([1,3,5,np.nan,6,8],index=dates).shift(2)

In [61]: s
```

```
2013-01-01   NaN
2013-01-02   NaN
2013-01-03     1
2013-01-04     3
2013-01-05     5
2013-01-06   NaN
Freq: D, dtype: float64
```

```
In [62]: df.sub(s,axis='index')
```

```
                   A         B         C   D   F
2013-01-01       NaN       NaN       NaN NaN NaN
2013-01-02       NaN       NaN       NaN NaN NaN
2013-01-03 -1.861849 -3.104569 -1.494929   4   1
2013-01-04 -2.278445 -3.706771 -4.039575   2   0
2013-01-05 -5.424972 -4.432980 -4.723768   0  -1
2013-01-06       NaN       NaN       NaN NaN NaN
```

5.5.2 Apply

Applying functions to the data

```
In [63]: df.apply(np.cumsum)
```

```
                   A         B         C   D   F
2013-01-01  0.000000  0.000000 -1.509059   5 NaN
2013-01-02  1.212112 -0.173215 -1.389850  10   1
2013-01-03  0.350263 -2.277784 -1.884779  15   3
2013-01-04  1.071818 -2.984555 -2.924354  20   6
2013-01-05  0.646846 -2.417535 -2.648122  25  10
2013-01-06 -0.026844 -2.303886 -4.126549  30  15
```

```
In [64]: df.apply(lambda x: x.max() - x.min())
```

```
A    2.073961
B    2.671590
C    1.785291
D    0.000000
F    4.000000
dtype: float64
```

5.5.3 Histogramming

See more at *Histogramming and Discretization*

```
In [65]: s = Series(np.random.randint(0,7,size=10))
```

```
In [66]: s
```

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

```
In [67]: s.value_counts()
```

```
4    5
```

```
6    2
2    2
1    1
dtype: int64
```

5.5.4 String Methods

See more at *Vectorized String Methods*

```
In [68]: s = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [69]: s.str.lower()
```

```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

5.6 Merge

5.6.1 Concat

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

See the *Merging section*

Concatenating pandas objects together

```
In [70]: df = pd.DataFrame(np.random.randn(10, 4))

In [71]: df
```

```
          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495
```

```
# break it into pieces
In [72]: pieces = [df[:3], df[3:7], df[7:]]

In [73]: concat(pieces)
```

```
          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495
```

5.6.2 Join

SQL style merges. See the *Database style joining*

```
In [74]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [75]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [76]: left

   key  lval
0  foo     1
1  foo     2

In [77]: right

   key  rval
0  foo     4
1  foo     5

In [78]: merge(left, right, on='key')

   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

5.6.3 Append

Append rows to a dataframe. See the *Appending*

```
In [79]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [80]: df

          A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
```

```
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758

In [81]: s = df.iloc[3]

In [82]: df.append(s, ignore_index=True)

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

5.7 Grouping

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

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

See the *Grouping section*

```
In [83]: df = pd.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 [84]: df

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

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

```
In [85]: df.groupby('A').sum()

            C        D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958
```

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

```
In [86]: df.groupby(['A','B']).sum()
```

```
                  C         D
A   B
bar one   -1.814470  2.395985
    three -0.595447  0.166599
    two   -0.392670 -0.136473
foo one   -1.195665 -0.616981
    three  1.928123 -1.623033
    two    2.414034  1.600434
```

5.8 Reshaping

See the section on *Hierarchical Indexing* and see the section on *Reshaping*).

5.8.1 Stack

```
In [87]: tuples = zip(*[['bar', 'bar', 'baz', 'baz',
   ....:                 'foo', 'foo', 'qux', 'qux'],
   ....:                ['one', 'two', 'one', 'two',
   ....:                 'one', 'two', 'one', 'two']])
   ....:

In [88]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [89]: df = pd.DataFrame(randn(8, 2), index=index, columns=['A', 'B'])

In [90]: df2 = df[:4]

In [91]: df2
```

```
                     A         B
first second
bar   one     0.029399 -0.542108
      two     0.282696 -0.087302
baz   one    -1.575170  1.771208
      two     0.816482  1.100230
```

The `stack` function "compresses" a level in the DataFrame's columns.

```
In [92]: stacked = df2.stack()

In [93]: stacked
```

```
first  second
bar    one     A    0.029399
               B   -0.542108
       two     A    0.282696
               B   -0.087302
baz    one     A   -1.575170
               B    1.771208
       two     A    0.816482
               B    1.100230
dtype: float64
```

With a "stacked" DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack` is `unstack`, which by default unstacks the **last level**:

```
In [94]: stacked.unstack()

                     A         B
first second
bar   one      0.029399 -0.542108
      two      0.282696 -0.087302
baz   one     -1.575170  1.771208
      two      0.816482  1.100230

In [95]: stacked.unstack(1)

second        one       two
first
bar   A  0.029399  0.282696
      B -0.542108 -0.087302
baz   A -1.575170  0.816482
      B  1.771208  1.100230

In [96]: stacked.unstack(0)

first          bar       baz
second
one    A  0.029399 -1.575170
       B -0.542108  1.771208
two    A  0.282696  0.816482
       B -0.087302  1.100230
```

5.8.2 Pivot Tables

See the section on *Pivot Tables*.

```
In [97]: df = DataFrame({'A' : ['one', 'one', 'two', 'three'] * 3,
   ....:                 'B' : ['A', 'B', 'C'] * 4,
   ....:                 'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
   ....:                 'D' : np.random.randn(12),
   ....:                 'E' : np.random.randn(12)})
   ....:

In [98]: df

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

We can produce pivot tables from this data very easily:

```
In [99]: pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
```

```
C             bar       foo
A     B
one   A -0.773723  1.418757
      B -0.029716 -1.879024
      C -1.146178  0.314665
three A  1.006160       NaN
      B       NaN -1.035018
      C  0.648740       NaN
two   A       NaN  0.100900
      B -1.170653       NaN
      C       NaN  0.536826
```

5.9 Time Series

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

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

In [101]: ts = pd.Series(randint(0, 500, len(rng)), index=rng)

In [102]: ts.resample('5Min', how='sum')
```

```
2012-01-01    25083
Freq: 5T, dtype: int64
```

Time zone representation

```
In [103]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [104]: ts = pd.Series(randn(len(rng)), rng)

In [105]: ts_utc = ts.tz_localize('UTC')

In [106]: ts_utc
```

```
2012-03-06 00:00:00+00:00    0.464000
2012-03-07 00:00:00+00:00    0.227371
2012-03-08 00:00:00+00:00   -0.496922
2012-03-09 00:00:00+00:00    0.306389
2012-03-10 00:00:00+00:00   -2.290613
Freq: D, dtype: float64
```

Convert to another time zone

```
In [107]: ts_utc.tz_convert('US/Eastern')
```

```
2012-03-05 19:00:00-05:00    0.464000
2012-03-06 19:00:00-05:00    0.227371
2012-03-07 19:00:00-05:00   -0.496922
2012-03-08 19:00:00-05:00    0.306389
2012-03-09 19:00:00-05:00   -2.290613
Freq: D, dtype: float64
```

Converting between time span representations

```
In [108]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [109]: ts = pd.Series(randn(len(rng)), index=rng)

In [110]: ts

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

In [111]: ps = ts.to_period()

In [112]: ps

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

In [113]: ps.to_timestamp()

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

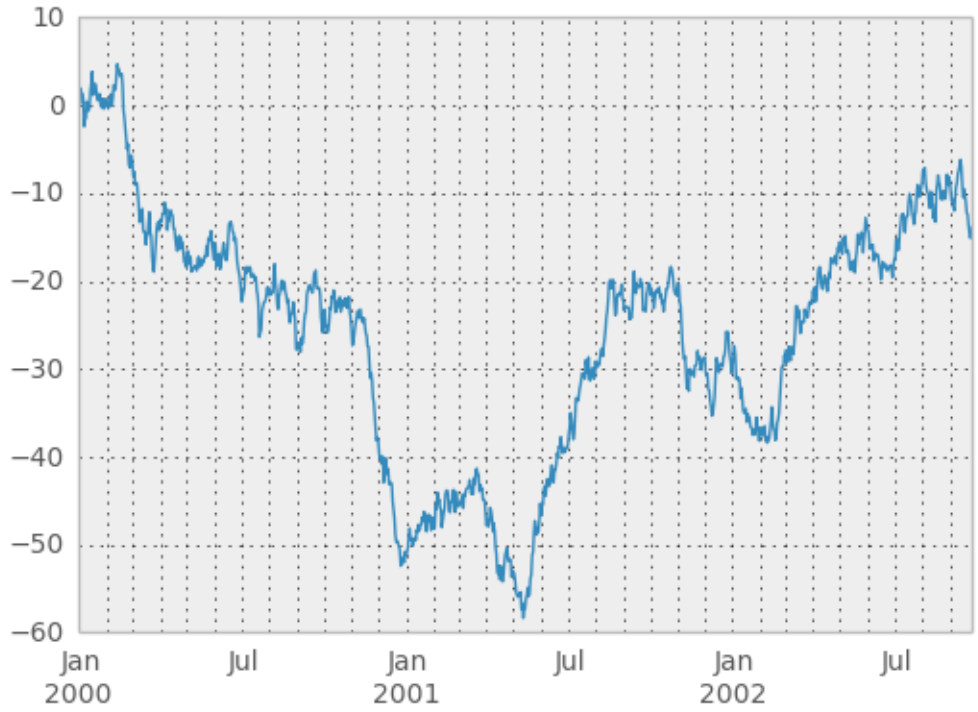
Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [114]: prng = period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [115]: ts = Series(randn(len(prng)), prng)

In [116]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [117]: ts.head()

1990-03-01 09:00   -0.902937
1990-06-01 09:00    0.068159
1990-09-01 09:00   -0.057873
1990-12-01 09:00   -0.368204
1991-03-01 09:00   -1.144073
Freq: H, dtype: float64
```

5.10 Plotting

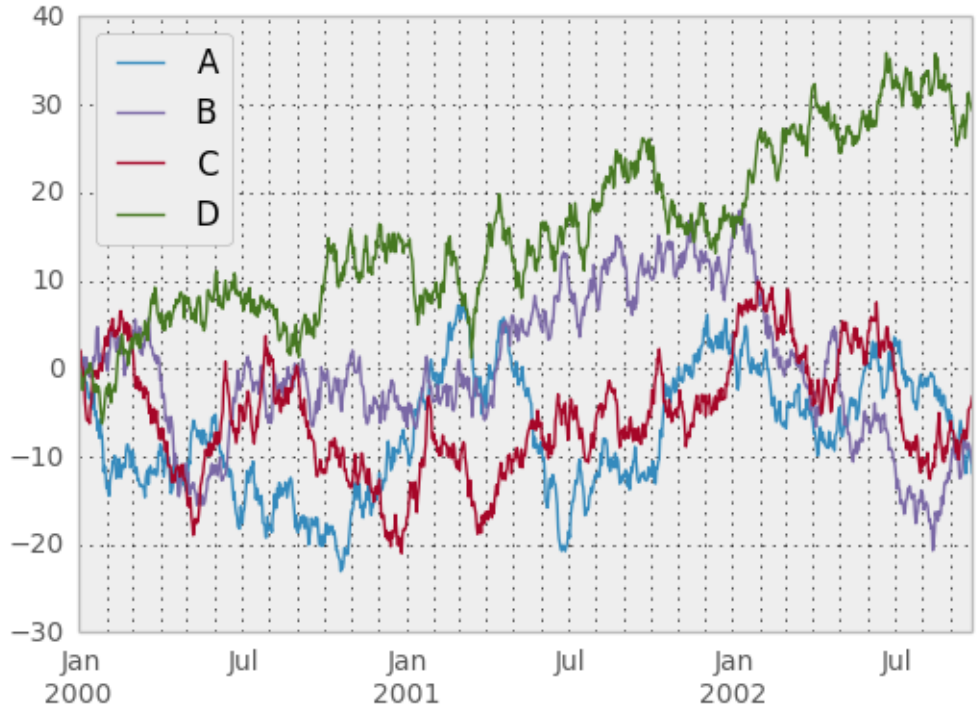
Plotting docs.

```
In [118]: ts = pd.Series(randn(1000), index=pd.date_range('1/1/2000', periods=1000))

In [119]: ts = ts.cumsum()

In [120]: ts.plot()
<matplotlib.axes.AxesSubplot at 0x5c9ed10>
```



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

```
In [121]: df = pd.DataFrame(randn(1000, 4), index=ts.index,
   .....:                   columns=['A', 'B', 'C', 'D'])
   .....:

In [122]: df = df.cumsum()

In [123]: plt.figure(); df.plot(); plt.legend(loc='best')
<matplotlib.legend.Legend at 0x5f01d50>
```



5.11 Getting Data In/Out

5.11.1 CSV

Writing to a csv file

```
In [124]: df.to_csv('foo.csv')
```

Reading from a csv file

```
In [125]: pd.read_csv('foo.csv')
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 0 to 999
Data columns (total 5 columns):
Unnamed: 0    1000  non-null values
A             1000  non-null values
B             1000  non-null values
C             1000  non-null values
D             1000  non-null values
dtypes: float64(4), object(1)
```

5.11.2 HDF5

Reading and writing to *HDFStores*

Writing to a HDF5 Store

```
In [126]: df.to_hdf('foo.h5','df')
```

Reading from a HDF5 Store

```
In [127]: read_hdf('foo.h5','df')
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1000 entries, 2000-01-01 00:00:00 to 2002-09-26 00:00:00
Freq: D
Data columns (total 4 columns):
A    1000  non-null values
B    1000  non-null values
C    1000  non-null values
D    1000  non-null values
dtypes: float64(4)
```

5.11.3 Excel

Reading and writing to *MS Excel*

Writing to an excel file

```
In [128]: df.to_excel('foo.xlsx', sheet_name='sheet1')
```

Reading from an excel file

```
In [129]: read_excel('foo.xlsx', 'sheet1', index_col=None, na_values=['NA'])
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1000 entries, 2000-01-01 00:00:00 to 2002-09-26 00:00:00
Data columns (total 4 columns):
A    1000  non-null values
B    1000  non-null values
C    1000  non-null values
D    1000  non-null values
dtypes: float64(4)
```

CHAPTER

SIX

COOKBOOK

This is a respository for *short and sweet* examples and links for useful pandas recipes. We encourage users to add to this documentation.

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

6.1 Idioms

These are some neat pandas `idioms`

How to do if-then-else?

How to split a frame with a boolean criterion?

How to select from a frame with complex criteria?

Select rows closest to a user defined numer

6.2 Selection

The *indexing* docs.

Indexing using both row labels and conditionals, see here

Use loc for label-oriented slicing and iloc positional slicing, see here

Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions, see here

Mask a panel by using `np.where` and then reconstructing the panel with the new masked values here

Using ~ to take the complement of a boolean array, see here

Efficiently creating columns using applymap

6.3 MultiIndexing

The *multindexing* docs.

Creating a multi-index from a labeled frame

6.3.1 Slicing

Slicing a multi-index with xs

Slicing a multi-index with xs #2

6.3.2 Sorting

Multi-index sorting

Partial Selection, the need for sortedness

6.3.3 Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

6.4 Missing Data

The *missing data* docs.

6.4.1 Replace

Using replace with backrefs

6.5 Grouping

The *grouping* docs.

Basic grouping with apply

Using get_group

Apply to different items in a group

Expanding Apply

Replacing values with groupby means

Sort by group with aggregation

Create multiple aggregated columns

Create a value counts column and reassign back to the DataFrame

6.5.1 Expanding Data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

6.5.2 Splitting

Splitting a frame

6.5.3 Pivot

The *Pivot* docs.

Partial sums and subtotals

Frequency table like plyr in R

6.5.4 Apply

Turning embeded lists into a multi-index frame

6.6 Timeseries

Between times

Using indexer between time

Vectorized Lookup

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

6.6.1 Resampling

The *Resample* docs.

TimeGrouping of values grouped across time

TimeGrouping #2

Using TimeGrouper and another grouping to create subgroups, then apply a custom function

Resampling with custom periods

Resample intraday frame without adding new days

Resample minute data

6.7 Merge

The *Concat* docs. The *Join* docs.

emulate R rbind

Self Join

How to set the index and join

KDB like asof join

Join with a criteria based on the values

6.8 Plotting

The *Plotting* docs.

Make Matplotlib look like R

Setting x-axis major and minor labels

Plotting multiple charts in an ipython notebook

Creating a multi-line plot

Plotting a heatmap

Annotate a time-series plot

6.9 Data In/Out

Performance comparison of SQL vs HDF5

6.9.1 CSV

The *CSV* docs

read_csv in action

appending to a csv

Reading a csv chunk-by-chunk

Reading the first few lines of a frame

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

Inferring dtypes from a file

Dealing with bad lines

Dealing with bad lines II

Reading CSV with Unix timestamps and converting to local timezone

Write a multi-row index CSV without writing duplicates

6.9.2 SQL

The *SQL* docs

Reading from databases with SQL

6.9.3 Excel

The *Excel* docs

Reading from a filelike handle

6.9.4 HDFStore

The *HDFStores* docs

Simple Queries with a Timestamp Index

Managing heteregenous data using a linked multiple table hierarchy

Merging on-disk tables with millions of rows

Deduplicating a large store by chunks, essentially a recusive reduction operation. Shows a function for taking in data from csv file and creating a store by chunks, with date parsing as well. See here

Large Data work flows

Reading in a sequence of files, then providing a global unique index to a store while appending

Groupby on a HDFStore

Troubleshoot HDFStore exceptions

Setting min_itemsize with strings

Storing Attributes to a group node

```
In [1]: df = DataFrame(np.random.randn(8,3))

In [2]: store = HDFStore('test.h5')

In [3]: store.put('df',df)

# you can store an arbitrary python object via pickle
In [4]: store.get_storer('df').attrs.my_attribute = dict(A = 10)

In [5]: store.get_storer('df').attrs.my_attribute
{'A': 10}
```

6.10 Computation

Numerical integration (sample-based) of a time series

6.11 Miscellaneous

The *Timedeltas* docs.

Operating with timedeltas

Create timedeltas with date differences

Adding days to dates in a dataframe

6.12 Aliasing Axis Names

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

```
In [6]: def set_axis_alias(cls, axis, alias):
   ...:     if axis not in cls._AXIS_NUMBERS:
   ...:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   ...:     cls._AXIS_ALIASES[alias] = axis
   ...:

In [7]: def clear_axis_alias(cls, axis, alias):
   ...:     if axis not in cls._AXIS_NUMBERS:
   ...:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   ...:     cls._AXIS_ALIASES.pop(alias,None)
   ...:

In [8]: set_axis_alias(DataFrame,'columns', 'myaxis2')

In [9]: df2 = DataFrame(randn(3,2),columns=['c1','c2'],index=['i1','i2','i3'])

In [10]: df2.sum(axis='myaxis2')

i1    0.981751
i2   -2.754270
i3   -1.528539
dtype: float64

In [11]: clear_axis_alias(DataFrame,'columns', 'myaxis2')
```

CHAPTER

SEVEN

INTRO TO DATA STRUCTURES

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

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

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

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

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

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

When using pandas, we recommend the following import convention:

```
import pandas as pd
```

7.1 Series

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

In [5]: s

a   -1.344
b    0.845
c    1.076
d   -0.109
e    1.644
dtype: float64

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

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

0   -1.469
1    0.357
2   -0.675
3   -1.777
4   -0.969
dtype: float64
```

Note: Starting in v0.8.0, pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

If `data` is a dict, if **index** is passed the values in data corresponding to the labels in the index will be pulled out. Otherwise, an index will be constructed from the sorted keys of the dict, if possible.

```
In [8]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [9]: Series(d)

a    0
b    1
c    2
dtype: float64

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

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

Note: NaN (not a number) is the standard missing data marker used in pandas

---

From scalar value If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

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

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

7.1.1 Series is ndarray-like

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 [12]: s[0]
-1.3443118127316671

In [13]: s[:3]

a   -1.344
b    0.845
c    1.076
dtype: float64

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

c    1.076
e    1.644
dtype: float64

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

e    1.644
d   -0.109
b    0.845
dtype: float64

In [16]: np.exp(s)

a    0.261
b    2.328
c    2.932
d    0.897
e    5.174
dtype: float64
```

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

7.1.2 Series is dict-like

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

```
In [17]: s['a']
-1.3443118127316671

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

In [19]: s
```

```
a    -1.344
b     0.845
c     1.076
d    -0.109
e    12.000
dtype: float64

In [20]: 'e' in s
True

In [21]: 'f' in s
False
```

If a label is not contained, an exception is raised:

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

Using the `get` method, a missing label will return None or specified default:

```
In [22]: s.get('f')

In [23]: s.get('f', np.nan)
nan
```

7.1.3 Vectorized operations and label alignment with Series

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

```
In [24]: s + s

a    -2.689
b     1.690
c     2.152
d    -0.218
e    24.000
dtype: float64

In [25]: s * 2

a    -2.689
b     1.690
c     2.152
d    -0.218
e    24.000
dtype: float64

In [26]: np.exp(s)

a         0.261
b         2.328
c         2.932
d         0.897
e    162754.791
dtype: float64
```

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

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

a      NaN
b    1.690
c    2.152
d   -0.218
e      NaN
dtype: float64
```

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

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

7.1.4 Name attribute

Series can also have a `name` attribute:

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

In [29]: s

0   -1.295
1    0.414
2    0.277
3   -0.472
4   -0.014
Name: something, dtype: float64

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

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

7.2 DataFrame

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

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray

- Structured or record ndarray
- A `Series`
- Another `DataFrame`

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

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

7.2.1 From dict of Series or dicts

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

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

In [32]: df = DataFrame(d)

In [33]: df

   one  two
a    1    1
b    2    2
c    3    3
d  NaN    4

In [34]: DataFrame(d, index=['d', 'b', 'a'])

   one  two
d  NaN    4
b    2    2
a    1    1

In [35]: DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])

   two three
d    4   NaN
b    2   NaN
a    1   NaN
```

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

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

```
In [36]: df.index
Index([u'a', u'b', u'c', u'd'], dtype=object)

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

7.2.2 From dict of ndarrays / lists

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

```
In [38]: d = {'one' : [1., 2., 3., 4.],
   ....:      'two' : [4., 3., 2., 1.]}
   ....:

In [39]: DataFrame(d)

   one  two
0    1    4
1    2    3
2    3    2
3    4    1

In [40]: DataFrame(d, index=['a', 'b', 'c', 'd'])

   one  two
a    1    4
b    2    3
c    3    2
d    4    1
```

7.2.3 From structured or record array

This case is handled identically to a dict of arrays.

```
In [41]: data = np.zeros((2,),dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])

In [42]: data[:] = [(1,2.,'Hello'),(2,3.,"World")]

In [43]: DataFrame(data)

   A  B      C
0  1  2  Hello
1  2  3  World

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

        A  B      C
first   1  2  Hello
second  2  3  World

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

       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.

7.2.4 From a list of dicts

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

In [47]: DataFrame(data2)

   a   b   c
0  1   2 NaN
1  5  10  20

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

        a   b   c
first   1   2 NaN
second  5  10  20

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

   a   b
0  1   2
1  5  10
```

7.2.5 From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Missing Data

Much more will be said on this topic in the *Missing data* section. To construct a DataFrame with missing data, use `np.nan` for those values which are missing. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

7.2.6 Alternate Constructors

DataFrame.from_dict

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

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

```
In [50]: data

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

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

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

DataFrame.from_items

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

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

   A  B
0  1  4
1  2  5
2  3  6
```

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

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

   one  two  three
A    1    2      3
B    4    5      6
```

7.2.7 Column selection, addition, deletion

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

```
In [54]: df['one']

a     1
b     2
c     3
d   NaN
Name: one, dtype: float64

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

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

In [57]: df

   one  two  three   flag
a    1    1      1  False
b    2    2      4  False
c    3    3      9   True
d  NaN    4    NaN  False
```

Columns can be deleted or popped like with a dict:

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

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

In [60]: df
```

```
   one   flag
a    1  False
b    2  False
c    3   True
d  NaN  False
```

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

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

In [62]: df

   one   flag  foo
a    1  False  bar
b    2  False  bar
c    3   True  bar
d  NaN  False  bar
```

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

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

In [64]: df

   one   flag  foo  one_trunc
a    1  False  bar          1
b    2  False  bar          2
c    3   True  bar        NaN
d  NaN  False  bar        NaN
```

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

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

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

In [66]: df

   one  bar   flag  foo  one_trunc
a    1    1  False  bar          1
b    2    2  False  bar          2
c    3    3   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

7.2.8 Indexing / Selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[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 [67]: df.loc['b']
```

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

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

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

For a more exhaustive treatment of more sophisticated label-based indexing and slicing, see the *section on indexing*. We will address the fundamentals of reindexing / conforming to new sets of lables in the *section on reindexing*.

7.2.9 Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on **both the columns and the index (row labels)**. Again, the resulting object will have the union of the column and row labels.

```
In [69]: df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])
```

```
In [70]: df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])
```

```
In [71]: df + df2
```

```
       A      B      C   D
0 -1.473 -0.626 -0.773 NaN
1  0.073 -0.519  2.742 NaN
2  1.744 -1.325  0.075 NaN
3 -1.366 -1.238 -1.782 NaN
4  0.275 -0.613 -2.263 NaN
5  1.263  2.338  1.260 NaN
6 -1.216  3.371 -1.992 NaN
7    NaN    NaN    NaN NaN
8    NaN    NaN    NaN NaN
9    NaN    NaN    NaN NaN
```

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

```
In [72]: df - df.iloc[0]
```

```
       A      B      C      D
0  0.000  0.000  0.000  0.000
1  1.168 -1.200  3.489  0.536
2  1.703 -1.164  0.697 -0.485
3  1.176  0.138  0.096 -0.972
4 -0.825  1.136 -0.514 -2.309
5  1.970  1.030  1.493 -0.020
6 -1.849  0.981 -1.084 -1.306
```

```
7  0.284  0.552 -0.296 -2.123
8  1.132 -1.275  0.195 -1.017
9  0.265  0.702  1.265  0.064
```

In the special case of working with time series data, if the Series is a TimeSeries (which it will be automatically if the index contains datetime objects), and the DataFrame index also contains dates, the broadcasting will be column-wise:

```
In [73]: index = date_range('1/1/2000', periods=8)

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

In [75]: df

                A      B      C
2000-01-01  3.357 -0.317 -1.236
2000-01-02  0.896 -0.488 -0.082
2000-01-03 -2.183  0.380  0.085
2000-01-04  0.432  1.520 -0.494
2000-01-05  0.600  0.274  0.133
2000-01-06 -0.024  2.410  1.451
2000-01-07  0.206 -0.252 -2.214
2000-01-08  1.063  1.266  0.299

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

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

            A      B      C
2000-01-01  0 -3.675 -4.594
2000-01-02  0 -1.384 -0.978
2000-01-03  0  2.563  2.268
2000-01-04  0  1.088 -0.926
2000-01-05  0 -0.326 -0.467
2000-01-06  0  2.434  1.474
2000-01-07  0 -0.458 -2.420
2000-01-08  0  0.203 -0.764
```

Warning:

```
df - df['A']
```

is now deprecated and will be removed in a future release. The preferred way to replicate this behavior is

```
df.sub(df['A'], axis=0)
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [78]: df * 5 + 2

                A       B      C
2000-01-01  18.787   0.413 -4.181
2000-01-02   6.481  -0.438  1.589
2000-01-03  -8.915   3.902  2.424
2000-01-04   4.162   9.600 -0.468
2000-01-05   5.001   3.371  2.664
```

```
2000-01-06   1.882  14.051  9.253
2000-01-07   3.030   0.740 -9.068
2000-01-08   7.317   8.331  3.497
```

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

```
                 A      B       C
2000-01-01   0.298 -3.150  -0.809
2000-01-02   1.116 -2.051 -12.159
2000-01-03  -0.458  2.629  11.786
2000-01-04   2.313  0.658  -2.026
2000-01-05   1.666  3.647   7.525
2000-01-06 -42.215  0.415   0.689
2000-01-07   4.853 -3.970  -0.452
2000-01-08   0.940  0.790   3.340
```

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

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

Boolean operators work as well:

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

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

```
In [83]: df1 & df2
```

```
       a      b
0  False  False
1  False   True
2   True  False
```

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

```
      a     b
0  True  True
1  True  True
2  True  True
```

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

```
       a      b
0   True   True
1   True  False
2  False   True
```

```
In [86]: -df1
```

```
       a      b
0  False   True
```

```
1   True  False
2  False  False
```

7.2.10 Transposing

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

```
# only show the first 5 rows
In [87]: df[:5].T
```

```
   2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A       3.357       0.896      -2.183       0.432       0.600
B      -0.317      -0.488       0.380       1.520       0.274
C      -1.236      -0.082       0.085      -0.494       0.133
```

7.2.11 DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on DataFrame, assuming the data within are numeric:

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

```
                A       B      C
2000-01-01  28.715   0.728  0.290
2000-01-02   2.450   0.614  0.921
2000-01-03   0.113   1.463  1.089
2000-01-04   1.541   4.572  0.610
2000-01-05   1.822   1.316  1.142
2000-01-06   0.977  11.136  4.265
2000-01-07   1.229   0.777  0.109
2000-01-08   2.896   3.547  1.349
```

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

```
array([[ 3.3574, -0.3174, -1.2363],
       [ 0.8962, -0.4876, -0.0822],
       [-2.1829,  0.3804,  0.0848],
       [ 0.4324,  1.52  , -0.4937],
       [ 0.6002,  0.2742,  0.1329],
       [-0.0237,  2.4102,  1.4505],
       [ 0.2061, -0.2519, -2.2136],
       [ 1.0633,  1.2661,  0.2994]])
```

The dot method on DataFrame implements matrix multiplication:

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

```
        A       B      C
A  18.562  -0.274 -4.715
B  -0.274  10.344  4.184
C  -4.715   4.184  8.897
```

Similarly, the dot method on Series implements dot product:

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

```
In [92]: s1.dot(s1)
255
```

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

7.2.12 Console display

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 [93]: baseball = read_csv('data/baseball.csv')

In [94]: print baseball
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 88641 to 89534
Data columns (total 22 columns):
id       100  non-null values
year     100  non-null values
stint    100  non-null values
team     100  non-null values
lg       100  non-null values
g        100  non-null values
ab       100  non-null values
r        100  non-null values
h        100  non-null values
X2b      100  non-null values
X3b      100  non-null values
hr       100  non-null values
rbi      100  non-null values
sb       100  non-null values
cs       100  non-null values
bb       100  non-null values
so       100  non-null values
ibb      100  non-null values
hbp      100  non-null values
sh       100  non-null values
sf       100  non-null values
gidp     100  non-null values
dtypes: float64(9), int64(10), object(3)
```

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

```
In [95]: print baseball.iloc[-20:, :12].to_string()
             id  year  stint team  lg    g   ab   r    h  X2b  X3b  hr
89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0   1
89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0   0
89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2  12
89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0  10
89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0  24
89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0   0
89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0   0
89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1   6
89497  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
```

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

```
In [96]: DataFrame(randn(3, 12))
```

```
          0         1         2         3         4         5         6   \
0 -0.863838  0.408204 -1.048089 -0.025747 -0.988387  0.094055  1.262731
1  0.369374 -0.034571 -2.484478 -0.281461  0.030711  0.109121  1.126203
2 -1.071357  0.441153  2.353925  0.583787  0.221471 -0.744471  0.758527
          7         8         9         10        11
0  1.289997  0.082423 -0.055758  0.536580 -0.489682
1 -0.977349  1.474071 -0.064034 -1.282782  0.781836
2  1.729689 -0.964980 -0.845696 -1.340896  1.846883
```

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

```
In [97]: set_option('line_width', 40) # default is 80

In [98]: DataFrame(randn(3, 12))
```

```
          0         1         2   \
0 -1.328865  1.682706 -1.717693
1  0.306996 -0.028665  0.384316
2 -1.137707 -0.891060 -0.693921
          3         4         5   \
0  0.888782  0.228440  0.901805
1  1.574159  1.588931  0.476720
2  1.613616  0.464000  0.227371
          6         7         8   \
0  1.171216  0.520260 -1.197071
1  0.473424 -0.242861 -0.014805
2 -0.496922  0.306389 -2.290613
          9         10        11
0 -1.066969 -0.303421 -0.858447
1 -0.284319  0.650776 -1.461665
2 -1.134623 -1.561819 -0.260838
```

You can also disable this feature via the `expand_frame_repr` option:

```
In [99]: set_option('expand_frame_repr', False)

In [100]: DataFrame(randn(3, 12))
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 12 columns):
0     3  non-null values
1     3  non-null values
2     3  non-null values
3     3  non-null values
4     3  non-null values
```

```
5     3  non-null values
6     3  non-null values
7     3  non-null values
8     3  non-null values
9     3  non-null values
10    3  non-null values
11    3  non-null values
dtypes: float64(12)
```

7.2.13 DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like attributes:

```
In [101]: df = DataFrame({'foo1' : np.random.randn(5),
   .....:                 'foo2' : np.random.randn(5)})
   .....:

In [102]: df

       foo1      foo2
0  0.967661 -0.681087
1 -1.057909  0.377953
2  1.375020  0.493672
3 -0.928797 -2.461467
4 -0.308853 -1.553902

In [103]: df.foo1

0    0.967661
1   -1.057909
2    1.375020
3   -0.928797
4   -0.308853
Name: foo1, dtype: float64
```

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

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

7.3 Panel

Panel is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

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

Construction of Panels works about like you would expect:

7.3.1 From 3D ndarray with optional axis labels

```
In [104]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:            major_axis=date_range('1/1/2000', periods=5),
   .....:            minor_axis=['A', 'B', 'C', 'D'])
   .....:

In [105]: wp

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

7.3.2 From dict of DataFrame objects

```
In [106]: data = {'Item1' : DataFrame(randn(4, 3)),
   .....:         'Item2' : DataFrame(randn(4, 2))}
   .....:

In [107]: Panel(data)

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

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

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

Parameter	Default	Description
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 [108]: Panel.from_dict(data, orient='minor')

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

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

```
In [109]: df = DataFrame({'a': ['foo', 'bar', 'baz'],
   .....:                 'b': np.random.randn(3)})
   .....:

In [110]: df
```

```
     a         b
0  foo -1.004168
1  bar -1.377627
2  baz  0.499281

In [111]: data = {'item1': df, 'item2': df}

In [112]: panel = Panel.from_dict(data, orient='minor')

In [113]: panel['a']

  item1 item2
0   foo   foo
1   bar   bar
2   baz   baz

In [114]: panel['b']

      item1     item2
0 -1.004168 -1.004168
1 -1.377627 -1.377627
2  0.499281  0.499281

In [115]: panel['b'].dtypes

item1    float64
item2    float64
dtype: object
```

Note: Unfortunately Panel, being less commonly used than Series and DataFrame, has been slightly neglected feature-wise. A number of methods and options available in DataFrame are not available in Panel. This will get worked on, of course, in future releases. And faster if you join me in working on the codebase.

7.3.3 From DataFrame using `to_panel` method

This method was introduced in v0.7 to replace `LongPanel.to_long`, and converts a DataFrame with a two-level index to a Panel.

```
In [116]: midx = MultiIndex(levels=[['one', 'two'], ['x','y']], labels=[[1,1,0,0],[1,0,1,0]])

In [117]: df = DataFrame({'A' : [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [118]: df.to_panel()

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: A to B
Major_axis axis: one to two
Minor_axis axis: x to y
```

7.3.4 Item selection / addition / deletion

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

```
In [119]: wp['Item1']

                   A         B         C         D
2000-01-01  2.015523 -1.833722  1.771740 -0.670027
2000-01-02  0.049307 -0.521493 -3.201750  0.792716
2000-01-03  0.146111  1.903247 -0.747169 -0.309038
2000-01-04  0.393876  1.861468  0.936527  1.255746
2000-01-05 -2.655452  1.219492  0.062297 -0.110388

In [120]: wp['Item3'] = wp['Item1'] / wp['Item2']
```

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

7.3.5 Transposing

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

```
In [121]: wp.transpose(2, 0, 1)

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

7.3.6 Indexing / Selection

Operation	Syntax	Result
Select item	`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 [122]: wp['Item1']

                   A         B         C         D
2000-01-01  2.015523 -1.833722  1.771740 -0.670027
2000-01-02  0.049307 -0.521493 -3.201750  0.792716
2000-01-03  0.146111  1.903247 -0.747169 -0.309038
2000-01-04  0.393876  1.861468  0.936527  1.255746
2000-01-05 -2.655452  1.219492  0.062297 -0.110388

In [123]: wp.major_xs(wp.major_axis[2])

      Item1     Item2     Item3
A  0.146111 -1.139050 -0.128275
B  1.903247  0.660342  2.882214
C -0.747169  0.464794 -1.607526
D -0.309038 -0.309337  0.999035

In [124]: wp.minor_axis
Index([u'A', u'B', u'C', u'D'], dtype=object)
```

```
In [125]: wp.minor_xs('C')

                Item1     Item2      Item3
2000-01-01  1.771740  0.077849  22.758618
2000-01-02 -3.201750  0.503703  -6.356422
2000-01-03 -0.747169  0.464794  -1.607526
2000-01-04  0.936527 -0.643834  -1.454609
2000-01-05  0.062297  0.787872   0.079070
```

7.3.7 Squeezing

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

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

                   A         B         C         D
2000-01-01  2.015523 -1.833722  1.771740 -0.670027
2000-01-02  0.049307 -0.521493 -3.201750  0.792716
2000-01-03  0.146111  1.903247 -0.747169 -0.309038
2000-01-04  0.393876  1.861468  0.936527  1.255746
2000-01-05 -2.655452  1.219492  0.062297 -0.110388

In [127]: wp.reindex(items=['Item1'],minor=['B']).squeeze()

2000-01-01   -1.833722
2000-01-02   -0.521493
2000-01-03    1.903247
2000-01-04    1.861468
2000-01-05    1.219492
Freq: D, Name: B, dtype: float64
```

7.3.8 Conversion to DataFrame

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

```
In [128]: panel = Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
   .....:               major_axis=date_range('1/1/2000', periods=5),
   .....:               minor_axis=['a', 'b', 'c', 'd'])
   .....:

In [129]: panel.to_frame()

                       one       two     three
major      minor
2000-01-01 a     -1.405256 -1.157886  0.086926
           b      0.162565 -0.551865 -0.445645
           c     -0.067785  1.592673 -0.217503
           d     -1.260006  1.559318 -1.420361
2000-01-02 a     -1.132896  1.562443 -0.015601
           b     -2.006481  0.763264 -1.150641
           c      0.301016  0.162027 -0.798334
           d      0.059117 -0.902704 -0.557697
2000-01-03 a      1.138469  1.106010  0.381353
           b     -2.400634 -0.199234  1.337122
```

```
           c     -0.280853  0.458265 -1.531095
           d      0.025653  0.491048  1.331458
2000-01-04 a     -1.386071  0.128594 -0.571329
           b      0.863937  1.147862 -0.026671
           c      0.252462 -1.256860 -1.085663
           d      1.500571  0.563637 -1.114738
2000-01-05 a      1.053202 -2.417312 -0.058216
           b     -2.338595  0.972827 -0.486768
           c     -0.374279  0.041293  1.685148
           d     -2.359958  1.129659  0.112572
```

7.4 Panel4D (Experimental)

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

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

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

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

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

7.4.1 From 4D ndarray with optional axis labels

```
In [130]: p4d = Panel4D(randn(2, 2, 5, 4),
   .....:               labels=['Label1','Label2'],
   .....:               items=['Item1', 'Item2'],
   .....:               major_axis=date_range('1/1/2000', periods=5),
   .....:               minor_axis=['A', 'B', 'C', 'D'])
   .....:

In [131]: p4d

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

7.4.2 From dict of Panel objects

```
In [132]: data = { 'Label1' : Panel({ 'Item1' : DataFrame(randn(4, 3)) }),
   .....:          'Label2' : Panel({ 'Item2' : DataFrame(randn(4, 2)) }) }
   .....:
```

```
In [133]: Panel4D(data)

<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 4 (major_axis) x 3 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

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

7.4.3 Slicing

Slicing works in a similar manner to a Panel. `[]` slices the first dimension. `.ix` allows you to slice abitrarily and get back lower dimensional objects

```
In [134]: p4d['Label1']

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

4D -> Panel

```
In [135]: p4d.ix[:,:,:,'A']

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 5 (minor_axis)
Items axis: Label1 to Label2
Major_axis axis: Item1 to Item2
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

4D -> DataFrame

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

         Label1    Label2
Item1 -1.495309 -0.739776
Item2  1.103949  0.403776
```

4D -> Series

```
In [137]: p4d.ix[:,0,0,'A']

Label1   -1.495309
Label2   -0.739776
Name: A, dtype: float64
```

7.4.4 Transposing

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

```
In [138]: p4d.transpose(3, 2, 1, 0)

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

7.5 PanelND (Experimental)

PanelND is a module with a set of factory functions to enable a user to construct N-dimensional named containers like Panel4D, with a custom set of axis labels. Thus a domain-specific container can easily be created.

The following creates a Panel5D. A new panel type object must be sliceable into a lower dimensional object. Here we slice to a Panel4D.

```
In [139]: from pandas.core import panelnd

In [140]: Panel5D = panelnd.create_nd_panel_factory(
   .....:     klass_name   = 'Panel5D',
   .....:     axis_orders  = [ 'cool', 'labels','items','major_axis','minor_axis'],
   .....:     axis_slices  = { 'labels' : 'labels', 'items' : 'items',
   .....:                      'major_axis' : 'major_axis', 'minor_axis' : 'minor_axis' },
   .....:     slicer       = Panel4D,
   .....:     axis_aliases = { 'major' : 'major_axis', 'minor' : 'minor_axis' },
   .....:     stat_axis    = 2)
   .....:

In [141]: p5d = Panel5D(dict(C1 = p4d))

In [142]: p5d

<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 1 (cool) x 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Cool axis: C1 to C1
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

# print a slice of our 5D
In [143]: p5d.ix['C1',:,:,0:3,:]

<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 3 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to D

# transpose it
In [144]: p5d.transpose(1,2,3,4,0)

<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 2 (cool) x 2 (labels) x 5 (items) x 4 (major_axis) x 1 (minor_axis)
```

```
Cool axis: Label1 to Label2
Labels axis: Item1 to Item2
Items axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Major_axis axis: A to D
Minor_axis axis: C1 to C1

# look at the shape & dim
In [145]: p5d.shape
(1, 2, 2, 5, 4)

In [146]: p5d.ndim
5
```

CHAPTER

EIGHT

ESSENTIAL BASIC FUNCTIONALITY

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

```
In [1]: index = date_range('1/1/2000', periods=8)

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

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

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

8.1 Head and Tail

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

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

In [6]: long_series.head()

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

In [7]: long_series.tail(3)

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

8.2 Attributes and the raw ndarray(s)

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

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- Axis labels
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*
  - **Panel**: *items*, *major_axis*, and *minor_axis*

Note, **these attributes can be safely assigned to**!

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

                   A         B         C
2000-01-01  0.232465 -0.789552 -0.364308
2000-01-02 -0.534541  0.822239 -0.443109

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

In [10]: df

                   a         b         c
2000-01-01  0.232465 -0.789552 -0.364308
2000-01-02 -0.534541  0.822239 -0.443109
2000-01-03 -2.119990 -0.460149  1.813962
2000-01-04 -1.053571  0.009412 -0.165966
2000-01-05 -0.848662 -0.495553 -0.176421
2000-01-06 -0.423595 -1.035433 -1.035374
2000-01-07 -2.369079  0.524408 -0.871120
2000-01-08  1.585433  0.039501  2.274101
```

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

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

In [12]: df.values

array([[ 0.2325, -0.7896, -0.3643],
       [-0.5345,  0.8222, -0.4431],
       [-2.12  , -0.4601,  1.814 ],
       [-1.0536,  0.0094, -0.166 ],
       [-0.8487, -0.4956, -0.1764],
       [-0.4236, -1.0354, -1.0354],
       [-2.3691,  0.5244, -0.8711],
       [ 1.5854,  0.0395,  2.2741]])

In [13]: wp.values

array([[[-1.1181,  0.4313,  0.5547, -1.3336],
        [-0.3322, -0.4859,  1.7259,  1.7993],
        [-0.9689, -0.7795, -2.0007, -1.8666],
        [-1.1013,  1.9575,  0.0589,  0.7581],
        [ 0.0766, -0.5485, -0.1605, -0.3778]],
       [[ 0.2499, -0.3413, -0.2726, -0.2774],
        [-1.1029,  0.1003, -1.6028,  0.9201],
```

```
       [-0.6439,  0.0603, -0.4349, -0.4943],
       [ 0.738 ,  0.4516,  0.3341, -0.7871],
       [ 0.6514, -0.7419,  1.1939, -2.3958]]])
```

If a DataFrame or Panel contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame's columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

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

8.3 Accelerated operations

Pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library (starting in 0.11.0) and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have `nans`.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

Operation	0.11.0 (ms)	Prior Vern (ms)	Ratio to Prior
`df1 > df2`	13.32	125.35	0.1063
`df1 * df2`	21.71	36.63	0.5928
`df1 + df2`	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section *Recommended Dependencies* for more installation info.

8.4 Flexible binary operations

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

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

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

8.4.1 Matching / broadcasting behavior

DataFrame has the methods **add, sub, mul, div** and related functions **radd, rsub, ...** for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [14]: d = {'one' : Series(randn(3), index=['a', 'b', 'c']),
   ....:      'two' : Series(randn(4), index=['a', 'b', 'c', 'd']),
   ....:      'three' : Series(randn(3), index=['b', 'c', 'd'])}
   ....:

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

```
In [16]: df

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

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

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

In [19]: df.sub(row, axis='columns')

        one     three       two
a -0.810701       NaN -0.724777
b  0.000000  0.000000  0.000000
c -0.340950  0.205973 -0.640340
d       NaN  0.186952 -0.533630

In [20]: df.sub(row, axis=1)

        one     three       two
a -0.810701       NaN -0.724777
b  0.000000  0.000000  0.000000
c -0.340950  0.205973 -0.640340
d       NaN  0.186952 -0.533630

In [21]: df.sub(column, axis='index')

        one     three  two
a -0.614265       NaN    0
b -0.528341 -0.992033    0
c -0.228950 -0.145720    0
d       NaN -0.271451    0

In [22]: df.sub(column, axis=0)

        one     three  two
a -0.614265       NaN    0
b -0.528341 -0.992033    0
c -0.228950 -0.145720    0
d       NaN -0.271451    0
```

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

```
In [23]: major_mean = wp.mean(axis='major')

In [24]: major_mean

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

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

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

And similarly for `axis="items"` and `axis="minor"`.

Note: I could be convinced to make the **axis** argument in the DataFrame methods match the broadcasting behavior of Panel. Though it would require a transition period so users can change their code...

8.4.2 Missing data / operations with fill values

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

```
In [26]: df

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

In [27]: df2

        one     three       two
a -0.701368  1.000000 -0.087103
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
d       NaN -0.167407  0.104044

In [28]: df + df2

        one     three       two
a -1.402736       NaN -0.174206
b  0.218666 -0.708719  1.275347
c -0.463233 -0.296773 -0.005333
d       NaN -0.334814  0.208088

In [29]: df.add(df2, fill_value=0)

        one     three       two
a -1.402736  1.000000 -0.174206
b  0.218666 -0.708719  1.275347
c -0.463233 -0.296773 -0.005333
d       NaN -0.334814  0.208088
```

8.4.3 Flexible Comparisons

Starting in v0.8, pandas introduced binary comparison methods eq, ne, lt, gt, le, and ge to Series and DataFrame whose behavior is analogous to the binary arithmetic operations described above:

```
In [30]: df.gt(df2)

     one  three    two
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [31]: df2.ne(df)

     one  three    two
a  False   True  False
b  False  False  False
c  False  False  False
d   True  False  False
```

8.4.4 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 [32]: df1 = DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
   ....:                  'B' : [np.nan, 2., 3., np.nan, 6.]})
   ....:

In [33]: df2 = DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
   ....:                  'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
   ....:

In [34]: df1

    A   B
0   1 NaN
1 NaN   2
2   3   3
3   5 NaN
4 NaN   6

In [35]: df2

    A   B
0   5 NaN
1   2 NaN
2   4   3
3 NaN   4
4   3   6
5   7   8
```

```
In [36]: df1.combine_first(df2)

   A   B
0  1 NaN
1  2   2
2  3   3
3  5   4
4  3   6
5  7   8
```

8.4.5 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 [37]: combiner = lambda x, y: np.where(isnull(x), y, x)

In [38]: df1.combine(df2, combiner)

   A   B
0  1 NaN
1  2   2
2  3   3
3  5   4
4  3   6
5  7   8
```

8.5 Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*, and *Panel*. Most of these are aggregations (hence producing a lower-dimensional result) like **sum**, **mean**, and **quantile**, but some of them, like **cumsum** and **cumprod**, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like *ndarray.{sum, std, ...}*, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: "index" (axis=0, default), "columns" (axis=1)
- **Panel**: "items" (axis=0), "major" (axis=1, default), "minor" (axis=2)

For example:

```
In [39]: df

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

In [40]: df.mean(0)

one     -0.274551
```

```
three   -0.223384
two      0.162987
dtype: float64

In [41]: df.mean(1)

a   -0.394235
b    0.130882
c   -0.127557
d   -0.031682
dtype: float64
```

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

```
In [42]: df.sum(0, skipna=False)

one           NaN
three         NaN
two      0.651948
dtype: float64

In [43]: df.sum(axis=1, skipna=True)

a   -0.788471
b    0.392647
c   -0.382670
d   -0.063363
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [44]: ts_stand = (df - df.mean()) / df.std()

In [45]: ts_stand.std()

one      1
three    1
two      1
dtype: float64

In [46]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [47]: xs_stand.std(1)

a    1
b    1
c    1
d    1
dtype: float64
```

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

```
In [48]: df.cumsum()

        one     three       two
a -0.701368       NaN -0.087103
b -0.592035 -0.354359  0.550570
c -0.823652 -0.502746  0.547904
```

```
d       NaN -0.670153  0.651948
```

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 [49]: np.mean(df['one'])
-0.27455055654271204

In [50]: np.mean(df['one'].values)
nan
```

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

```
In [51]: series = Series(randn(500))

In [52]: series[20:500] = np.nan

In [53]: series[10:20]  = 5

In [54]: series.nunique()
11
```

8.5.1 Summarizing data: describe

There is a convenient `describe` function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [55]: series = Series(randn(1000))

In [56]: series[::2] = np.nan

In [57]: series.describe()

count    500.000000
```

```
mean      -0.019898
std        1.019180
min       -2.628792
25%       -0.649795
50%       -0.059405
75%        0.651932
max        3.240991
dtype: float64
```

```
In [58]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [59]: frame.ix[::2] = np.nan

In [60]: frame.describe()
```

```
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.051388    0.053476   -0.035612    0.015388    0.057804
std      0.989217    0.995961    0.977047    0.968385    1.022528
min     -3.224136   -2.606460   -2.762875   -2.961757   -2.829100
25%     -0.657420   -0.597123   -0.688961   -0.695019   -0.738097
50%      0.042928    0.018837   -0.071830   -0.011326    0.073287
75%      0.702445    0.693542    0.600454    0.680924    0.807670
max      3.034008    3.104512    2.812028    2.623914    3.542846
```

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

```
In [61]: s = Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [62]: s.describe()
```

```
count     9
unique    4
top       a
freq      5
dtype: object
```

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

8.5.2 Index of Min/Max Values

The `idxmin` and `idxmax` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [63]: s1 = Series(randn(5))

In [64]: s1
```

```
0   -0.574018
1    0.668292
2    0.303418
3   -1.190271
4    0.138399
dtype: float64
```

```
In [65]: s1.idxmin(), s1.idxmax()
(3, 1)

In [66]: df1 = DataFrame(randn(5,3), columns=['A','B','C'])

In [67]: df1

          A         B         C
0 -0.184355 -1.054354 -1.613138
1 -0.050807 -2.130168 -1.852271
2  0.455674  2.571061 -1.152538
3 -1.638940 -0.364831 -0.348520
4  0.202856  0.777088 -0.358316

In [68]: df1.idxmin(axis=0)

A    3
B    1
C    1
dtype: int64

In [69]: df1.idxmax(axis=1)

0    A
1    A
2    B
3    C
4    B
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin` and `idxmax` return the first matching index:

```
In [70]: df3 = DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [71]: df3

    A
e   2
d   1
c   1
b   3
a NaN

In [72]: df3['A'].idxmin()
'd'
```

Note: `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

8.5.3 Value counts (histogramming)

The `value_counts` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [73]: data = np.random.randint(0, 7, size=50)

In [74]: data

array([4, 6, 6, 1, 2, 1, 0, 5, 3, 2, 4, 3, 1, 3, 5, 3, 0, 0, 4, 4, 6, 1, 0,
       4, 3, 2, 1, 3, 1, 5, 6, 3, 1, 2, 4, 4, 3, 3, 2, 2, 2, 3, 2, 3, 0, 1,
       2, 4, 5, 5])

In [75]: s = Series(data)

In [76]: s.value_counts()

3    11
2     9
4     8
1     8
5     5
0     5
6     4
dtype: int64

In [77]: value_counts(data)

3    11
2     9
4     8
1     8
5     5
0     5
6     4
dtype: int64
```

8.5.4 Discretization and quantiling

Continuous values can be discretized using the `cut` (bins based on values) and `qcut` (bins based on sample quantiles) functions:

```
In [78]: arr = np.random.randn(20)

In [79]: factor = cut(arr, 4)

In [80]: factor

Categorical:
[(-0.837, -0.0162], (-1.658, -0.837], (-2.483, -1.658], (-1.658, -0.837], (-0.837, -0.0162], (-0.0162
Levels (4): Index(['(-2.483, -1.658]', '(-1.658, -0.837]',
                   '(-0.837, -0.0162]', '(-0.0162, 0.805]'], dtype=object)

In [81]: factor = cut(arr, [-5, -1, 0, 1, 5])

In [82]: factor

Categorical:
[(-1, 0], (-5, -1], (-5, -1], (-5, -1], (-1, 0], (0, 1], (-5, -1], (0, 1], (0, 1], (0, 1], (-1, 0],
Levels (4): Index(['(-5, -1]', '(-1, 0]', '(0, 1]', '(1, 5]'], dtype=object)
```

`qcut` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size

quartiles like so:

```
In [83]: arr = np.random.randn(30)

In [84]: factor = qcut(arr, [0, .25, .5, .75, 1])

In [85]: factor

Categorical:
[[-2.891, -0.868], (0.525, 3.19], (-0.868, -0.0118], (-0.0118, 0.525], (-0.0118, 0.525], (0.525, 3.19
Levels (4): Index(['[-2.891, -0.868]', '(-0.868, -0.0118]',
                   '(-0.0118, 0.525]', '(0.525, 3.19]'], dtype=object)

In [86]: value_counts(factor)

[-2.891, -0.868]     8
(0.525, 3.19]        8
(-0.868, -0.0118]    7
(-0.0118, 0.525]     7
dtype: int64
```

8.6 Function application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the `apply` method, which, like the descriptive statistics methods, take an optional `axis` argument:

```
In [87]: df.apply(np.mean)

one     -0.274551
three   -0.223384
two      0.162987
dtype: float64

In [88]: df.apply(np.mean, axis=1)

a   -0.394235
b    0.130882
c   -0.127557
d   -0.031682
dtype: float64

In [89]: df.apply(lambda x: x.max() - x.min())

one      0.810701
three    0.205973
two      0.724777
dtype: float64

In [90]: df.apply(np.cumsum)

        one     three       two
a -0.701368       NaN -0.087103
b -0.592035 -0.354359  0.550570
c -0.823652 -0.502746  0.547904
d       NaN -0.670153  0.651948

In [91]: df.apply(np.exp)
```

```
        one     three       two
a  0.495907       NaN  0.916583
b  1.115534  0.701623  1.892074
c  0.793250  0.862098  0.997337
d       NaN  0.845855  1.109649
```

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

In [93]: tsdf.apply(lambda x: x.index[x.dropna().argmax()])

A   2000-10-05 00:00:00
B   2002-05-26 00:00:00
C   2000-07-10 00:00:00
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the `apply` method. For instance, consider the following function you would like to apply:

```
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [94]: tsdf

                   A         B         C
2000-01-01 -0.748358  0.938378 -0.421370
2000-01-02  0.310699  0.247939  0.480243
2000-01-03 -0.135533 -0.754617  0.669998
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -1.421098 -1.527750 -0.391382
2000-01-09  0.881063  0.173443 -0.290646
2000-01-10  2.189553  2.017892 -1.140611

In [95]: tsdf.apply(Series.interpolate)

                   A         B         C
2000-01-01 -0.748358  0.938378 -0.421370
2000-01-02  0.310699  0.247939  0.480243
2000-01-03 -0.135533 -0.754617  0.669998
2000-01-04 -0.392646 -0.909243  0.457722
2000-01-05 -0.649759 -1.063870  0.245446
2000-01-06 -0.906872 -1.218497  0.033170
2000-01-07 -1.163985 -1.373123 -0.179106
```

```
2000-01-08 -1.421098 -1.527750 -0.391382
2000-01-09  0.881063  0.173443 -0.290646
2000-01-10  2.189553  2.017892 -1.140611
```

Finally, `apply` takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

See Also:

The section on *GroupBy* demonstrates related, flexible functionality for grouping by some criterion, applying, and combining the results into a Series, DataFrame, etc.

8.6.1 Applying elementwise Python functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods `applymap` on DataFrame and analogously `map` on Series accept any Python function taking a single value and returning a single value. For example:

```
In [96]: f = lambda x: len(str(x))

In [97]: df['one'].map(f)

a    15
b    14
c    15
d     3
Name: one, dtype: int64

In [98]: df.applymap(f)

   one  three  two
a   15      3   16
b   14     15   14
c   15     15   17
d    3     15   14
```

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

In [100]: t = Series({'six' : 6., 'seven' : 7.})

In [101]: s

a      six
b    seven
c      six
d    seven
e      six
dtype: object

In [102]: s.map(t)

a    6
```

```
b    7
c    6
d    7
e    6
dtype: float64
```

8.7 Reindexing and altering labels

`reindex` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To *reindex* means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, **fill** data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [103]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [104]: s

a    1.721293
b    0.355636
c    0.498722
d   -0.277859
e    0.713249
dtype: float64

In [105]: s.reindex(['e', 'b', 'f', 'd'])

e    0.713249
b    0.355636
f         NaN
d   -0.277859
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [106]: df

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

In [107]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])

      three       two       one
c -0.148387 -0.002666 -0.231617
f       NaN       NaN       NaN
b -0.354359  0.637674  0.109333
```

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

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [108]: rs = s.reindex(df.index)

In [109]: rs

a    1.721293
b    0.355636
c    0.498722
d   -0.277859
dtype: float64

In [110]: rs.index is df.index
True
```

This means that the reindexed Series's index is the same Python object as the DataFrame's index.

See Also:

Advanced indexing is an even more concise way of doing reindexing.

Note: When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinking a few explicit `reindex` calls here and there can have an impact.

8.7.1 Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like` method is available to make this simpler:

```
In [111]: df

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

In [112]: df2

        one       two
a -0.426817 -0.269738
b  0.383883  0.455039
c  0.042934 -0.185301

In [113]: df.reindex_like(df2)

        one       two
a -0.701368 -0.087103
b  0.109333  0.637674
c -0.231617 -0.002666
```

8.7.2 Reindexing with `reindex_axis`

8.7.3 Aligning objects with each other with `align`

The `align` method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [114]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [115]: s1 = s[:4]

In [116]: s2 = s[1:]

In [117]: s1.align(s2)

(a   -0.013026
b    2.249919
c    0.449017
d   -0.486899
e         NaN
dtype: float64,
 a         NaN
b    2.249919
c    0.449017
d   -0.486899
e   -1.666155
dtype: float64)

In [118]: s1.align(s2, join='inner')

(b    2.249919
c    0.449017
d   -0.486899
dtype: float64,
 b    2.249919
c    0.449017
d   -0.486899
dtype: float64)

In [119]: s1.align(s2, join='left')

(a   -0.013026
b    2.249919
c    0.449017
d   -0.486899
dtype: float64,
 a         NaN
b    2.249919
c    0.449017
```

```
d   -0.486899
dtype: float64)
```

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

```
In [120]: df.align(df2, join='inner')

(        one       two
a -0.701368 -0.087103
b  0.109333  0.637674
c -0.231617 -0.002666,
         one       two
a -0.426817 -0.269738
b  0.383883  0.455039
c  0.042934 -0.185301)
```

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

```
In [121]: df.align(df2, join='inner', axis=0)

(        one     three       two
a -0.701368       NaN -0.087103
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666,
         one       two
a -0.426817 -0.269738
b  0.383883  0.455039
c  0.042934 -0.185301)
```

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 [122]: df.align(df2.ix[0], axis=1)

(        one     three       two
a -0.701368       NaN -0.087103
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
d       NaN -0.167407  0.104044,
 one     -0.426817
three         NaN
two     -0.269738
Name: a, dtype: float64)
```

8.7.4 Filling while reindexing

`reindex` takes an optional parameter `method` which is a filling method chosen from the following table:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward

Other fill methods could be added, of course, but these are the two most commonly used for time series data. In a way they only make sense for time series or otherwise ordered data, but you may have an application on non-time series data where this sort of "interpolation" logic is the correct thing to do. More sophisticated interpolation of missing values would be an obvious extension.

We illustrate these fill methods on a simple TimeSeries:

```
In [123]: rng = date_range('1/3/2000', periods=8)

In [124]: ts = Series(randn(8), index=rng)

In [125]: ts2 = ts[[0, 3, 6]]

In [126]: ts

2000-01-03    1.093167
2000-01-04    0.214964
2000-01-05   -0.355204
2000-01-06    1.228301
2000-01-07   -0.449976
2000-01-08   -0.923040
2000-01-09    0.701979
2000-01-10   -0.629836
Freq: D, dtype: float64

In [127]: ts2

2000-01-03    1.093167
2000-01-06    1.228301
2000-01-09    0.701979
dtype: float64

In [128]: ts2.reindex(ts.index)

2000-01-03    1.093167
2000-01-04         NaN
2000-01-05         NaN
2000-01-06    1.228301
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    0.701979
2000-01-10         NaN
Freq: D, dtype: float64

In [129]: ts2.reindex(ts.index, method='ffill')

2000-01-03    1.093167
2000-01-04    1.093167
2000-01-05    1.093167
2000-01-06    1.228301
2000-01-07    1.228301
2000-01-08    1.228301
2000-01-09    0.701979
2000-01-10    0.701979
Freq: D, dtype: float64

In [130]: ts2.reindex(ts.index, method='bfill')

2000-01-03    1.093167
2000-01-04    1.228301
2000-01-05    1.228301
2000-01-06    1.228301
2000-01-07    0.701979
2000-01-08    0.701979
2000-01-09    0.701979
```

```
2000-01-10         NaN
Freq: D, dtype: float64
```

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

```
In [131]: ts2.reindex(ts.index).fillna(method='ffill')

2000-01-03    1.093167
2000-01-04    1.093167
2000-01-05    1.093167
2000-01-06    1.228301
2000-01-07    1.228301
2000-01-08    1.228301
2000-01-09    0.701979
2000-01-10    0.701979
Freq: D, dtype: float64
```

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.

8.7.5 Dropping labels from an axis

A method closely related to `reindex` is the `drop` function. It removes a set of labels from an axis:

```
In [132]: df

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

In [133]: df.drop(['a', 'd'], axis=0)

        one     three       two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666

In [134]: df.drop(['one'], axis=1)

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

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

```
In [135]: df.reindex(df.index - ['a', 'd'])

        one     three       two
b  0.109333 -0.354359  0.637674
c -0.231617 -0.148387 -0.002666
```

8.7.6 Renaming / mapping labels

The `rename` method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [136]: s

a   -0.013026
b    2.249919
c    0.449017
d   -0.486899
e   -1.666155
dtype: float64

In [137]: s.rename(str.upper)

A   -0.013026
B    2.249919
C    0.449017
D   -0.486899
E   -1.666155
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). But if you pass a dict or Series, it need only contain a subset of the labels as keys:

```
In [138]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
   .....:           index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
   .....:

             foo     three       bar
apple  -0.701368       NaN -0.087103
banana  0.109333 -0.354359  0.637674
c      -0.231617 -0.148387 -0.002666
durian       NaN -0.167407  0.104044
```

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

8.8 Iteration

Because Series is array-like, basic iteration produces the values. Other data structures follow the dict-like convention of iterating over the "keys" of the objects. In short:

- **Series**: values
- **DataFrame**: column labels
- **Panel**: item labels

Thus, for example:

```
In [139]: for col in df:
   .....:     print col
   .....:
one
three
two
```

8.8.1 iteritems

Consistent with the dict-like interface, **iteritems** iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs
- **Panel**: (item, DataFrame) pairs

For example:

```
In [140]: for item, frame in wp.iteritems():
   .....:     print item
   .....:     print frame
   .....:
Item1
                   A         B         C         D
2000-01-01 -1.118121  0.431279  0.554724 -1.333649
2000-01-02 -0.332174 -0.485882  1.725945  1.799276
2000-01-03 -0.968916 -0.779465 -2.000701 -1.866630
2000-01-04 -1.101268  1.957478  0.058889  0.758071
2000-01-05  0.076612 -0.548502 -0.160485 -0.377780
Item2
                   A         B         C         D
2000-01-01  0.249911 -0.341270 -0.272599 -0.277446
2000-01-02 -1.102896  0.100307 -1.602814  0.920139
2000-01-03 -0.643870  0.060336 -0.434942 -0.494305
2000-01-04  0.737973  0.451632  0.334124 -0.787062
2000-01-05  0.651396 -0.741919  1.193881 -2.395763
```

8.8.2 iterrows

New in v0.7 is the ability to iterate efficiently through rows of a DataFrame. It returns an iterator yielding each index value along with a Series containing the data in each row:

```
In [141]: for row_index, row in df2.iterrows():
   .....:     print '%s\n%s' % (row_index, row)
   .....:
a
one   -0.426817
two   -0.269738
Name: a, dtype: float64
b
one    0.383883
two    0.455039
Name: b, dtype: float64
c
one    0.042934
two   -0.185301
Name: c, dtype: float64
```

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

```
In [142]: df2 = DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [143]: print df2
   x  y
0  1  4
```

```
1  2  5
2  3  6

In [144]: print df2.T
   0  1  2
x  1  2  3
y  4  5  6

In [145]: df2_t = DataFrame(dict((idx,values) for idx, values in df2.iterrows()))

In [146]: print df2_t
   0  1  2
x  1  2  3
y  4  5  6
```

Note: `iterrows` does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [147]: df_iter = DataFrame([[1, 1.0]], columns=['x', 'y'])

In [148]: row = next(df_iter.iterrows())[1]

In [149]: print row['x'].dtype
float64

In [150]: print df_iter['x'].dtype
int64
```

8.8.3 itertuples

This method will return an iterator yielding a tuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values proper.

For instance,

```
In [151]: for r in df2.itertuples(): print r
(0, 1, 4)
(1, 2, 5)
(2, 3, 6)
```

8.9 Vectorized string methods

Series is equipped (as of pandas 0.8.1) with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's `str` attribute and generally have names matching the equivalent (scalar) build-in string methods:

```
In [152]: s = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [153]: s.str.lower()

0       a
1       b
```

```
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

```
In [154]: s.str.upper()
```

```
0       A
1       B
2       C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object
```

```
In [155]: s.str.len()
```

```
0     1
1     1
2     1
3     4
4     4
5   NaN
6     4
7     3
8     3
dtype: float64
```

Methods like `split` return a Series of lists:

```
In [156]: s2 = Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])
```

```
In [157]: s2.str.split('_')
```

```
0    [a, b, c]
1    [c, d, e]
2          NaN
3    [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using `get` or `[]` notation:

```
In [158]: s2.str.split('_').str.get(1)
```

```
0      b
1      d
2    NaN
3      g
dtype: object
```

```
In [159]: s2.str.split('_').str[1]
```

```
0      b
1      d
2    NaN
3      g
dtype: object
```

Methods like `replace` and `findall` take regular expressions, too:

```
In [160]: s3 = Series(['A', 'B', 'C', 'Aaba', 'Baca',
   .....:              '', np.nan, 'CABA', 'dog', 'cat'])
   .....:

In [161]: s3

0       A
1       B
2       C
3    Aaba
4    Baca
5
6     NaN
7    CABA
8     dog
9     cat
dtype: object

In [162]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)

0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6         NaN
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: object
```

Methods like `contains`, `startswith`, and `endswith` takes an extra `na` arguement so missing values can be considered True or False:

```
In [163]: s4 = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [164]: s4.str.contains('A', na=False)

0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
dtype: bool
```

Method	Description
cat	Concatenate strings
split	Split strings on delimiter
get	Index into each element (retrieve i-th element)
join	Join strings in each element of the Series with passed separator
contains	Return boolean array if each string contains pattern/regex
replace	Replace occurrences of pattern/regex with some other string
repeat	Duplicate values (`s.str.repeat(3)` equivalent to `x * 3`)
pad	Add whitespace to left, right, or both sides of strings
center	Equivalent to `pad(side='both')`
slice	Slice each string in the Series
slice_replace	Replace slice in each string with passed value
count	Count occurrences of pattern
startswith	Equivalent to `str.startswith(pat)` for each element
endswidth	Equivalent to `str.endswith(pat)` for each element
findall	Compute list of all occurrences of pattern/regex for each string
match	Call `re.match` on each element, returning matched groups as list
len	Compute string lengths
strip	Equivalent to `str.strip`
rstrip	Equivalent to `str.rstrip`
lstrip	Equivalent to `str.lstrip`
lower	Equivalent to `str.lower`
upper	Equivalent to `str.upper`

8.10 Sorting by index and value

There are two obvious kinds of sorting that you may be interested in: sorting by label and sorting by actual values. The primary method for sorting axis labels (indexes) across data structures is the `sort_index` method.

```
In [165]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....:                          columns=['three', 'two', 'one'])
   .....:

In [166]: unsorted_df.sort_index()

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

In [167]: unsorted_df.sort_index(ascending=False)

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

In [168]: unsorted_df.sort_index(axis=1)

        one     three       two
a -0.701368       NaN -0.087103
d       NaN -0.167407  0.104044
```

```
c -0.231617 -0.148387 -0.002666
b  0.109333 -0.354359  0.637674
```

`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 [169]: df.sort_index(by='two')

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

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

```
In [170]: df1 = DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})

In [171]: df1[['one', 'two', 'three']].sort_index(by=['one','two'])

   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

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 [172]: s[2] = np.nan

In [173]: s.order()

0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
2     NaN
5     NaN
dtype: object

In [174]: s.order(na_last=False)

2     NaN
5     NaN
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
dtype: object
```

Some other sorting notes / nuances:

- `Series.sort` sorts a Series by value in-place. This is to provide compatibility with NumPy methods which

expect the `ndarray.sort` behavior.

- `DataFrame.sort` takes a `column` argument instead of `by`. This method will likely be deprecated in a future release in favor of just using `sort_index`.

8.11 Copying

The `copy` method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column
- Assigning to the `index` or `columns` attributes
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing

To be clear, no pandas methods have the side effect of modifying your data; almost all methods return new objects, leaving the original object untouched. If data is modified, it is because you did so explicitly.

8.12 dtypes

The main types stored in pandas objects are `float`, `int`, `bool`, `datetime64[ns]`, `timedelta[ns]`, and `object`. In addition these dtypes have item sizes, e.g. `int64` and `int32`. A convenient `dtypes` attribute for DataFrames returns a Series with the data type of each column.

```
In [175]: dft = DataFrame(dict( A = np.random.rand(3),
   .....:                       B = 1,
   .....:                       C = 'foo',
   .....:                       D = Timestamp('20010102'),
   .....:                       E = Series([1.0]*3).astype('float32'),
   .....:                       F = False,
   .....:                       G = Series([1]*3,dtype='int8')))
   .....:

In [176]: dft

          A  B    C                   D  E      F  G
0  0.736120  1  foo 2001-01-02 00:00:00  1  False  1
1  0.364264  1  foo 2001-01-02 00:00:00  1  False  1
2  0.091972  1  foo 2001-01-02 00:00:00  1  False  1

In [177]: dft.dtypes

A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
F              bool
G              int8
dtype: object
```

On a `Series` use the `dtype` method.

```
In [178]: dft['A'].dtype
dtype('float64')
```

If a pandas object contains data multiple dtypes *IN A SINGLE COLUMN*, the dtype of the column will be chosen to accommodate all of the data types (`object` is the most general).

```
# these ints are coerced to floats
In [179]: Series([1, 2, 3, 4, 5, 6.])

0    1
1    2
2    3
3    4
4    5
5    6
dtype: float64

# string data forces an ``object`` dtype
In [180]: Series([1, 2, 3, 6., 'foo'])

0      1
1      2
2      3
3      6
4    foo
dtype: object
```

The method `get_dtype_counts` will return the number of columns of each type in a `DataFrame`:

```
In [181]: dft.get_dtype_counts()

bool              1
datetime64[ns]    1
float32           1
float64           1
int64             1
int8              1
object            1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames (starting in v0.11.0). If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [182]: df1 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')

In [183]: df1

          A
0 -0.693708
1  0.084626
2 -0.003949
3  0.268088
4  0.357356
5  0.052999
6 -0.632983
7  1.332674

In [184]: df1.dtypes
```

```
A    float32
dtype: object

In [185]: df2 = DataFrame(dict( A = Series(randn(8),dtype='float16'),
   .....:                       B = Series(randn(8)),
   .....:                       C = Series(np.array(randn(8),dtype='uint8')) ))
   .....:

In [186]: df2

          A         B    C
0  1.921875 -0.311588    0
1 -0.101746  0.550255    1
2  1.352539  0.718337    2
3  1.264648  1.252982  255
4 -1.261719 -0.453845    0
5 -1.037109  1.151367    1
6  1.552734  1.406869    0
7 -0.503418 -2.264574    0

In [187]: df2.dtypes

A    float16
B    float64
C      uint8
dtype: object
```

8.12.1 defaults

By default integer types are `int64` and float types are `float64`, *REGARDLESS* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [188]: DataFrame([1,2],columns=['a']).dtypes

a    int64
dtype: object

In [189]: DataFrame({'a' : [1,2] }).dtypes

a    int64
dtype: object

In [190]: DataFrame({'a' : 1 }, index=range(2)).dtypes

a    int64
dtype: object
```

Numpy, however will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [191]: frame = DataFrame(np.array([1,2]))
```

8.12.2 upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (say `int` to `float`)

```
In [192]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [193]: df3

          A         B    C
0  1.228167 -0.311588    0
1 -0.017120  0.550255    1
2  1.348590  0.718337    2
3  1.532737  1.252982  255
4 -0.904363 -0.453845    0
5 -0.984110  1.151367    1
6  0.919751  1.406869    0
7  0.829256 -2.264574    0

In [194]: df3.dtypes

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

The `values` attribute on a DataFrame return the *lower-common-denominator* of the dtypes, meaning the dtype that can accomodate **ALL** of the types in the resulting homogenous dtyped numpy array. This can force some *upcasting*.

```
In [195]: df3.values.dtype
dtype('float64')
```

8.12.3 astype

You can use the `astype` method to explicity convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **numpy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [196]: df3

          A         B    C
0  1.228167 -0.311588    0
1 -0.017120  0.550255    1
2  1.348590  0.718337    2
3  1.532737  1.252982  255
4 -0.904363 -0.453845    0
5 -0.984110  1.151367    1
6  0.919751  1.406869    0
7  0.829256 -2.264574    0

In [197]: df3.dtypes
```

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

# conversion of dtypes
In [198]: df3.astype('float32').dtypes

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

8.12.4 object conversion

`convert_objects` is a method to try to force conversion of types from the `object` dtype to other types. To force conversion of specific types that are *number like*, e.g. could be a string that represents a number, pass `convert_numeric=True`. This will force strings and numbers alike to be numbers if possible, otherwise they will be set to `np.nan`.

```
In [199]: df3['D'] = '1.'

In [200]: df3['E'] = '1'

In [201]: df3.convert_objects(convert_numeric=True).dtypes

A    float32
B    float64
C    float64
D    float64
E      int64
dtype: object

# same, but specific dtype conversion
In [202]: df3['D'] = df3['D'].astype('float16')

In [203]: df3['E'] = df3['E'].astype('int32')

In [204]: df3.dtypes

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

To force conversion to `datetime64[ns]`, pass `convert_dates='coerce'`. This will convert any datetimelike object to dates, forcing other values to `NaT`. This might be useful if you are reading in data which is mostly dates, but occasionally has non-dates intermixed and you want to represent as missing.

```
In [205]: s = Series([datetime(2001,1,1,0,0),
   .....:             'foo', 1.0, 1, Timestamp('20010104'),
   .....:             '20010105'],dtype='O')
   .....:
```

```
In [206]: s

0    2001-01-01 00:00:00
1                    foo
2                      1
3                      1
4    2001-01-04 00:00:00
5               20010105
dtype: object

In [207]: s.convert_objects(convert_dates='coerce')

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

In addition, `convert_objects` will attempt the *soft* conversion of any *object* dtypes, meaning that if all the objects in a Series are of the same type, the Series will have that dtype.

8.12.5 gotchas

Performing selection operations on `integer` type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where `nans` are not introduced (starting in 0.11.0) See also *integer na gotchas*

```
In [208]: dfi = df3.astype('int32')

In [209]: dfi['E'] = 1

In [210]: dfi

   A  B    C  D  E
0  1  0    0  1  1
1  0  0    1  1  1
2  1  0    2  1  1
3  1  1  255  1  1
4  0  0    0  1  1
5  0  1    1  1  1
6  0  1    0  1  1
7  0 -2    0  1  1

In [211]: dfi.dtypes

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

In [212]: casted = dfi[dfi>0]

In [213]: casted
```

```
    A   B    C  D  E
0   1 NaN  NaN  1  1
1 NaN NaN    1  1  1
2   1 NaN    2  1  1
3   1   1  255  1  1
4 NaN NaN  NaN  1  1
5 NaN   1    1  1  1
6 NaN   1  NaN  1  1
7 NaN NaN  NaN  1  1
```

```
In [214]: casted.dtypes
```

```
A    float64
B    float64
C    float64
D      int32
E      int64
dtype: object
```

While float dtypes are unchanged.

```
In [215]: dfa = df3.copy()
```

```
In [216]: dfa['A'] = dfa['A'].astype('float32')
```

```
In [217]: dfa.dtypes
```

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

```
In [218]: casted = dfa[df2>0]
```

```
In [219]: casted
```

```
          A         B    C   D   E
0  1.228167       NaN  NaN NaN NaN
1       NaN  0.550255    1 NaN NaN
2  1.348590  0.718337    2 NaN NaN
3  1.532737  1.252982  255 NaN NaN
4       NaN       NaN  NaN NaN NaN
5       NaN  1.151367    1 NaN NaN
6  0.919751  1.406869  NaN NaN NaN
7       NaN       NaN  NaN NaN NaN
```

```
In [220]: casted.dtypes
```

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

8.13 Working with package options

New in version 0.10.1. Pandas has an options system that let's you customize some aspects of it's behaviour, display-related options being those the user is must likely to adjust.

Options have a full "dotted-style", case-insensitive name (e.g. `display.max_rows`), You can get/set options directly as attributes of the top-level `options` attribute:

```
In [221]: import pandas as pd

In [222]: pd.options.display.max_rows
60

In [223]: pd.options.display.max_rows = 999

In [224]: pd.options.display.max_rows
999
```

There is also an API composed of 4 relevant functions, available directly from the `pandas` namespace, and they are:

- `get_option` / `set_option` - get/set the value of a single option.
- `reset_option` - reset one or more options to their default value.
- `describe_option` - print the descriptions of one or more options.

Note: developers can check out pandas/core/config.py for more info.

All of the functions above accept a regexp pattern (`re.search` style) as an argument, and so passing in a substring will work - as long as it is unambiguous :

```
In [225]: get_option("display.max_rows")
999

In [226]: set_option("display.max_rows",101)

In [227]: get_option("display.max_rows")
101

In [228]: set_option("max_r",102)

In [229]: get_option("display.max_rows")
102
```

The following will **not work** because it matches multiple option names, e.g.‘‘display.max_colwidth‘‘, `display.max_rows`, `display.max_columns`:

```
In [230]: try:
   .....:     get_option("display.max_")
   .....: except KeyError as e:
   .....:     print(e)
   .....:
'Pattern matched multiple keys'
```

Note: Using this form of convenient shorthand may make your code break if new options with similar names are added in future versions.

You can get a list of available options and their descriptions with `describe_option`. When called with no argument `describe_option` will print out the descriptions for all available options.

```
In [231]: describe_option()
display.chop_threshold: [default: None] [currently: None]
: float or None
        if set to a float value, all float values smaller then the given threshold
        will be displayed as exactly 0 by repr and friends.
display.colheader_justify: [default: right] [currently: right]
: 'left'/'right'
        Controls the justification of column headers. used by DataFrameFormatter.
display.column_space: [default: 12] [currently: 12]No description available.
display.date_dayfirst: [default: False] [currently: False]
: boolean
        When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst: [default: False] [currently: False]
: boolean
        When True, prints and parses dates with the year first, eg 2005/01/20
display.encoding: [default: UTF-8] [currently: UTF-8]
: str/unicode
        Defaults to the detected encoding of the console.
        Specifies the encoding to be used for strings returned by to_string,
        these are generally strings meant to be displayed on the console.
display.expand_frame_repr: [default: True] [currently: True]
: boolean
        Whether to print out the full DataFrame repr for wide DataFrames
        across multiple lines, `max_columns` is still respected, but the output will
        wrap-around across multiple "pages" if it's width exceeds `display.width`.
display.float_format: [default: None] [currently: None]
: callable
        The callable should accept a floating point number and return
        a string with the desired format of the number. This is used
        in some places like SeriesFormatter.
        See core.format.EngFormatter for an example.
display.height: [default: 60] [currently: 60]
: int
        Deprecated.
        (Deprecated, use `display.height` instead.)
display.line_width: [default: 80] [currently: 80]
: int
        Deprecated.
        (Deprecated, use `display.width` instead.)
display.max_columns: [default: 20] [currently: 20]
: int
        max_rows and max_columns are used in __repr__() methods to decide if
        to_string() or info() is used to render an object to a string.  In case
        python/IPython is running in a terminal this can be set to 0 and pandas
        will correctly auto-detect the width the terminal and swap to a smaller
        format in case all columns would not fit vertically. The IPython notebook,
        IPython qtconsole, or IDLE do not run in a terminal and hence it is not
        possible to do correct auto-detection.
        'None' value means unlimited.
display.max_colwidth: [default: 50] [currently: 50]
: int
        The maximum width in characters of a column in the repr of
        a pandas data structure. When the column overflows, a "..."
        placeholder is embedded in the output.
display.max_info_columns: [default: 100] [currently: 100]
: int
        max_info_columns is used in DataFrame.info method to decide if
        per column information will be printed.
```

```
display.max_info_rows: [default: 1690785] [currently: 1690785]
: int or None
        max_info_rows is the maximum number of rows for which a frame will
        perform a null check on its columns when repr'ing To a console.
        The default is 1,000,000 rows. So, if a DataFrame has more
        1,000,000 rows there will be no null check performed on the
        columns and thus the representation will take much less time to
        display in an interactive session. A value of None means always
        perform a null check when repr'ing.
display.max_rows: [default: 60] [currently: 102]
: int
        This sets the maximum number of rows pandas should output when printing
        out various output. For example, this value determines whether the repr()
        for a dataframe prints out fully or just a summary repr.
        'None' value means unlimited.
display.max_seq_items: [default: None] [currently: None]
: int or None

        when pretty-printing a long sequence, no more then `max_seq_items`
        will be printed. If items are ommitted, they will be denoted by the addition
        of "..." to the resulting string.

        If set to None, the number of items to be printed is unlimited.
display.mpl_style: [default: None] [currently: default]
: bool

        Setting this to 'default' will modify the rcParams used by matplotlib
        to give plots a more pleasing visual style by default.
        Setting this to None/False restores the values to their initial value.
display.multi_sparse: [default: True] [currently: True]
: boolean
        "sparsify" MultiIndex display (don't display repeated
        elements in outer levels within groups)
display.notebook_repr_html: [default: True] [currently: True]
: boolean
        When True, IPython notebook will use html representation for
        pandas objects (if it is available).
display.pprint_nest_depth: [default: 3] [currently: 3]
: int
        Controls the number of nested levels to process when pretty-printing
display.precision: [default: 7] [currently: 7]
: int
        Floating point output precision (number of significant digits). This is
        only a suggestion
display.width: [default: 80] [currently: 80]
: int
        Width of the display in characters. In case python/IPython is running in
        a terminal this can be set to None and pandas will correctly auto-detect the
        width.
        Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a
        terminal and hence it is not possible to correctly detect the width.
mode.sim_interactive: [default: False] [currently: False]
: boolean
        Whether to simulate interactive mode for purposes of testing
mode.use_inf_as_null: [default: False] [currently: False]
: boolean
        True means treat None, NaN, INF, -INF as null (old way),
        False means None and NaN are null, but INF, -INF are not null
```

```
        (new way).
```

or you can get the description for just the options that match the regexp you pass in:

```
In [232]: describe_option("date")
display.date_dayfirst: [default: False] [currently: False]
: boolean
        When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst: [default: False] [currently: False]
: boolean
        When True, prints and parses dates with the year first, eg 2005/01/20
```

All options also have a default value, and you can use the `reset_option` to do just that:

```
In [233]: get_option("display.max_rows")
60

In [234]: set_option("display.max_rows",999)

In [235]: get_option("display.max_rows")
999

In [236]: reset_option("display.max_rows")

In [237]: get_option("display.max_rows")
60
```

It's also possible to reset multiple options at once:

```
In [238]: reset_option("^display\.")
```

8.14 Console Output Formatting

Note: `set_printoptions`/ `reset_printoptions` are now deprecated (but functioning), and both, as well as `set_eng_float_format`, use the options API behind the scenes. The corresponding options now live under "print.XYZ", and you can set them directly with `get/set_option`.

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 [239]: set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [240]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [241]: s/1.e3

a      1.067m
b    -64.337u
c      1.484m
d   -524.332u
e   -688.585u
dtype: float64

In [242]: s/1.e6
```

```
a      1.067u
b    -64.337n
c      1.484u
d   -524.332n
e   -688.585n
dtype: float64
```

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

NINE

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.

Note: The Python and NumPy indexing operators `[]` and attribute operator `.` provide quick and easy access to pandas data structures across a wide range of use cases. This makes interactive work intuitive, as there's little new to learn if you already know how to deal with Python dictionaries and NumPy arrays. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommended that you take advantage of the optimized pandas data access methods exposed in this chapter.

In addition, whether a copy or a reference is returned for a selection operation, may depend on the context. See *Returning a View versus Copy*

See the *cookbook* for some advanced strategies

9.1 Choice

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

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

  See more at *Selection by Label*

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

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

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

  See more at *Advanced Indexing*, *Advanced Hierarchical* and *Fallback Indexing*

Getting values from an object with multi-axes selection uses the following notation (using `.loc` as an example, but applies to `.iloc` and `.ix` as well). Any of the axes accessors may be the null slice `:`. Axes left out of the specification are assumed to be `:`. (e.g. `p.loc['a']` is equiv to `p.loc['a',:,:]`)

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`
Panel	`p.loc[item_indexer,major_indexer,minor_indexer]`

9.1.1 Deprecations

Beginning with version 0.11.0, it's recommended that you transition away from the following methods as they *may* be deprecated in future versions.

- `irow`
- `icol`
- `iget_value`

See the section *Selection by Position* for substitutes.

9.2 Basics

As mentioned when introducing the data structures in the *last section*, the primary function of indexing with `[]` (a.k.a. `__getitem__` for those familiar with implementing class behavior in Python) is selecting out lower-dimensional slices. Thus,

Object Type	Selection	Return Value Type
Series	`series[label]`	scalar value
DataFrame	`frame[colname]`	`Series` corresponding to colname
Panel	`panel[itemname]`	`DataFrame` corresponing to the itemname

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

```
In [1]: dates = date_range('1/1/2000', periods=8)

In [2]: df = DataFrame(randn(8, 4), index=dates, columns=['A', 'B', 'C', 'D'])
```

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

```
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
```

```
In [4]: panel = Panel({'one' : df, 'two' : df - df.mean()})
```

```
In [5]: panel
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D
```

Note: None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using `[]`:

```
In [6]: s = df['A']
```

```
In [7]: s[dates[5]]
-0.67368970808837059
```

```
In [8]: panel['two']
```

```
                   A         B         C         D
2000-01-01  0.409571  0.113086 -0.610826 -0.936507
2000-01-02  1.152571  0.222735  1.017442 -0.845111
2000-01-03 -0.921390 -1.708620  0.403304  1.270929
2000-01-04  0.662014 -0.310822 -0.141342  0.470985
2000-01-05 -0.484513  0.962970  1.174465 -0.888276
2000-01-06 -0.733231  0.509598 -0.580194  0.724113
2000-01-07  0.345164  0.972995 -0.816769 -0.840143
2000-01-08 -0.430188 -0.761943 -0.446079  1.044010
```

You can pass a list of columns to `[]` to select columns in that order. If a column is not contained in the DataFrame, an exception will be raised. Multiple columns can also be set in this manner:

```
In [9]: df
```

```
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
```

```
In [10]: df[['B', 'A']] = df[['A', 'B']]

In [11]: df

                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04 -0.706771  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885
```

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

9.2.1 Attribute Access

You may access a column on a `DataFrame`, and a item on a `Panel` directly as an attribute:

```
In [12]: df.A

2000-01-01   -0.282863
2000-01-02   -0.173215
2000-01-03   -2.104569
2000-01-04   -0.706771
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
2000-01-08   -1.157892
Freq: D, Name: A, dtype: float64

In [13]: panel.one

                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
```

If you are using the IPython environment, you may also use tab-completion to see these accessable attributes.

9.2.2 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the *Selection by Position* section detailing the `.iloc` method. For now, we explain the semantics of slicing using the `[]` operator.

With Series, the syntax works exactly as with an ndarray, returning a slice of the values and the corresponding labels:

```
In [14]: s[:5]

2000-01-01   -0.282863
```

```
2000-01-02   -0.173215
2000-01-03   -2.104569
2000-01-04   -0.706771
2000-01-05    0.567020
Freq: D, Name: A, dtype: float64
```

```
In [15]: s[::2]
```

```
2000-01-01   -0.282863
2000-01-03   -2.104569
2000-01-05    0.567020
2000-01-07    0.577046
Freq: 2D, Name: A, dtype: float64
```

```
In [16]: s[::-1]
```

```
2000-01-08   -1.157892
2000-01-07    0.577046
2000-01-06    0.113648
2000-01-05    0.567020
2000-01-04   -0.706771
2000-01-03   -2.104569
2000-01-02   -0.173215
2000-01-01   -0.282863
Freq: -1D, Name: A, dtype: float64
```

Note that setting works as well:

```
In [17]: s2 = s.copy()
```

```
In [18]: s2[:5] = 0
```

```
In [19]: s2
```

```
2000-01-01    0.000000
2000-01-02    0.000000
2000-01-03    0.000000
2000-01-04    0.000000
2000-01-05    0.000000
2000-01-06    0.113648
2000-01-07    0.577046
2000-01-08   -1.157892
Freq: D, Name: A, dtype: float64
```

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

```
In [20]: df[:3]
```

```
                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
```

```
In [21]: df[::-1]
```

```
                   A         B         C         D
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
```

```
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-04 -0.706771  0.721555 -1.039575  0.271860
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
```

9.2.3 Selection By Label

Pandas provides a suite of methods in order to have **purely label based indexing**. This is a strict inclusion based protocol. **ALL** of the labels for which you ask, must be in the index or a `KeyError` will be raised! When slicing, the start bound is *included*, **AND** the stop bound is *included*. Integers are valid labels, but they refer to the label **and not the position**.

The `.loc` attribute is the primary access method. The following are valid inputs:

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

```
In [22]: s1 = Series(np.random.randn(6),index=list('abcdef'))

In [23]: s1

a    1.075770
b   -0.109050
c    1.643563
d   -1.469388
e    0.357021
f   -0.674600
dtype: float64

In [24]: s1.loc['c':]

c    1.643563
d   -1.469388
e    0.357021
f   -0.674600
dtype: float64

In [25]: s1.loc['b']
-0.10904997528022223
```

Note that setting works as well:

```
In [26]: s1.loc['c':] = 0

In [27]: s1

a    1.07577
b   -0.10905
c    0.00000
d    0.00000
```

```
e    0.00000
f    0.00000
dtype: float64
```

With a DataFrame

```
In [28]: df1 = DataFrame(np.random.randn(6,4),
   ....:                 index=list('abcdef'),
   ....:                 columns=list('ABCD'))
   ....:

In [29]: df1

          A         B         C         D
a -1.776904 -0.968914 -1.294524  0.413738
b  0.276662 -0.472035 -0.013960 -0.362543
c -0.006154 -0.923061  0.895717  0.805244
d -1.206412  2.565646  1.431256  1.340309
e -1.170299 -0.226169  0.410835  0.813850
f  0.132003 -0.827317 -0.076467 -1.187678

In [30]: df1.loc[['a','b','d'],:]

          A         B         C         D
a -1.776904 -0.968914 -1.294524  0.413738
b  0.276662 -0.472035 -0.013960 -0.362543
d -1.206412  2.565646  1.431256  1.340309
```

Accessing via label slices

```
In [31]: df1.loc['d':,'A':'C']

          A         B         C
d -1.206412  2.565646  1.431256
e -1.170299 -0.226169  0.410835
f  0.132003 -0.827317 -0.076467
```

For getting a cross section using a label (equiv to `df.xs('a')`)

```
In [32]: df1.loc['a']

A   -1.776904
B   -0.968914
C   -1.294524
D    0.413738
Name: a, dtype: float64
```

For getting values with a boolean array

```
In [33]: df1.loc['a']>0

A    False
B    False
C    False
D     True
Name: a, dtype: bool

In [34]: df1.loc[:,df1.loc['a']>0]

          D
```

```
a  0.413738
b -0.362543
c  0.805244
d  1.340309
e  0.813850
f -1.187678
```

For getting a value explicity (equiv to deprecated `df.get_value('a','A')`)

```
# this is also equivalent to ``df1.at['a','A']``
In [35]: df1.loc['a','A']
-1.7769037169718671
```

9.2.4 Selection By Position

Pandas provides a suite of methods in order to get **purely integer based indexing**. The semantics follow closely python and numpy slicing. These are `0-based` indexing. When slicing, the start bounds is *included*, while the upper bound is *excluded*. Trying to use a non-integer, even a **valid** label will raise a `IndexError`.

The `.iloc` attribute is the primary access method. The following are valid inputs:

- An integer e.g. `5`
- A list or array of integers `[4, 3, 0]`
- A slice object with ints `1:7`

```
In [36]: s1 = Series(np.random.randn(5),index=range(0,10,2))

In [37]: s1

0    1.130127
2   -1.436737
4   -1.413681
6    1.607920
8    1.024180
dtype: float64

In [38]: s1.iloc[:3]

0    1.130127
2   -1.436737
4   -1.413681
dtype: float64

In [39]: s1.iloc[3]
1.6079204745847746
```

Note that setting works as well:

```
In [40]: s1.iloc[:3] = 0

In [41]: s1

0    0.00000
2    0.00000
4    0.00000
6    1.60792
```

```
8    1.02418
dtype: float64
```

With a DataFrame

```
In [42]: df1 = DataFrame(np.random.randn(6,4),
   ....:                 index=range(0,12,2),
   ....:                 columns=range(0,8,2))
   ....:

In [43]: df1

           0         2         4         6
0   0.569605  0.875906 -2.211372  0.974466
2  -2.006747 -0.410001 -0.078638  0.545952
4  -1.219217 -1.226825  0.769804 -1.281247
6  -0.727707 -0.121306 -0.097883  0.695775
8   0.341734  0.959726 -1.110336 -0.619976
10  0.149748 -0.732339  0.687738  0.176444
```

Select via integer slicing

```
In [44]: df1.iloc[:3]

          0         2         4         6
0  0.569605  0.875906 -2.211372  0.974466
2 -2.006747 -0.410001 -0.078638  0.545952
4 -1.219217 -1.226825  0.769804 -1.281247

In [45]: df1.iloc[1:5,2:4]

          4         6
2 -0.078638  0.545952
4  0.769804 -1.281247
6 -0.097883  0.695775
8 -1.110336 -0.619976
```

Select via integer list

```
In [46]: df1.iloc[[1,3,5],[1,3]]

           2         6
2  -0.410001  0.545952
6  -0.121306  0.695775
10 -0.732339  0.176444
```

For slicing rows explicitly (equiv to deprecated `df.irow(slice(1,3))`).

```
In [47]: df1.iloc[1:3,:]

          0         2         4         6
2 -2.006747 -0.410001 -0.078638  0.545952
4 -1.219217 -1.226825  0.769804 -1.281247
```

For slicing columns explicitly (equiv to deprecated `df.icol(slice(1,3))`).

```
In [48]: df1.iloc[:,1:3]

           2         4
0   0.875906 -2.211372
```

```
2  -0.410001 -0.078638
4  -1.226825  0.769804
6  -0.121306 -0.097883
8   0.959726 -1.110336
10 -0.732339  0.687738
```

For getting a scalar via integer position (equiv to deprecated `df.get_value(1,1)`)

```
# this is also equivalent to ``df1.iat[1,1]``
In [49]: df1.iloc[1,1]
-0.41000056806065832
```

For getting a cross section using an integer position (equiv to `df.xs(1)`)

```
In [50]: df1.iloc[1]

0   -2.006747
2   -0.410001
4   -0.078638
6    0.545952
Name: 2, dtype: float64
```

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

```
# these are allowed in python/numpy.
In [51]: x = list('abcdef')

In [52]: x[4:10]
['e', 'f']

In [53]: x[8:10]
[]
```

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

```
>>> df.iloc[:,3:6]
IndexError: out-of-bounds on slice (end)
```

9.2.5 Fast scalar value getting and setting

Since indexing with `[]` must handle a lot of cases (single-label access, slicing, boolean indexing, etc.), it has a bit of overhead in order to figure out what you're asking for. If you only want to access a scalar value, the fastest way is to use the `at` and `iat` methods, which are implemented on all of the data structures.

Similary to `loc`, `at` provides **label** based scalar lookups, while, `iat` provides **integer** based lookups analagously to `iloc`

```
In [54]: s.iat[5]
0.1136484096888855

In [55]: df.at[dates[5], 'A']
0.1136484096888855

In [56]: df.iat[3, 0]
-0.70677113363008448
```

You can also set using these same indexers. These have the additional capability of enlarging an object. This method *always* returns a reference to the object it modified, which in the case of enlargement, will be a **new object**:

```
In [57]: df.at[dates[5], 'E'] = 7

In [58]: df.iat[3, 0] = 7
```

9.2.6 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: `|` for `or`, `&` for `and`, and `~` for `not`. These **must** be grouped by using parentheses.

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

```
In [59]: s[s > 0]

2000-01-04    7.000000
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
Freq: D, Name: A, dtype: float64

In [60]: s[(s < 0) & (s > -0.5)]

2000-01-01   -0.282863
2000-01-02   -0.173215
Freq: D, Name: A, dtype: float64

In [61]: s[(s < -1) | (s > 1 )]

2000-01-03   -2.104569
2000-01-04    7.000000
2000-01-08   -1.157892
Name: A, dtype: float64

In [62]: s[~(s < 0)]

2000-01-04    7.000000
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
Freq: D, Name: A, dtype: float64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame's index (for example, something derived from one of the columns of the DataFrame):

```
In [63]: df[df['A'] > 0]

                   A         B         C         D
2000-01-04  7.000000  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
```

Consider the `isin` method of Series, which returns a boolean vector that is true wherever the Series elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

```
In [64]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   ....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   ....:                  'c' : randn(7)})
   ....:
```

```
In [65]: df2[df2['a'].isin(['one', 'two'])]

     a  b         c
0  one  x  0.403310
1  one  y -0.154951
2  two  y  0.301624
4  two  y -1.369849
5  one  x -0.954208
```

List comprehensions and `map` method of Series can also be used to produce more complex criteria:

```
# only want 'two' or 'three'
In [66]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [67]: df2[criterion]

       a  b         c
2    two  y  0.301624
3  three  x -2.179861
4    two  y -1.369849

# equivalent but slower
In [68]: df2[[x.startswith('t') for x in df2['a']]]

       a  b         c
2    two  y  0.301624
3  three  x -2.179861
4    two  y -1.369849

# Multiple criteria
In [69]: df2[criterion & (df2['b'] == 'x')]

       a  b         c
3  three  x -2.179861
```

Note, with the choice methods *Selection by Label*, *Selection by Position*, and *Advanced Indexing* you may select along more than one axis using boolean vectors combined with other indexing expressions.

```
In [70]: df2.loc[criterion & (df2['b'] == 'x'),'b':'c']

   b         c
3  x -2.179861
```

9.2.7 Where and Masking

Selecting values from a Series with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows

```
In [71]: s[s > 0]

2000-01-04    7.000000
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
Freq: D, Name: A, dtype: float64
```

To return a Series of the same shape as the original

```
In [72]: s.where(s > 0)

2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04    7.000000
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
2000-01-08         NaN
Freq: D, Name: A, dtype: float64
```

Selecting values from a DataFrame with a boolean critierion now also preserves input data shape. `where` is used under the hood as the implementation. Equivalent is `df.where(df < 0)`

```
In [73]: df[df < 0]

                   A         B         C         D
2000-01-01 -0.282863       NaN -1.509059 -1.135632
2000-01-02 -0.173215       NaN       NaN -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929       NaN
2000-01-04       NaN       NaN -1.039575       NaN
2000-01-05       NaN -0.424972       NaN -1.087401
2000-01-06       NaN -0.673690 -1.478427       NaN
2000-01-07       NaN       NaN -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312       NaN
```

In addition, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```
In [74]: df.where(df < 0, -df)

                   A         B         C         D
2000-01-01 -0.282863 -0.469112 -1.509059 -1.135632
2000-01-02 -0.173215 -1.212112 -0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929 -1.071804
2000-01-04 -7.000000 -0.721555 -1.039575 -0.271860
2000-01-05 -0.567020 -0.424972 -0.276232 -1.087401
2000-01-06 -0.113648 -0.673690 -1.478427 -0.524988
2000-01-07 -0.577046 -0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312 -0.844885
```

You may wish to set values based on some boolean criteria. This can be done intuitively like so:

```
In [75]: s2 = s.copy()

In [76]: s2[s2 < 0] = 0

In [77]: s2

2000-01-01    0.000000
2000-01-02    0.000000
2000-01-03    0.000000
2000-01-04    7.000000
2000-01-05    0.567020
2000-01-06    0.113648
2000-01-07    0.577046
2000-01-08    0.000000
```

```
Freq: D, Name: A, dtype: float64

In [78]: df2 = df.copy()

In [79]: df2[df2 < 0] = 0

In [80]: df2
```

	A	B	C	D
2000-01-01	0.000000	0.469112	0.000000	0.000000
2000-01-02	0.000000	1.212112	0.119209	0.000000
2000-01-03	0.000000	0.000000	0.000000	1.071804
2000-01-04	7.000000	0.721555	0.000000	0.271860
2000-01-05	0.567020	0.000000	0.276232	0.000000
2000-01-06	0.113648	0.000000	0.000000	0.524988
2000-01-07	0.577046	0.404705	0.000000	0.000000
2000-01-08	0.000000	0.000000	0.000000	0.844885

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

```
In [81]: df2 = df.copy()

In [82]: df2[ df2[1:4] > 0 ] = 3

In [83]: df2
```

	A	B	C	D
2000-01-01	-0.282863	0.469112	-1.509059	-1.135632
2000-01-02	-0.173215	3.000000	3.000000	-1.044236
2000-01-03	-2.104569	-0.861849	-0.494929	3.000000
2000-01-04	3.000000	3.000000	-1.039575	3.000000
2000-01-05	0.567020	-0.424972	0.276232	-1.087401
2000-01-06	0.113648	-0.673690	-1.478427	0.524988
2000-01-07	0.577046	0.404705	-1.715002	-1.039268
2000-01-08	-1.157892	-0.370647	-1.344312	0.844885

By default, `where` returns a modified copy of the data. There is an optional parameter `inplace` so that the original data can be modified without creating a copy:

```
In [84]: df_orig = df.copy()

In [85]: df_orig.where(df > 0, -df, inplace=True);
In [85]: df_orig
```

	A	B	C	D
2000-01-01	0.282863	0.469112	1.509059	1.135632
2000-01-02	0.173215	1.212112	0.119209	1.044236
2000-01-03	2.104569	0.861849	0.494929	1.071804
2000-01-04	7.000000	0.721555	1.039575	0.271860
2000-01-05	0.567020	0.424972	0.276232	1.087401
2000-01-06	0.113648	0.673690	1.478427	0.524988
2000-01-07	0.577046	0.404705	1.715002	1.039268
2000-01-08	1.157892	0.370647	1.344312	0.844885

`mask` is the inverse boolean operation of `where`.

```
In [86]: s.mask(s >= 0)
```

```
2000-01-01   -0.282863
2000-01-02   -0.173215
2000-01-03   -2.104569
2000-01-04         NaN
2000-01-05         NaN
2000-01-06         NaN
2000-01-07         NaN
2000-01-08   -1.157892
Freq: D, Name: A, dtype: float64

In [87]: df.mask(df >= 0)

                   A         B         C         D
2000-01-01 -0.282863       NaN -1.509059 -1.135632
2000-01-02 -0.173215       NaN       NaN -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929       NaN
2000-01-04       NaN       NaN -1.039575       NaN
2000-01-05       NaN -0.424972       NaN -1.087401
2000-01-06       NaN -0.673690 -1.478427       NaN
2000-01-07       NaN       NaN -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312       NaN
```

9.2.8 Take Methods

Similar to numpy ndarrays, pandas Index, Series, and DataFrame also provides the `take` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [88]: index = Index(randint(0, 1000, 10))

In [89]: index
Int64Index([350, 634, 637, 430, 270, 333, 264, 738, 801, 829], dtype=int64)

In [90]: positions = [0, 9, 3]

In [91]: index[positions]
Int64Index([350, 829, 430], dtype=int64)

In [92]: index.take(positions)
Int64Index([350, 829, 430], dtype=int64)

In [93]: ser = Series(randn(10))

In [94]: ser.ix[positions]

0    0.007207
9   -1.623033
3    2.395985
dtype: float64

In [95]: ser.take(positions)

0    0.007207
9   -1.623033
3    2.395985
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [96]: frm = DataFrame(randn(5, 3))

In [97]: frm.take([1, 4, 3])

          0         1         2
1 -0.087302 -1.575170  1.771208
4  1.074803  0.173520  0.211027
3  1.586976  0.019234  0.264294

In [98]: frm.take([0, 2], axis=1)

          0         2
0  0.029399  0.282696
1 -0.087302  1.771208
2  0.816482 -0.612665
3  1.586976  0.264294
4  1.074803  0.211027
```

It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```
In [99]: arr = randn(10)

In [100]: arr.take([False, False, True, True])
array([ 1.3571,  1.3571,  1.4188,  1.4188])

In [101]: arr[[0, 1]]
array([ 1.3571,  1.4188])

In [102]: ser = Series(randn(10))

In [103]: ser.take([False, False, True, True])

0   -0.773723
0   -0.773723
1   -1.170653
1   -1.170653
dtype: float64

In [104]: ser.ix[[0, 1]]

0   -0.773723
1   -1.170653
dtype: float64
```

Finally, as a small note on performance, because the `take` method handles a narrower range of inputs, it can offer performance that is a good deal faster than fancy indexing.

9.2.9 Duplicate Data

If you want to identify and remove duplicate rows in a DataFrame, there are two methods that will help: `duplicated` and `drop_duplicates`. Each takes as an argument the columns to use to identify duplicated rows.

- `duplicated` returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.
- `drop_duplicates` removes duplicate rows.

By default, the first observed row of a duplicate set is considered unique, but each method has a `take_last` parameter that indicates the last observed row should be taken instead.

```
In [105]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   .....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                  'c' : np.random.randn(7)})
   .....:

In [106]: df2.duplicated(['a','b'])

0    False
1    False
2    False
3    False
4     True
5     True
6    False
dtype: bool

In [107]: df2.drop_duplicates(['a','b'])

       a  b         c
0    one  x  1.024098
1    one  y -0.106062
2    two  y  1.824375
3  three  x  0.595974
6    six  x -1.237881

In [108]: df2.drop_duplicates(['a','b'], take_last=True)

       a  b         c
1    one  y -0.106062
3  three  x  0.595974
4    two  y  1.167115
5    one  x  0.601544
6    six  x -1.237881
```

9.2.10 Dictionary-like `get` method

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

```
In [109]: s = Series([1,2,3], index=['a','b','c'])

In [110]: s.get('a')               # equivalent to s['a']
1

In [111]: s.get('x', default=-1)
-1
```

9.3 Advanced Indexing with `.ix`

Note: The recent addition of `.loc` and `.iloc` have enabled users to be quite explicit about indexing choices. `.ix` allows a great flexibility to specify indexing locations by *label* and/or *integer position*. Pandas will attempt to use any

passed *integer* as *label* locations first (like what `.loc` would do, then to fall back on *positional* indexing, like what `.iloc` would do). See *Fallback Indexing* for an example.

The syntax of using `.ix` is identical to `.loc`, in *Selection by Label*, and `.iloc` in *Selection by Position*.

The `.ix` attribute takes the following inputs:

- An integer or single label, e.g. `5` or `'a'`
- A list or array of labels `['a', 'b', 'c']` or integers `[4, 3, 0]`
- A slice object with ints `1:7` or labels `'a':'f'`
- A boolean array

We'll illustrate all of these methods. First, note that this provides a concise way of reindexing on multiple axes at once:

```
In [112]: subindex = dates[[3,4,5]]

In [113]: df.reindex(index=subindex, columns=['C', 'B'])

                   C         B
2000-01-04 -1.039575  0.721555
2000-01-05  0.276232 -0.424972
2000-01-06 -1.478427 -0.673690

In [114]: df.ix[subindex, ['C', 'B']]

                   C         B
2000-01-04 -1.039575  0.721555
2000-01-05  0.276232 -0.424972
2000-01-06 -1.478427 -0.673690
```

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

```
In [115]: df2 = df.copy()

In [116]: df2.ix[subindex, ['C', 'B']] = 0

In [117]: df2

                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04  7.000000  0.000000  0.000000  0.271860
2000-01-05  0.567020  0.000000  0.000000 -1.087401
2000-01-06  0.113648  0.000000  0.000000  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885
```

Indexing with an array of integers can also be done:

```
In [118]: df.ix[[4,3,1]]

                   A         B         C         D
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-04  7.000000  0.721555 -1.039575  0.271860
2000-01-02 -0.173215  1.212112  0.119209 -1.044236

In [119]: df.ix[dates[[4,3,1]]]
```

```
                   A         B         C         D
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-04  7.000000  0.721555 -1.039575  0.271860
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
```

Slicing has standard Python semantics for integer slices:

```
In [120]: df.ix[1:7, :2]
```

```
                   A         B
2000-01-02 -0.173215  1.212112
2000-01-03 -2.104569 -0.861849
2000-01-04  7.000000  0.721555
2000-01-05  0.567020 -0.424972
2000-01-06  0.113648 -0.673690
2000-01-07  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 [121]: date1, date2 = dates[[2, 4]]

In [122]: print date1, date2
2000-01-03 00:00:00 2000-01-05 00:00:00

In [123]: df.ix[date1:date2]
```

```
                   A         B         C         D
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04  7.000000  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401

In [124]: df['A'].ix[date1:date2]

2000-01-03   -2.104569
2000-01-04    7.000000
2000-01-05    0.567020
Freq: D, Name: A, dtype: float64
```

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

```
In [125]: df2 = df[:5].copy()

In [126]: df2.ix[3]

A    7.000000
B    0.721555
C   -1.039575
D    0.271860
Name: 2000-01-04 00:00:00, dtype: float64

In [127]: df2.ix[3] = np.arange(len(df2.columns))

In [128]: df2

                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
```

```
2000-01-04  0.000000  1.000000  2.000000  3.000000
2000-01-05  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 [129]: df.ix[df['A'] > 0, 'B']

2000-01-04    0.721555
2000-01-05   -0.424972
2000-01-06   -0.673690
2000-01-07    0.404705
Freq: D, Name: B, dtype: float64

In [130]: df.ix[date1:date2, 'B']

2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
Freq: D, Name: B, dtype: float64

In [131]: df.ix[date1, 'B']
-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).

9.3.1 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 [132]: df.select(lambda x: x == 'A', axis=1)

                   A
2000-01-01 -0.282863
2000-01-02 -0.173215
2000-01-03 -2.104569
2000-01-04  7.000000
2000-01-05  0.567020
2000-01-06  0.113648
2000-01-07  0.577046
2000-01-08 -1.157892
```

9.3.2 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 [133]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])

In [134]: dflookup.lookup(xrange(0,10,2), ['B','C','A','B','D'])
array([ 0.5277,  0.4201,  0.2442,  0.1239,  0.5722])
```

9.3.3 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 [135]: df2 = df[:4]

In [136]: df2['foo'] = 'bar'

In [137]: print df2
                   A         B         C         D  foo
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632  bar
2000-01-02 -0.173215  1.212112  0.119209 -1.044236  bar
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804  bar
2000-01-04  7.000000  0.721555 -1.039575  0.271860  bar

In [138]: df2.ix[2] = np.nan

In [139]: print df2
                   A         B         C         D  foo
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632  bar
2000-01-02 -0.173215  1.212112  0.119209 -1.044236  bar
2000-01-03       NaN       NaN       NaN       NaN  NaN
2000-01-04  7.000000  0.721555 -1.039575  0.271860  bar

In [140]: print df2.dtypes
A      float64
B      float64
C      float64
D      float64
foo     object
dtype: object
```

9.3.4 Returning a view versus a copy

The rules about when a view on the data is returned are entirely dependent on NumPy. Whenever an array of labels or a boolean vector are involved in the indexing operation, the result will be a copy. With single label / scalar indexing and slicing, e.g. `df.ix[3:6]` or `df.ix[:, 'A']`, a view will be returned.

In chained expressions, the order may determine whether a copy is returned or not:

```
In [141]: dfb = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
   .....:                  'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                  'c' : randn(7)})
   .....:

In [142]: dfb[dfb.a.str.startswith('o')]['c'] = 42  # goes to copy (will be lost)

In [143]: dfb['c'][dfb.a.str.startswith('o')] = 42  # passed via reference (will stay)
```

When assigning values to subsets of your data, thus, make sure to either use the pandas access methods or explicitly handle the assignment creating a copy.

9.3.5 Fallback indexing

Float indexes should be used only with caution. If you have a float indexed `DataFrame` and try to select using an integer, the row that Pandas returns might not be what you expect. Pandas first attempts to use the *integer* as a *label* location, but fails to find a match (because the types are not equal). Pandas then falls back to back to positional indexing.

```
In [144]: df = pd.DataFrame(np.random.randn(4,4),
   .....:      columns=list('ABCD'), index=[1.0, 2.0, 3.0, 4.0])
   .....:

In [145]: df

          A         B         C         D
1 -0.823761  0.535420 -1.032853  1.469725
2  1.304124  1.449735  0.203109 -1.032011
3  0.969818 -0.962723  1.382083 -0.938794
4  0.669142 -0.433567 -0.273610  0.680433

In [146]: df.ix[1]

A    1.304124
B    1.449735
C    0.203109
D   -1.032011
Name: 2.0, dtype: float64
```

To select the row you do expect, instead use a float label or use `iloc`.

```
In [147]: df.ix[1.0]

A   -0.823761
B    0.535420
C   -1.032853
D    1.469725
Name: 1.0, dtype: float64

In [148]: df.iloc[0]

A   -0.823761
B    0.535420
C   -1.032853
D    1.469725
Name: 1.0, dtype: float64
```

Instead of using a float index, it is often better to convert to an integer index:

```
In [149]: df_new = df.reset_index()

In [150]: df_new[df_new['index'] == 1.0]

   index         A        B         C         D
0      1 -0.823761  0.53542 -1.032853  1.469725

# now you can also do "float selection"
In [151]: df_new[(df_new['index'] >= 1.0) & (df_new['index'] < 2)]

   index         A        B         C         D
0      1 -0.823761  0.53542 -1.032853  1.469725
```

9.4 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 [152]: index = Index(['e', 'd', 'a', 'b'])

In [153]: index
Index([u'e', u'd', u'a', u'b'], dtype=object)

In [154]: 'd' in index
True
```

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

```
In [155]: index = Index(['e', 'd', 'a', 'b'], name='something')

In [156]: index.name
'something'
```

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

```
In [157]: index = Index(range(5), name='rows')

In [158]: columns = Index(['A', 'B', 'C'], name='cols')

In [159]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [160]: df

cols         A         B         C
rows
0    -0.308450 -0.276099 -1.821168
1    -1.993606 -1.927385 -2.027924
2     1.624972  0.551135  3.059267
3     0.455264 -0.030740  0.935716
4     1.061192 -2.107852  0.199905

In [161]: df['A']

rows
0      -0.308450
1      -1.993606
2       1.624972
3       0.455264
4       1.061192
Name: A, dtype: float64
```

9.4.1 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 [162]: a = Index(['c', 'b', 'a'])

In [163]: b = Index(['c', 'e', 'd'])
```

```
In [164]: a.union(b)
Index([u'a', u'b', u'c', u'd', u'e'], dtype=object)

In [165]: a | b
Index([u'a', u'b', u'c', u'd', u'e'], dtype=object)

In [166]: a & b
Index([u'c'], dtype=object)

In [167]: a - b
Index([u'a', u'b'], dtype=object)
```

9.4.2 `isin` method of Index objects

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

9.5 Hierarchical indexing (MultiIndex)

Hierarchical indexing (also referred to as "multi-level" indexing) is brand new in the pandas 0.4 release. It is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like Series (1d) and DataFrame (2d).

In this section, we will show what exactly we mean by "hierarchical" indexing and how it integrates with the all of the pandas indexing functionality described above and in prior sections. Later, when discussing *group by* and *pivoting and reshaping data*, we'll show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the *cookbook* for some advanced strategies

Note: Given that hierarchical indexing is so new to the library, it is definitely "bleeding-edge" functionality but is certainly suitable for production. But, there may inevitably be some minor API changes as more use cases are explored and any weaknesses in the design / implementation are identified. pandas aims to be "eminently usable" so any feedback about new functionality like this is extremely helpful.

9.5.1 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 [168]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   .....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   .....:

In [169]: tuples = zip(*arrays)

In [170]: tuples

[('bar', 'one'),
 ('bar', 'two'),
```

```
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [171]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [172]: s = Series(randn(8), index=index)

In [173]: s

first  second
bar    one       0.323586
       two      -0.641630
baz    one      -0.587514
       two       0.053897
foo    one       0.194889
       two      -0.381994
qux    one       0.318587
       two       2.089075
dtype: float64
```

As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

```
In [174]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'])
   .....: ,
   .....:           np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])
   .....:           ]
   .....:

In [175]: s = Series(randn(8), index=arrays)

In [176]: s

bar  one   -0.728293
     two   -0.090255
baz  one   -0.748199
     two    1.318931
foo  one   -2.029766
     two    0.792652
qux  one    0.461007
     two   -0.542749
dtype: float64

In [177]: df = DataFrame(randn(8, 4), index=arrays)

In [178]: df

                0         1         2         3
bar one -0.305384 -0.479195  0.095031 -0.270099
    two -0.707140 -0.773882  0.229453  0.304418
baz one  0.736135 -0.859631 -0.424100 -0.776114
    two  1.279293  0.943798 -1.001859  0.306546
foo one  0.307453 -0.906534 -1.505397  1.392009
    two -0.027793 -0.631023 -0.662357  2.725042
```

```
qux one -1.847240 -0.529247  0.614656 -1.590742
    two -0.156479 -1.696377  0.819712 -2.107728
```

All of the `MultiIndex` constructors accept a `names` argument which stores string names for the levels themselves. If no names are provided, `None` will be assigned:

```
In [179]: df.index.names
[None, None]
```

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

```
In [180]: df = DataFrame(randn(3, 8), index=['A', 'B', 'C'], columns=index)

In [181]: df

first        bar                 baz                 foo                 qux  \
second       one       two       one       two       one       two       one
A      -0.488326  0.851918 -1.242101 -0.654708 -1.647369  0.828258 -0.352362
B       0.289685 -1.982371  0.840166 -0.411403 -2.049028  2.846612 -1.208049
C       2.423905  0.121108  0.266916  0.843826 -0.222540  2.021981 -0.716789
first
second       two
A      -0.814324
B      -0.450392
C      -2.224485

In [182]: DataFrame(randn(6, 6), index=index[:6], columns=index[:6])

first              bar                 baz                 foo
second             one       two       one       two       one       two
first second
bar   one    -1.061137 -0.232825  0.430793 -0.665478  1.829807 -1.406509
      two     1.078248  0.322774  0.200324  0.890024  0.194813  0.351633
baz   one     0.448881 -0.197915  0.965714 -1.522909 -0.116619  0.295575
      two    -1.047704  1.640556  1.905836  2.772115  0.088787 -1.144197
foo   one    -0.633372  0.925372 -0.006438 -0.820408 -0.600874 -1.039266
      two     0.824758 -0.824095 -0.337730 -0.927764 -0.840123  0.248505
```

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 [183]: Series(randn(8), index=tuples)

(bar, one)   -0.109250
(bar, two)    0.431977
(baz, one)   -0.460710
(baz, two)    0.336505
(foo, one)   -3.207595
(foo, two)   -1.535854
(qux, one)    0.409769
(qux, two)   -0.673145
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

Note that how the index is displayed by be controlled using the `multi_sparse` option in

pandas.set_printoptions:

```
In [184]: pd.set_option('display.multi_sparse', False)

In [185]: df

first        bar       bar       baz       baz       foo       foo       qux  \
second       one       two       one       two       one       two       one
A      -0.488326  0.851918 -1.242101 -0.654708 -1.647369  0.828258 -0.352362
B       0.289685 -1.982371  0.840166 -0.411403 -2.049028  2.846612 -1.208049
C       2.423905  0.121108  0.266916  0.843826 -0.222540  2.021981 -0.716789
first        qux
second       two
A      -0.814324
B      -0.450392
C      -2.224485

In [186]: pd.set_option('display.multi_sparse', True)
```

9.5.2 Reconstructing the level labels

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

```
In [187]: index.get_level_values(0)
Index([u'bar', u'bar', u'baz', u'baz', u'foo', u'foo', u'qux', u'qux'], dtype=object)

In [188]: index.get_level_values('second')
Index([u'one', u'two', u'one', u'two', u'one', u'two', u'one', u'two'], dtype=object)
```

9.5.3 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 [189]: df['bar']

second       one       two
A      -0.488326  0.851918
B       0.289685 -1.982371
C       2.423905  0.121108

In [190]: df['bar', 'one']

A   -0.488326
B    0.289685
C    2.423905
Name: (bar, one), dtype: float64

In [191]: df['bar']['one']

A   -0.488326
B    0.289685
C    2.423905
Name: one, dtype: float64
```

```
In [192]: s['qux']

one    0.461007
two   -0.542749
dtype: float64
```

9.5.4 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 [193]: s + s[:-2]

bar  one   -1.456587
     two   -0.180509
baz  one   -1.496398
     two    2.637862
foo  one   -4.059533
     two    1.585304
qux  one         NaN
     two         NaN
dtype: float64

In [194]: s + s[::2]

bar  one   -1.456587
     two         NaN
baz  one   -1.496398
     two         NaN
foo  one   -4.059533
     two         NaN
qux  one    0.922013
     two         NaN
dtype: float64
```

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

```
In [195]: s.reindex(index[:3])

first  second
bar    one      -0.728293
       two      -0.090255
baz    one      -0.748199
dtype: float64

In [196]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])

foo  two    0.792652
bar  one   -0.728293
qux  one    0.461007
baz  one   -0.748199
dtype: float64
```

9.5.5 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 [197]: df = df.T

In [198]: df
```

```
                     A         B         C
first second
bar   one    -0.488326  0.289685  2.423905
      two     0.851918 -1.982371  0.121108
baz   one    -1.242101  0.840166  0.266916
      two    -0.654708 -0.411403  0.843826
foo   one    -1.647369 -2.049028 -0.222540
      two     0.828258  2.846612  2.021981
qux   one    -0.352362 -1.208049 -0.716789
      two    -0.814324 -0.450392 -2.224485
```

```
In [199]: df.ix['bar']
```

```
               A         B         C
second
one    -0.488326  0.289685  2.423905
two     0.851918 -1.982371  0.121108
```

```
In [200]: df.ix['bar', 'two']
```

```
A    0.851918
B   -1.982371
C    0.121108
Name: (bar, two), dtype: float64
```

"Partial" slicing also works quite nicely:

```
In [201]: df.ix['baz':'foo']
```

```
                     A         B         C
first second
baz   one    -1.242101  0.840166  0.266916
      two    -0.654708 -0.411403  0.843826
foo   one    -1.647369 -2.049028 -0.222540
      two     0.828258  2.846612  2.021981
```

```
In [202]: df.ix[('baz', 'two'):('qux', 'one')]
```

```
                     A         B         C
first second
baz   two    -0.654708 -0.411403  0.843826
foo   one    -1.647369 -2.049028 -0.222540
      two     0.828258  2.846612  2.021981
qux   one    -0.352362 -1.208049 -0.716789
```

```
In [203]: df.ix[('baz', 'two'):'foo']
```

```
                     A         B         C
first second
baz   two    -0.654708 -0.411403  0.843826
```

```
foo   one    -1.647369 -2.049028 -0.222540
      two     0.828258  2.846612  2.021981
```

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

```
In [204]: df.ix[[('bar', 'two'), ('qux', 'one')]]

                     A         B         C
first second
bar   two     0.851918 -1.982371  0.121108
qux   one    -0.352362 -1.208049 -0.716789
```

The following does not work, and it's not clear if it should or not:

```
>>> df.ix[['bar', 'qux']]
```

The code for implementing `.ix` makes every attempt to "do the right thing" but as you use it you may uncover corner cases or unintuitive behavior. If you do find something like this, do not hesitate to report the issue or ask on the mailing list.

9.5.6 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 [205]: df.xs('one', level='second')

              A         B         C
first
bar   -0.488326  0.289685  2.423905
baz   -1.242101  0.840166  0.266916
foo   -1.647369 -2.049028 -0.222540
qux   -0.352362 -1.208049 -0.716789
```

9.5.7 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 [206]: midx = MultiIndex(levels=[['zero', 'one'], ['x','y']],
   .....:                   labels=[[1,1,0,0],[1,0,1,0]])
   .....:

In [207]: df = DataFrame(randn(4,2), index=midx)

In [208]: print df
                0         1
one  y -0.741113 -0.110891
     x -2.672910  0.864492
zero y  0.060868  0.933092
     x  0.288841  1.324969

In [209]: df2 = df.mean(level=0)

In [210]: print df2
             0        1
```

```
zero  0.174854  1.12903
one  -1.707011  0.37680

In [211]: print df2.reindex(df.index, level=0)
                0        1
one  y -1.707011  0.37680
     x -1.707011  0.37680
zero y  0.174854  1.12903
     x  0.174854  1.12903

In [212]: df_aligned, df2_aligned = df.align(df2, level=0)

In [213]: print df_aligned
                0         1
one  y -0.741113 -0.110891
     x -2.672910  0.864492
zero y  0.060868  0.933092
     x  0.288841  1.324969

In [214]: print df2_aligned
                0        1
one  y -1.707011  0.37680
     x -1.707011  0.37680
zero y  0.174854  1.12903
     x  0.174854  1.12903
```

9.5.8 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 [215]: import random; random.shuffle(tuples)

In [216]: s = Series(randn(8), index=MultiIndex.from_tuples(tuples))

In [217]: s

bar  one    0.589220
foo  two    0.531415
bar  two   -1.198747
foo  one   -0.236866
qux  one   -1.317798
baz  two    0.373766
     one   -0.675588
qux  two    0.981295
dtype: float64

In [218]: s.sortlevel(0)

bar  one    0.589220
     two   -1.198747
```

```
baz  one   -0.675588
     two    0.373766
foo  one   -0.236866
     two    0.531415
qux  one   -1.317798
     two    0.981295
dtype: float64
```

```
In [219]: s.sortlevel(1)
```

```
bar  one    0.589220
baz  one   -0.675588
foo  one   -0.236866
qux  one   -1.317798
bar  two   -1.198747
baz  two    0.373766
foo  two    0.531415
qux  two    0.981295
dtype: float64
```

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

```
In [220]: s.index.names = ['L1', 'L2']
```

```
In [221]: s.sortlevel(level='L1')
```

```
L1   L2
bar  one    0.589220
     two   -1.198747
baz  one   -0.675588
     two    0.373766
foo  one   -0.236866
     two    0.531415
qux  one   -1.317798
     two    0.981295
dtype: float64
```

```
In [222]: s.sortlevel(level='L2')
```

```
L1   L2
bar  one    0.589220
baz  one   -0.675588
foo  one   -0.236866
qux  one   -1.317798
bar  two   -1.198747
baz  two    0.373766
foo  two    0.531415
qux  two    0.981295
dtype: float64
```

Some indexing will work even if the data are not sorted, but will be rather inefficient and will also return a copy of the data rather than a view:

```
In [223]: s['qux']
```

```
L2
one   -1.317798
two    0.981295
dtype: float64
```

```
In [224]: s.sortlevel(1)['qux']

L2
one   -1.317798
two    0.981295
dtype: float64
```

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

```
In [225]: df.T.sortlevel(1, axis=1)

       zero       one      zero       one
          x         x         y         y
0  0.288841 -2.672910  0.060868 -0.741113
1  1.324969  0.864492  0.933092 -0.110891
```

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 [226]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]

In [227]: idx = MultiIndex.from_tuples(tuples)

In [228]: idx.lexsort_depth
2

In [229]: reordered = idx[[1, 0, 3, 2]]

In [230]: reordered.lexsort_depth
1

In [231]: s = Series(randn(4), index=reordered)

In [232]: s.ix['a':'a']

a  b   -0.100323
   a    0.935523
dtype: float64
```

However:

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

9.5.9 Swapping levels with `swaplevel`

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

```
In [233]: df[:5]

               0         1
one  y -0.741113 -0.110891
     x -2.672910  0.864492
zero y  0.060868  0.933092
     x  0.288841  1.324969

In [234]: df[:5].swaplevel(0, 1, axis=0)
```

```
                0         1
y one  -0.741113 -0.110891
x one  -2.672910  0.864492
y zero  0.060868  0.933092
x zero  0.288841  1.324969
```

9.5.10 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 [235]: df[:5].reorder_levels([1,0], axis=0)
```

```
                0         1
y one  -0.741113 -0.110891
x one  -2.672910  0.864492
y zero  0.060868  0.933092
x zero  0.288841  1.324969
```

9.5.11 Some gory internal details

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

```
In [236]: index

MultiIndex
[(u'bar', u'one'), (u'bar', u'two'), (u'baz', u'one'), (u'baz', u'two'), (u'foo', u'one'), (u'foo', u

In [237]: index.levels

[Index([u'bar', u'baz', u'foo', u'qux'], dtype=object),
 Index([u'one', u'two'], dtype=object)]

In [238]: index.labels
[array([0, 0, 1, 1, 2, 2, 3, 3]), array([0, 1, 0, 1, 0, 1, 0, 1])]

In [239]: index.names
['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.

9.6 Adding an index to an existing DataFrame

Occasionally you will load or create a data set into a DataFrame and want to add an index after you've already done so. There are a couple of different ways.

9.6.1 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 [240]: data

     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4

In [241]: indexed1 = data.set_index('c')

In [242]: indexed1

     a    b  d
c
z  bar  one  1
y  bar  two  2
x  foo  one  3
w  foo  two  4

In [243]: indexed2 = data.set_index(['a', 'b'])

In [244]: indexed2

         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
    two  w  4
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [245]: frame = data.set_index('c', drop=False)

In [246]: frame = frame.set_index(['a', 'b'], append=True)

In [247]: frame

           c  d
c a   b
z bar one  z  1
y bar two  y  2
x foo one  x  3
w foo two  w  4
```

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 [248]: data.set_index('c', drop=False)

     a    b  c  d
c
z  bar  one  z  1
y  bar  two  y  2
```

```
x  foo  one  x  3
w  foo  two  w  4
```

```
In [249]: data.set_index(['a', 'b'], inplace=True)
```

```
In [250]: data
```

```
          c  d
a   b
bar one   z  1
    two   y  2
foo one   x  3
    two   w  4
```

9.6.2 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 [251]: data
```

```
          c  d
a   b
bar one   z  1
    two   y  2
foo one   x  3
    two   w  4
```

```
In [252]: data.reset_index()
```

```
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4
```

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

You can use the `level` keyword to remove only a portion of the index:

```
In [253]: frame
```

```
            c  d
c a   b
z bar one   z  1
y bar two   y  2
x foo one   x  3
w foo two   w  4
```

```
In [254]: frame.reset_index(level=1)
```

```
          a  c  d
c b
z one   bar  z  1
y two   bar  y  2
x one   foo  x  3
w two   foo  w  4
```

`reset_index` takes an optional parameter `drop` which if true simply discards the index, instead of putting index values in the DataFrame's columns.

Note: The `reset_index` method used to be called `delevel` which is now deprecated.

9.6.3 Adding an ad hoc index

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

```
data.index = index
```

9.7 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
- `date_range`: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Python datetime objects

The motivation for having an `Index` class in the first place was to enable different implementations of indexing. This means that it's possible for you, the user, to implement a custom `Index` subclass that may be better suited to a particular application than the ones provided in pandas.

From an internal implementation point of view, the relevant methods that an `Index` must define are one or more of the following (depending on how incompatible the new object internals are with the `Index` functions):

- `get_loc`: returns an "indexer" (an integer, or in some cases a slice object) for a label
- `slice_locs`: returns the "range" to slice between two labels
- `get_indexer`: Computes the indexing vector for reindexing / data alignment purposes. See the source / docstrings for more on this
- `get_indexer_non_unique`: Computes the indexing vector for reindexing / data alignment purposes when the index is non-unique. See the source / docstrings for more on this
- `reindex`: Does any pre-conversion of the input index then calls `get_indexer`
- `union`, `intersection`: computes the union or intersection of two Index objects
- `insert`: Inserts a new label into an Index, yielding a new object
- `delete`: Delete a label, yielding a new object
- `drop`: Deletes a set of labels
- `take`: Analogous to ndarray.take

CHAPTER

TEN

COMPUTATIONAL TOOLS

10.1 Statistical functions

10.1.1 Percent Change

Both `Series` and `DataFrame` has a method `pct_change` to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values).

```
In [1]: ser = Series(randn(8))

In [2]: ser.pct_change()

0         NaN
1   -1.602976
2    4.334938
3   -0.247456
4   -2.067345
5   -1.142903
6   -1.688214
7   -9.759729
dtype: float64

In [3]: df = DataFrame(randn(10, 4))

In [4]: df.pct_change(periods=3)

          0         1         2         3
0       NaN       NaN       NaN       NaN
1       NaN       NaN       NaN       NaN
2       NaN       NaN       NaN       NaN
3 -0.218320 -1.054001  1.987147 -0.510183
4 -0.439121 -1.816454  0.649715 -4.822809
5 -0.127833 -3.042065 -5.866604 -1.776977
6 -2.596833 -1.959538 -2.111697 -3.798900
7 -0.117826 -2.169058  0.036094 -0.067696
8  2.492606 -1.357320 -1.205802 -1.558697
9 -1.012977  2.324558 -1.003744 -0.371806
```

10.1.2 Covariance

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

```
In [5]: s1 = Series(randn(1000))

In [6]: s2 = Series(randn(1000))

In [7]: s1.cov(s2)
0.0006801088174310957
```

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

```
In [8]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [9]: frame.cov()

          a         b         c         d         e
a  1.000882 -0.003177 -0.002698 -0.006889  0.031912
b -0.003177  1.024721  0.000191  0.009212  0.000857
c -0.002698  0.000191  0.950735 -0.031743 -0.005087
d -0.006889  0.009212 -0.031743  1.002983 -0.047952
e  0.031912  0.000857 -0.005087 -0.047952  1.042487
```

`DataFrame.cov` also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```
In [10]: frame = DataFrame(randn(20, 3), columns=['a', 'b', 'c'])

In [11]: frame.ix[:5, 'a'] = np.nan

In [12]: frame.ix[5:10, 'b'] = np.nan

In [13]: frame.cov()

          a         b         c
a  1.210090 -0.430629  0.018002
b -0.430629  1.240960  0.347188
c  0.018002  0.347188  1.301149

In [14]: frame.cov(min_periods=12)

          a         b         c
a  1.210090       NaN  0.018002
b       NaN  1.240960  0.347188
c  0.018002  0.347188  1.301149
```

10.1.3 Correlation

Several methods for computing correlations are provided. Several kinds of correlation methods are provided:

Method name	Description
`pearson (default)`	Standard correlation coefficient
`kendall`	Kendall Tau correlation coefficient
`spearman`	Spearman rank correlation coefficient

All of these are currently computed using pairwise complete observations.

```
In [15]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [16]: frame.ix[::2] = np.nan
```

```
# Series with Series
In [17]: frame['a'].corr(frame['b'])
0.013479040400098763

In [18]: frame['a'].corr(frame['b'], method='spearman')
-0.0072898851595406388

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()

          a         b         c         d         e
a  1.000000  0.013479 -0.049269 -0.042239 -0.028525
b  0.013479  1.000000 -0.020433 -0.011139  0.005654
c -0.049269 -0.020433  1.000000  0.018587 -0.054269
d -0.042239 -0.011139  0.018587  1.000000 -0.017060
e -0.028525  0.005654 -0.054269 -0.017060  1.000000
```

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

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```
In [20]: frame = DataFrame(randn(20, 3), columns=['a', 'b', 'c'])

In [21]: frame.ix[:5, 'a'] = np.nan

In [22]: frame.ix[5:10, 'b'] = np.nan

In [23]: frame.corr()

          a         b         c
a  1.000000 -0.076520  0.160092
b -0.076520  1.000000  0.135967
c  0.160092  0.135967  1.000000

In [24]: frame.corr(min_periods=12)

          a         b         c
a  1.000000       NaN  0.160092
b       NaN  1.000000  0.135967
c  0.160092  0.135967  1.000000
```

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

```
In [25]: index = ['a', 'b', 'c', 'd', 'e']

In [26]: columns = ['one', 'two', 'three', 'four']

In [27]: df1 = DataFrame(randn(5, 4), index=index, columns=columns)

In [28]: df2 = DataFrame(randn(4, 4), index=index[:4], columns=columns)

In [29]: df1.corrwith(df2)

one     -0.125501
two     -0.493244
three    0.344056
four     0.004183
dtype: float64
```

```
In [30]: df2.corrwith(df1, axis=1)

a   -0.675817
b    0.458296
c    0.190809
d   -0.186275
e         NaN
dtype: float64
```

10.1.4 Data ranking

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

```
In [31]: s = Series(np.random.randn(5), index=list('abcde'))

In [32]: s['d'] = s['b'] # so there's a tie

In [33]: s.rank()

a    5.0
b    2.5
c    1.0
d    2.5
e    4.0
dtype: float64
```

`rank` is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). `NaN` values are excluded from the ranking.

```
In [34]: df = DataFrame(np.random.randn(10, 6))

In [35]: df[4] = df[2][:5] # some ties

In [36]: df

          0         1         2         3         4         5
0 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.204848       NaN  0.463347
6  0.376892  0.959292  0.095572 -0.593740       NaN -0.069180
7 -1.002601  1.957794 -0.120708  0.094214       NaN -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254       NaN -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477       NaN  1.375064

In [37]: df.rank(1)

   0  1    2  3    4  5
0  4  3  1.5  5  1.5  6
1  2  6  4.5  1  4.5  3
2  1  6  3.5  5  3.5  2
3  4  5  1.5  3  1.5  6
4  5  3  1.5  4  1.5  6
5  1  2  5.0  3  NaN  4
6  4  5  3.0  1  NaN  2
```

```
7  2  5  3.0  4  NaN  1
8  2  5  3.0  4  NaN  1
9  2  3  1.0  4  NaN  5
```

`rank` optionally takes a parameter `ascending` which by default is true; when false, data is reverse-ranked, with larger values assigned a smaller rank.

`rank` supports different tie-breaking methods, specified with the `method` parameter:

- `average` : average rank of tied group
- `min` : lowest rank in the group
- `max` : highest rank in the group
- `first` : ranks assigned in the order they appear in the array

10.2 Moving (rolling) statistics / moments

For working with time series data, a number of functions are provided for computing common *moving* or *rolling* statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis. All of these methods are in the `pandas` namespace, but otherwise they can be found in `pandas.stats.moments`.

Function	Description
`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
`rolling_window`	Moving window function

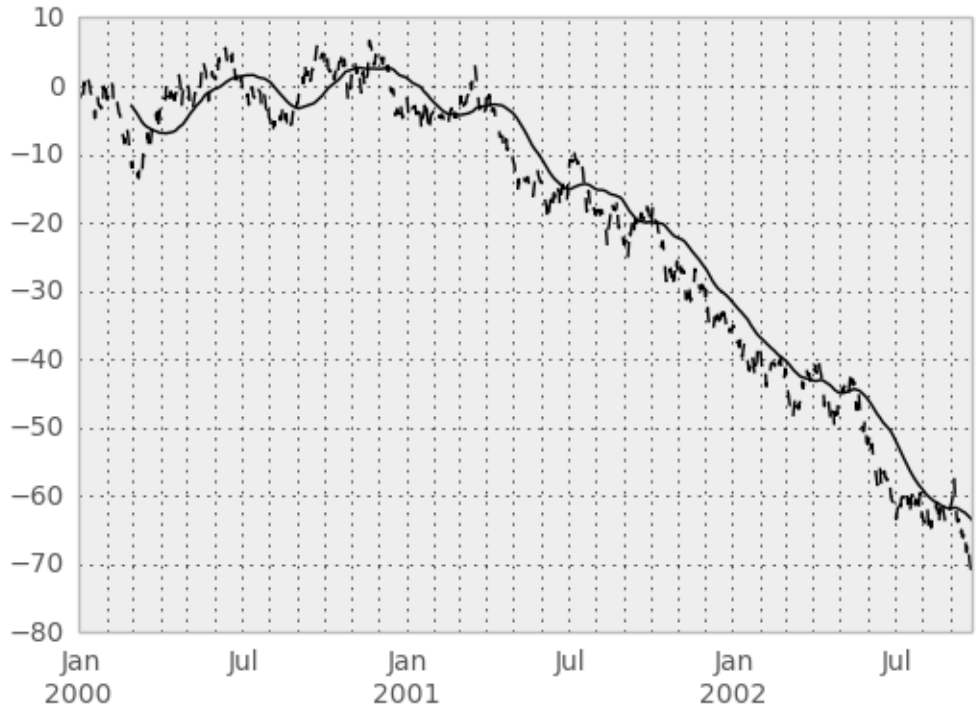
Generally these methods all have the same interface. The binary operators (e.g. `rolling_corr`) take two Series or DataFrames. Otherwise, they all accept the following arguments:

- `window`: size of moving window
- `min_periods`: threshold of non-null data points to require (otherwise result is NA)
- `freq`: optionally specify a *frequency string* or *DateOffset* to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument `time_rule` was used instead of `freq` that referred to the legacy time rule constants

These functions can be applied to ndarrays or Series objects:

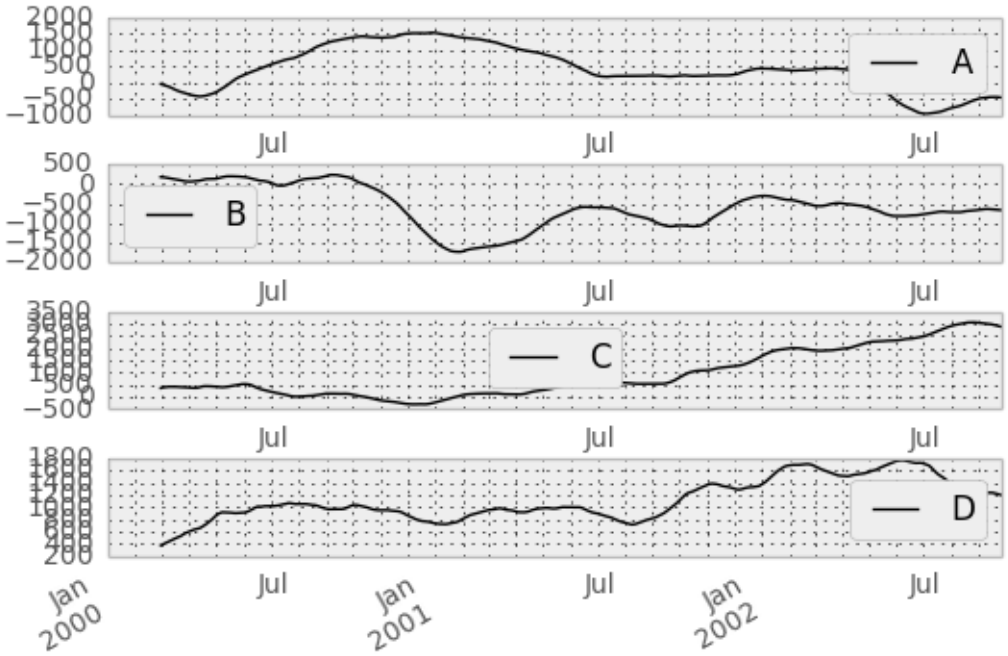
```
In [38]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [39]: ts = ts.cumsum()
```

```
In [40]: ts.plot(style='k--')
<matplotlib.axes.AxesSubplot at 0x635b4d0>

In [41]: rolling_mean(ts, 60).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x635b4d0>
```



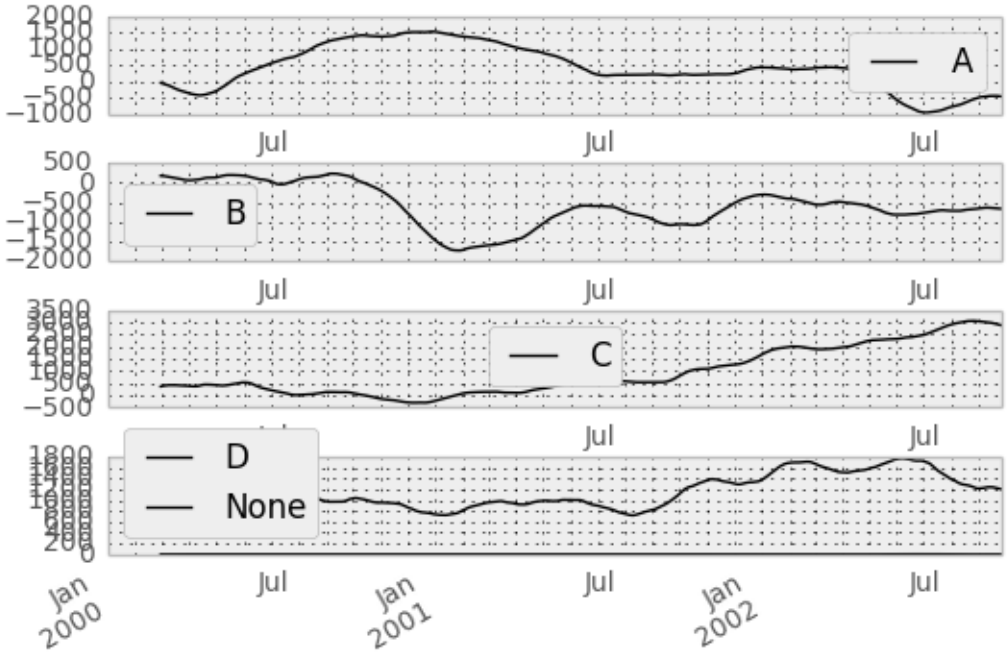
They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrame's columns:

```
In [42]: df = DataFrame(randn(1000, 4), index=ts.index,
   ....:                columns=['A', 'B', 'C', 'D'])
   ....:

In [43]: df = df.cumsum()

In [44]: rolling_sum(df, 60).plot(subplots=True)

array([<matplotlib.axes.AxesSubplot object at 0x5f7f390>,
       <matplotlib.axes.AxesSubplot object at 0x67483d0>,
       <matplotlib.axes.AxesSubplot object at 0x628e790>,
       <matplotlib.axes.AxesSubplot object at 0x5eef150>], dtype=object)
```



The `rolling_apply` function takes an extra `func` argument and performs generic rolling computations. The `func` argument should be a single function that produces a single value from an ndarray input. Suppose we wanted to compute the mean absolute deviation on a rolling basis:

```
In [45]: mad = lambda x: np.fabs(x - x.mean()).mean()

In [46]: rolling_apply(ts, 60, mad).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x5eef150>
```



The `rolling_window` function performs a generic rolling window computation on the input data. The weights

used in the window are specified by the `win_type` keyword. The list of recognized types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

```
In [47]: ser = Series(randn(10), index=date_range('1/1/2000', periods=10))

In [48]: rolling_window(ser, 5, 'triang')

2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.622722
2000-01-06   -0.460623
2000-01-07   -0.229918
2000-01-08   -0.237308
2000-01-09   -0.335064
2000-01-10   -0.403449
Freq: D, dtype: float64
```

Note that the `boxcar` window is equivalent to `rolling_mean`:

```
In [49]: rolling_window(ser, 5, 'boxcar')

2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64

In [50]: rolling_mean(ser, 5)
```

```
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [51]: rolling_window(ser, 5, 'gaussian', std=0.1)
```

```
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.261998
2000-01-06   -0.230600
2000-01-07    0.121276
2000-01-08   -0.136220
2000-01-09   -0.057945
2000-01-10   -0.199326
Freq: D, dtype: float64
```

By default the labels are set to the right edge of the window, but a `center` keyword is available so the labels can be set at the center. This keyword is available in other rolling functions as well.

```
In [52]: rolling_window(ser, 5, 'boxcar')
```

```
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

```
In [53]: rolling_window(ser, 5, 'boxcar', center=True)
```

```
2000-01-01         NaN
2000-01-02         NaN
2000-01-03   -0.841164
2000-01-04   -0.779948
2000-01-05   -0.565487
2000-01-06   -0.502815
2000-01-07   -0.553755
2000-01-08   -0.472211
2000-01-09         NaN
2000-01-10         NaN
Freq: D, dtype: float64
```

```
In [54]: rolling_mean(ser, 5, center=True)

2000-01-01         NaN
2000-01-02         NaN
2000-01-03   -0.841164
2000-01-04   -0.779948
2000-01-05   -0.565487
2000-01-06   -0.502815
2000-01-07   -0.553755
2000-01-08   -0.472211
2000-01-09         NaN
2000-01-10         NaN
Freq: D, dtype: float64
```

10.2.1 Binary rolling moments

`rolling_cov` and `rolling_corr` can compute moving window statistics about two `Series` or any combination of `DataFrame/Series` or `DataFrame/DataFrame`. Here is the behavior in each case:

- two `Series`: compute the statistic for the pairing
- `DataFrame/Series`: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame
- `DataFrame/DataFrame`: compute statistic for matching column names, returning a DataFrame

For example:

```
In [55]: df2 = df[:20]

In [56]: rolling_corr(df2, df2['B'], window=5)

                   A   B         C         D
2000-01-01       NaN NaN       NaN       NaN
2000-01-02       NaN NaN       NaN       NaN
2000-01-03       NaN NaN       NaN       NaN
2000-01-04       NaN NaN       NaN       NaN
2000-01-05 -0.262853   1  0.334449  0.193380
2000-01-06 -0.083745   1 -0.521587 -0.556126
2000-01-07 -0.292940   1 -0.658532 -0.458128
2000-01-08  0.840416   1  0.796505 -0.498672
2000-01-09 -0.135275   1  0.753895 -0.634445
2000-01-10 -0.346229   1 -0.682232 -0.645681
2000-01-11 -0.365524   1 -0.775831 -0.561991
2000-01-12 -0.204761   1 -0.855874 -0.382232
2000-01-13  0.575218   1 -0.747531  0.167892
2000-01-14  0.519499   1 -0.687277  0.192822
2000-01-15  0.048982   1  0.167669 -0.061463
2000-01-16  0.217190   1  0.167564 -0.326034
2000-01-17  0.641180   1 -0.164780 -0.111487
2000-01-18  0.130422   1  0.322833  0.632383
2000-01-19  0.317278   1  0.384528  0.813656
2000-01-20  0.293598   1  0.159538  0.742381
```

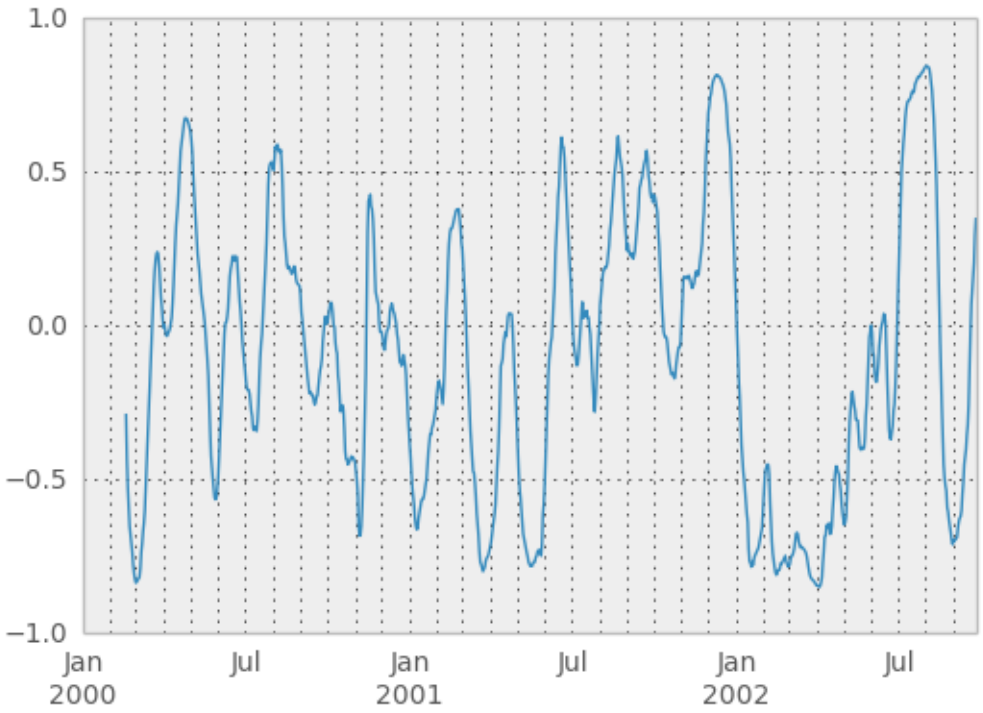
10.2.2 Computing rolling pairwise correlations

In financial data analysis and other fields it's common to compute correlation matrices for a collection of time series. More difficult is to compute a moving-window correlation matrix. This can be done using the `rolling_corr_pairwise` function, which yields a `Panel` whose `items` are the dates in question:

```
In [57]: correls = rolling_corr_pairwise(df, 50)

In [58]: correls[df.index[-50]]

          A         B         C         D
A  1.000000  0.604221  0.767429 -0.776170
B  0.604221  1.000000  0.461484 -0.381148
C  0.767429  0.461484  1.000000 -0.748863
D -0.776170 -0.381148 -0.748863  1.000000
```

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

```
In [59]: correls.ix[:, 'A', 'C'].plot()
<matplotlib.axes.AxesSubplot at 0x78f4690>
```



10.3 Expanding window moment functions

A common alternative to rolling statistics is to use an *expanding* window, which yields the value of the statistic with all the data available up to that point in time. As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```
In [60]: rolling_mean(df, window=len(df), min_periods=1)[:5]

                   A         B         C         D
```

```
2000-01-01 -1.388345  3.317290  0.344542 -0.036968
2000-01-02 -1.123132  3.622300  1.675867  0.595300
2000-01-03 -0.628502  3.626503  2.455240  1.060158
2000-01-04 -0.768740  3.888917  2.451354  1.281874
2000-01-05 -0.824034  4.108035  2.556112  1.140723
```

```
In [61]: expanding_mean(df)[:5]
```

```
                   A         B         C         D
2000-01-01 -1.388345  3.317290  0.344542 -0.036968
2000-01-02 -1.123132  3.622300  1.675867  0.595300
2000-01-03 -0.628502  3.626503  2.455240  1.060158
2000-01-04 -0.768740  3.888917  2.451354  1.281874
2000-01-05 -0.824034  4.108035  2.556112  1.140723
```

Like the `rolling_` functions, the following methods are included in the `pandas` namespace or can be located in `pandas.stats.moments`.

Function	Description
`expanding_count`	Number of non-null observations
`expanding_sum`	Sum of values
`expanding_mean`	Mean of values
`expanding_median`	Arithmetic median of values
`expanding_min`	Minimum
`expanding_max`	Maximum
`expanding_std`	Unbiased standard deviation
`expanding_var`	Unbiased variance
`expanding_skew`	Unbiased skewness (3rd moment)
`expanding_kurt`	Unbiased kurtosis (4th moment)
`expanding_quantile`	Sample quantile (value at %)
`expanding_apply`	Generic apply
`expanding_cov`	Unbiased covariance (binary)
`expanding_corr`	Correlation (binary)
`expanding_corr_pairwise`	Pairwise correlation of DataFrame columns

Aside from not having a `window` parameter, these functions have the same interfaces as their `rolling_` counterpart. Like above, the parameters they all accept are:

- `min_periods`: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No `NaNs` will be output once `min_periods` non-null data points have been seen.
- `freq`: optionally specify a *frequency string* or *DateOffset* to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument `time_rule` was used instead of `freq` that referred to the legacy time rule constants

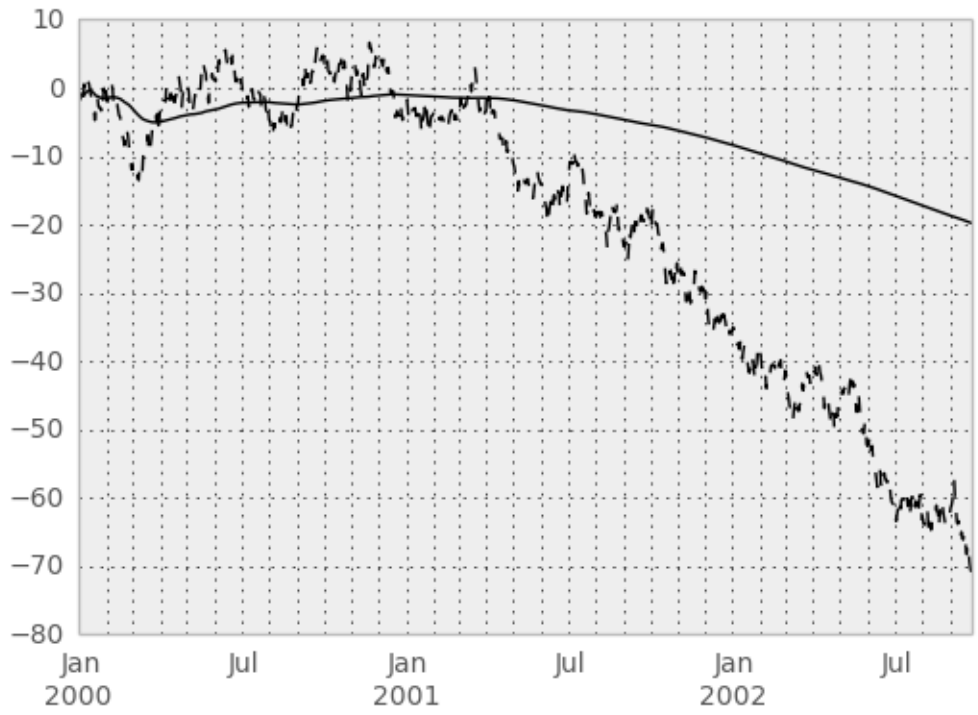
---

Note: The output of the `rolling_` and `expanding_` functions do not return a `NaN` if there are at least `min_periods` non-null values in the current window. This differs from `cumsum`, `cumprod`, `cummax`, and `cummin`, which return `NaN` in the output wherever a `NaN` is encountered in the input.

An expanding window statistic will be more stable (and less responsive) than its rolling window counterpart as the increasing window size decreases the relative impact of an individual data point. As an example, here is the `expanding_mean` output for the previous time series dataset:

```
In [62]: ts.plot(style='k--')
<matplotlib.axes.AxesSubplot at 0x6372750>
```

```
In [63]: expanding_mean(ts).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x6372750>
```



10.4 Exponentially weighted moment functions

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

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

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

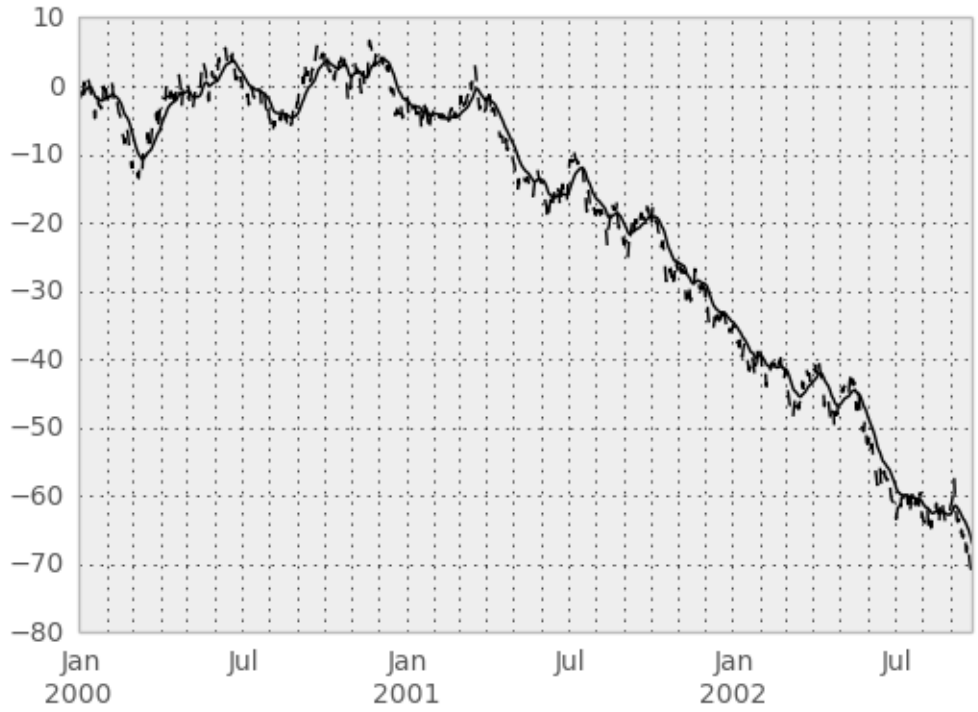
$$\alpha = \begin{cases} \frac{2}{s+1}, s = \text{span} \\ \frac{1}{c+1}, c = \text{center of mass} \end{cases}$$

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
`ewmvar`	EW moving variance
`ewmstd`	EW moving standard deviation
`ewmcorr`	EW moving correlation
`ewmcov`	EW moving covariance

Here are an example for a univariate time series:

```
In [64]: plt.close('all')

In [65]: ts.plot(style='k--')
<matplotlib.axes.AxesSubplot at 0x6166150>

In [66]: ewma(ts, span=20).plot(style='k')
<matplotlib.axes.AxesSubplot at 0x6166150>
```



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.

CHAPTER

ELEVEN

WORKING WITH MISSING DATA

In this section, we will discuss missing (also referred to as NA) values in pandas.

Note: The choice of using `NaN` internally to denote missing data was largely for simplicity and performance reasons. It differs from the MaskedArray approach of, for example, `scikits.timeseries`. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

See the *cookbook* for some advanced strategies

11.1 Missing data basics

11.1.1 When / why does data become missing?

Some might quibble over our usage of *missing*. By "missing" we simply mean **null** or "not present for whatever reason". Many data sets simply arrive with missing data, either because it exists and was not collected or it never existed. For example, in a collection of financial time series, some of the time series might start on different dates. Thus, values prior to the start date would generally be marked as missing.

In pandas, one of the most common ways that missing data is **introduced** into a data set is by reindexing. For example

```
In [1]: df = DataFrame(randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   ...:                columns=['one', 'two', 'three'])
   ...:

In [2]: df['four'] = 'bar'

In [3]: df['five'] = df['one'] > 0

In [4]: df

        one       two     three four   five
a -0.438460  1.619664 -0.156589  bar  False
c -0.426514 -1.028828  0.409237  bar  False
e  1.422925  1.199683 -0.106996  bar   True
f -0.908243  1.422547 -0.647947  bar  False
h  0.087149 -1.679253 -1.636722  bar   True

In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
```

```
        one       two     three four   five
a -0.438460  1.619664 -0.156589  bar  False
b       NaN       NaN       NaN  NaN    NaN
c -0.426514 -1.028828  0.409237  bar  False
d       NaN       NaN       NaN  NaN    NaN
e  1.422925  1.199683 -0.106996  bar   True
f -0.908243  1.422547 -0.647947  bar  False
g       NaN       NaN       NaN  NaN    NaN
h  0.087149 -1.679253 -1.636722  bar   True
```

11.1.2 Values considered "missing"

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While `NaN` is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python `None` will arise and we wish to also consider that "missing" or "null".

Until recently, for legacy reasons `inf` and `-inf` were also considered to be "null" in computations. This is no longer the case by default; use the `mode.use_inf_as_null` option to recover it. To make detecting missing values easier (and across different array dtypes), pandas provides the `isnull()` and `notnull()` functions, which are also methods on `Series` objects:

```
In [7]: df2['one']

a   -0.438460
b         NaN
c   -0.426514
d         NaN
e    1.422925
f   -0.908243
g         NaN
h    0.087149
Name: one, dtype: float64

In [8]: isnull(df2['one'])

a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool

In [9]: df2['four'].notnull()

a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
dtype: bool
```

Summary: `NaN` and `None` (in object arrays) are considered missing by the `isnull` and `notnull` functions. `inf` and `-inf` are no longer considered missing by default.

11.2 Datetimes

For datetime64[ns] types, `NaT` represents missing values. This is a pseudo-native sentinal value that can be represented by numpy in a singular dtype (datetime64[ns]). Pandas objects provide intercompatibility between `NaT` and `NaN`.

```
In [10]: df2 = df.copy()

In [11]: df2['timestamp'] = Timestamp('20120101')

In [12]: df2

        one       two     three four   five           timestamp
a -0.438460  1.619664 -0.156589  bar  False 2012-01-01 00:00:00
c -0.426514 -1.028828  0.409237  bar  False 2012-01-01 00:00:00
e  1.422925  1.199683 -0.106996  bar   True 2012-01-01 00:00:00
f -0.908243  1.422547 -0.647947  bar  False 2012-01-01 00:00:00
h  0.087149 -1.679253 -1.636722  bar   True 2012-01-01 00:00:00

In [13]: df2.ix[['a','c','h'],['one','timestamp']] = np.nan

In [14]: df2

        one       two     three four   five           timestamp
a       NaN  1.619664 -0.156589  bar  False                 NaT
c       NaN -1.028828  0.409237  bar  False                 NaT
e  1.422925  1.199683 -0.106996  bar   True 2012-01-01 00:00:00
f -0.908243  1.422547 -0.647947  bar  False 2012-01-01 00:00:00
h       NaN -1.679253 -1.636722  bar   True                 NaT

In [15]: df2.get_dtype_counts()

bool              1
datetime64[ns]    1
float64           3
object            1
dtype: int64
```

11.3 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [16]: a

        one       two
a       NaN  1.619664
c       NaN -1.028828
e  1.422925  1.199683
f -0.908243  1.422547
h -0.908243 -1.679253

In [17]: b
```

```
        one       two     three
a       NaN  1.619664 -0.156589
c       NaN -1.028828  0.409237
e  1.422925  1.199683 -0.106996
f -0.908243  1.422547 -0.647947
h       NaN -1.679253 -1.636722

In [18]: a + b

        one  three       two
a       NaN    NaN  3.239329
c       NaN    NaN -2.057655
e  2.845850    NaN  2.399366
f -1.816486    NaN  2.845093
h       NaN    NaN -3.358505
```

The descriptive statistics and computational methods discussed in the *data structure overview* (and listed *here* and *here*) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero
- If the data are all NA, the result will be NA
- Methods like **cumsum** and **cumprod** ignore NA values, but preserve them in the resulting arrays

```
In [19]: df

        one       two     three
a       NaN  1.619664 -0.156589
c       NaN -1.028828  0.409237
e  1.422925  1.199683 -0.106996
f -0.908243  1.422547 -0.647947
h       NaN -1.679253 -1.636722

In [20]: df['one'].sum()
0.51468201281996417

In [21]: df.mean(1)

a    0.731538
c   -0.309795
e    0.838537
f   -0.044548
h   -1.657987
dtype: float64

In [22]: df.cumsum()

        one       two     three
a       NaN  1.619664 -0.156589
c       NaN  0.590837  0.252648
e  1.422925  1.790520  0.145653
f  0.514682  3.213067 -0.502294
h       NaN  1.533814 -2.139016
```

11.3.1 NA values in GroupBy

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

11.4 Cleaning / filling missing data

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

11.4.1 Filling missing values: fillna

The **fillna** function can "fill in" NA values with non-null data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [23]: df2

        one       two     three four   five           timestamp
a       NaN  1.619664 -0.156589  bar  False                 NaT
c       NaN -1.028828  0.409237  bar  False                 NaT
e  1.422925  1.199683 -0.106996  bar   True 2012-01-01 00:00:00
f -0.908243  1.422547 -0.647947  bar  False 2012-01-01 00:00:00
h       NaN -1.679253 -1.636722  bar   True                 NaT

In [24]: df2.fillna(0)

        one       two     three four   five           timestamp
a  0.000000  1.619664 -0.156589  bar  False 1970-01-01 00:00:00
c  0.000000 -1.028828  0.409237  bar  False 1970-01-01 00:00:00
e  1.422925  1.199683 -0.106996  bar   True 2012-01-01 00:00:00
f -0.908243  1.422547 -0.647947  bar  False 2012-01-01 00:00:00
h  0.000000 -1.679253 -1.636722  bar   True 1970-01-01 00:00:00

In [25]: df2['four'].fillna('missing')

a    bar
c    bar
e    bar
f    bar
h    bar
Name: four, dtype: object
```

Fill gaps forward or backward

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

```
In [26]: df

        one       two     three
a       NaN  1.619664 -0.156589
c       NaN -1.028828  0.409237
e  1.422925  1.199683 -0.106996
f -0.908243  1.422547 -0.647947
h       NaN -1.679253 -1.636722

In [27]: df.fillna(method='pad')

        one       two     three
a       NaN  1.619664 -0.156589
c       NaN -1.028828  0.409237
e  1.422925  1.199683 -0.106996
f -0.908243  1.422547 -0.647947
h -0.908243 -1.679253 -1.636722
```

Limit the amount of filling

If we only want consecutive gaps filled up to a certain number of data points, we can use the *limit* keyword:

```
In [28]: df

   one       two     three
a  NaN  1.619664 -0.156589
c  NaN -1.028828  0.409237
e  NaN       NaN       NaN
f  NaN       NaN       NaN
h  NaN -1.679253 -1.636722

In [29]: df.fillna(method='pad', limit=1)

   one       two     three
a  NaN  1.619664 -0.156589
c  NaN -1.028828  0.409237
e  NaN -1.028828  0.409237
f  NaN       NaN       NaN
h  NaN -1.679253 -1.636722
```

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.

11.4.2 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 [30]: df

   one       two     three
a  NaN  1.619664 -0.156589
c  NaN -1.028828  0.409237
e  NaN  0.000000  0.000000
f  NaN  0.000000  0.000000
h  NaN -1.679253 -1.636722

In [31]: df.dropna(axis=0)

Empty DataFrame
Columns: [one, two, three]
Index: []

In [32]: df.dropna(axis=1)

        two     three
a  1.619664 -0.156589
c -1.028828  0.409237
e  0.000000  0.000000
f  0.000000  0.000000
h -1.679253 -1.636722
```

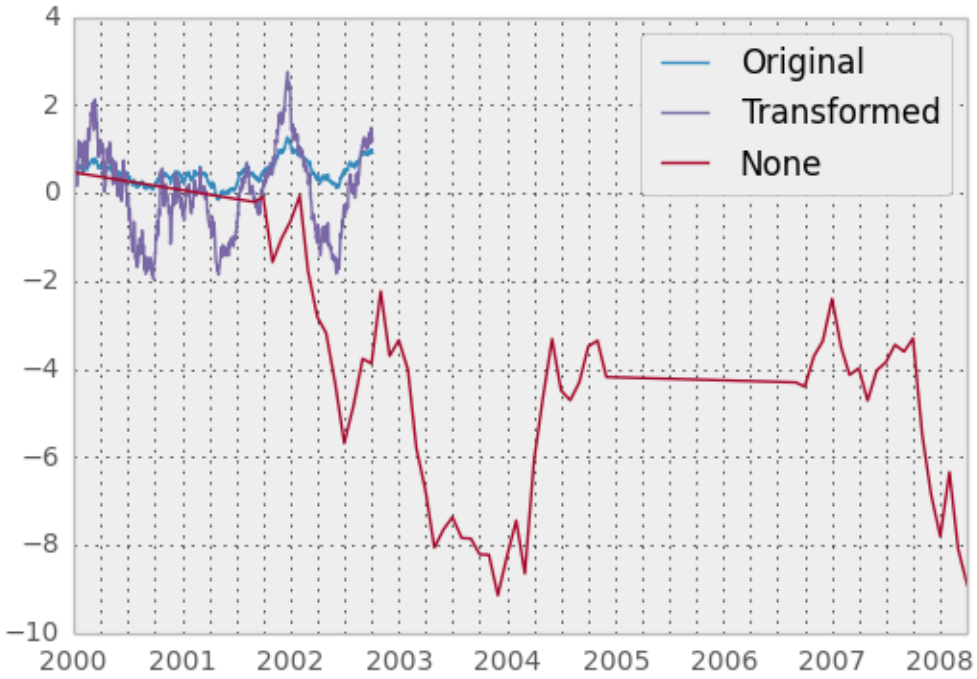
```
In [33]: df['one'].dropna()
Series([], dtype: float64)
```

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

11.4.3 Interpolation

A linear **interpolate** method has been implemented on Series. The default interpolation assumes equally spaced points.

```
In [34]: ts.count()
61

In [35]: ts.head()

2000-01-31    0.469112
2000-02-29         NaN
2000-03-31         NaN
2000-04-28         NaN
2000-05-31         NaN
Freq: BM, dtype: float64

In [36]: ts.interpolate().count()
100

In [37]: ts.interpolate().head()

2000-01-31    0.469112
2000-02-29    0.435428
2000-03-31    0.401743
2000-04-28    0.368059
2000-05-31    0.334374
Freq: BM, dtype: float64

In [38]: ts.interpolate().plot()
<matplotlib.axes.AxesSubplot at 0x6eb9810>
```



Index aware interpolation is available via the `method` keyword:

```
In [39]: ts

2000-01-31    0.469112
2000-02-29         NaN
2002-07-31   -5.689738
2005-01-31         NaN
2008-04-30   -8.916232
dtype: float64

In [40]: ts.interpolate()

2000-01-31    0.469112
2000-02-29   -2.610313
2002-07-31   -5.689738
2005-01-31   -7.302985
2008-04-30   -8.916232
dtype: float64

In [41]: ts.interpolate(method='time')

2000-01-31    0.469112
2000-02-29    0.273272
2002-07-31   -5.689738
2005-01-31   -7.095568
2008-04-30   -8.916232
dtype: float64
```

For a floating-point index, use `method='values'`:

```
In [42]: ser

0      0
```

```
1    NaN
10    10
dtype: float64

In [43]: ser.interpolate()

0      0
1      5
10    10
dtype: float64

In [44]: ser.interpolate(method='values')

0      0
1      1
10    10
dtype: float64
```

11.4.4 Replacing Generic Values

Often times we want to replace arbitrary values with other values. New in v0.8 is the `replace` method in Series/DataFrame that provides an efficient yet flexible way to perform such replacements.

For a Series, you can replace a single value or a list of values by another value:

```
In [45]: ser = Series([0., 1., 2., 3., 4.])

In [46]: ser.replace(0, 5)

0    5
1    1
2    2
3    3
4    4
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [47]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])

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

You can also specify a mapping dict:

```
In [48]: ser.replace({0: 10, 1: 100})

0     10
1    100
2      2
3      3
4      4
dtype: float64
```

For a DataFrame, you can specify individual values by column:

```
In [49]: df = DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [50]: df.replace({'a': 0, 'b': 5}, 100)

     a    b
0  100  100
1    1    6
2    2    7
3    3    8
4    4    9
```

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

```
In [51]: ser.replace([1, 2, 3], method='pad')

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

11.4.5 String/Regular Expression Replacement

Note: Python strings prefixed with the `r` character such as `r'hello world'` are so-called "raw" strings. They have different semantics regarding backslashes than strings without this prefix. Backslashes in raw strings will be interpreted as an escaped backslash, e.g., `r'\' == '\\'`. You should read about them if this is unclear.

Replace the '.' with `nan` (str -> str)

```
In [52]: d = {'a': range(4), 'b': list('ab..'), 'c': ['a', 'b', nan, 'd']}

In [53]: df = DataFrame(d)

In [54]: df.replace('.', nan)

   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Now do it with a regular expression that removes surrounding whitespace (regex -> regex)

```
In [55]: df.replace(r'\s*\.\s*', nan, regex=True)

   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Replace a few different values (list -> list)

```
In [56]: df.replace(['a', '.'], ['b', nan])
```

```
   a    b    c
0  0    b    b
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

list of regex -> list of regex

```
In [57]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
```

```
   a       b       c
0  0  {stuff  {stuff
1  1       b       b
2  2     dot     NaN
3  3     dot       d
```

Only search in column `'b'` (dict -> dict)

```
In [58]: df.replace({'b': '.'}, {'b': nan})
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict)

```
In [59]: df.replace({'b': r'\s*\.\s*'}, {'b': nan}, regex=True)
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can pass nested dictionaries of regular expressions that use `regex=True`

```
In [60]: df.replace({'b': {'b': r''}}, regex=True)
```

```
   a  b    c
0  0  a    a
1  1       b
2  2  .  NaN
3  3  .    d
```

or you can pass the nested dictionary like so

```
In [61]: df.replace(regex={'b': {r'\s*\.\s*': nan}})
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well

```
In [62]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
```

```
   a    b    c
0  0    a    a
1  1    b    b
2  2  .ty  NaN
3  3  .ty    d
```

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex)

```
In [63]: df.replace([r'\s*\.\s*', r'a|b'], nan, regex=True)
```

```
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicity by name or `regex` must be a nested dictionary. The previous example, in this case, would then be

```
In [64]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=nan)
```

```
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

This can be convenient if you do not want to pass `regex=True` every time you want to use a regular expression.

Note: Anywhere in the above `replace` examples that you see a regular expression a compiled regular expression is valid as well.

11.4.6 Numeric Replacement

Similiar to `DataFrame.fillna`

```
In [65]: df = DataFrame(randn(10, 2))

In [66]: df[rand(df.shape[0]) > 0.5] = 1.5

In [67]: df.replace(1.5, nan)
```

```
          0         1
0       NaN       NaN
1       NaN       NaN
2  2.396780  0.014871
3       NaN       NaN
4       NaN       NaN
5       NaN       NaN
6       NaN       NaN
7  0.084844  0.432390
```

```
8  1.519970 -0.493662
9       NaN       NaN
```

Replacing more than one value via lists works as well

```
In [68]: df00 = df.values[0, 0]

In [69]: df.replace([1.5, df00], [nan, 'a'])

             0           1
0            a           a
1            a           a
2      2.39678  0.01487095
3            a           a
4            a           a
5            a           a
6            a           a
7  0.08484421   0.4323898
8     1.51997  -0.4936621
9            a           a

In [70]: df[1].dtype
dtype('float64')
```

You can also operate on the DataFrame in place

```
In [71]: df.replace(1.5, nan, inplace=True)
```

11.5 Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we've established some "casting rules" when reindexing will cause missing data to be introduced into, say, a Series or DataFrame. Here they are:

data type	Cast to
integer	float
boolean	object
float	no cast
object	no cast

For example:

```
In [72]: s = Series(randn(5), index=[0, 2, 4, 6, 7])

In [73]: s > 0

0    False
2     True
4     True
6     True
7    False
dtype: bool

In [74]: (s > 0).dtype
dtype('bool')

In [75]: crit = (s > 0).reindex(range(8))
```

```
In [76]: crit

0    False
1      NaN
2     True
3      NaN
4     True
5      NaN
6     True
7    False
dtype: object

In [77]: crit.dtype
dtype('O')
```

Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

```
In [78]: reindexed = s.reindex(range(8)).fillna(0)

In [79]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-79-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]
/home/docbuild/CI/pandas/pandas/core/series.pyc in __getitem__(self, key)
    636         # special handling of boolean data with NAs stored in object
    637         # arrays. Since we can't represent NA with dtype=bool
--> 638         if _is_bool_indexer(key):
    639             key = _check_bool_indexer(self.index, key)
    640
/home/docbuild/CI/pandas/pandas/core/common.pyc in _is_bool_indexer(key)
   1236         if not lib.is_bool_array(key):
   1237             if isnull(key).any():
-> 1238                 raise ValueError('cannot index with vector containing '
   1239                                  'NA / NaN values')
   1240             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 [80]: reindexed[crit.fillna(False)]

2    1.063327
4    1.266143
6    0.299368
dtype: float64

In [81]: reindexed[crit.fillna(True)]

1    0.000000
2    1.063327
3    0.000000
4    1.266143
5    0.000000
6    0.299368
dtype: float64
```

CHAPTER

TWELVE

GROUP BY: SPLIT-APPLY-COMBINE

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

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

Of these, the split step is the most straightforward. In fact, in many situations you may wish to split the data set into groups and do something with those groups yourself. In the apply step, we might wish to one of the following:

- **Aggregation**: computing a summary statistic (or statistics) about each group. Some examples:
  - Compute group sums or means
  - Compute group sizes / counts
- **Transformation**: perform some group-specific computations and return a like-indexed. Some examples:
  - Standardizing data (zscore) within group
  - Filling NAs within groups with a value derived from each group
- **Filtration**: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  - Discarding data that belongs to groups with only a few members
  - Filtering out data based on the group sum or mean
- Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn't fit into either of the above two categories

Since the set of object instance method on pandas data structures are generally rich and expressive, we often simply want to invoke, say, a DataFrame function on each group. The name GroupBy should be quite familiar to those who have used a SQL-based tool (or `itertools`), in which you can write code like:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

We aim to make operations like this natural and easy to express using pandas. We'll address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the *cookbook* for some advanced strategies

12.1 Splitting an object into groups

pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you do the following:

```
>>> grouped = obj.groupby(key)
>>> grouped = obj.groupby(key, axis=1)
>>> grouped = obj.groupby([key1, key2])
```

The mapping can be specified many different ways:

- A Python function, to be called on each of the axis labels
- A list or NumPy array of the same length as the selected axis
- A dict or Series, providing a `label -> group name` mapping
- For DataFrame objects, a string indicating a column to be used to group. Of course `df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`, but it makes life simpler
- A list of any of the above things

Collectively we refer to the grouping objects as the **keys**. For example, consider the following DataFrame:

```
In [1]: df = DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
   ...:                        'foo', 'bar', 'foo', 'foo'],
   ...:                 'B' : ['one', 'one', 'two', 'three',
   ...:                        'two', 'two', 'one', 'three'],
   ...:                 'C' : randn(8), 'D' : randn(8)})
   ...:

In [2]: df
```

```
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

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

```
In [3]: grouped = df.groupby('A')

In [4]: grouped = df.groupby(['A', 'B'])
```

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

```
In [5]: def get_letter_type(letter):
   ...:     if letter.lower() in 'aeiou':
   ...:         return 'vowel'
   ...:     else:
   ...:         return 'consonant'
   ...:

In [6]: grouped = df.groupby(get_letter_type, axis=1)
```

Starting with 0.8, pandas Index objects now supports duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```
In [7]: lst = [1, 2, 3, 1, 2, 3]

In [8]: s = Series([1, 2, 3, 10, 20, 30], lst)

In [9]: grouped = s.groupby(level=0)

In [10]: grouped.first()

1    1
2    2
3    3
dtype: int64

In [11]: grouped.last()

1    10
2    20
3    30
dtype: int64

In [12]: grouped.sum()

1    11
2    22
3    33
dtype: int64
```

Note that **no splitting occurs** until it's needed. Creating the GroupBy object only verifies that you've passed a valid mapping.

Note: Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though can't be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.

12.1.1 GroupBy object attributes

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```
In [13]: df.groupby('A').groups
{'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}

In [14]: df.groupby(get_letter_type, axis=1).groups
{'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [15]: grouped = df.groupby(['A', 'B'])

In [16]: grouped.groups

{('bar', 'one'): [1],
```

```
 ('bar', 'three'): [3],
 ('bar', 'two'): [5],
 ('foo', 'one'): [0, 6],
 ('foo', 'three'): [7],
 ('foo', 'two'): [2, 4]}

In [17]: len(grouped)
6
```

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

```
In [18]: df2 = DataFrame({'X' : ['B', 'B', 'A', 'A'], 'Y' : [1, 2, 3, 4]})

In [19]: df2.groupby(['X'], sort=True).sum()

   Y
X
A  7
B  3

In [20]: df2.groupby(['X'], sort=False).sum()

   Y
X
B  3
A  7
```

12.1.2 GroupBy with MultiIndex

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

```
In [21]: s

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
dtype: float64

In [22]: grouped = s.groupby(level=0)

In [23]: grouped.sum()

first
bar      0.142048
baz     -0.811169
foo     -0.560041
qux     -0.953439
dtype: float64
```

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

```
In [24]: s.groupby(level='second').sum()

second
one      -2.300857
two       0.118256
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

```
In [25]: s.sum(level='second')

second
one      -2.300857
two       0.118256
dtype: float64
```

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

```
In [26]: s

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    bop     one     -1.344312
               two      0.844885
dtype: float64

In [27]: s.groupby(level=['first','second']).sum()

first  second
bar    doo       0.981751
baz    bee      -2.754270
foo    bop      -1.528539
qux    bop      -0.499427
dtype: float64
```

More on the `sum` function and aggregation later.

12.1.3 DataFrame column selection in GroupBy

Once you have created the GroupBy object from a DataFrame, for example, you might want to do something different for each of the columns. Thus, using `[]` similar to getting a column from a DataFrame, you can do:

```
In [28]: grouped = df.groupby(['A'])

In [29]: grouped_C = grouped['C']

In [30]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [31]: df['C'].groupby(df['A'])
<pandas.core.groupby.SeriesGroupBy object at 0x588f490>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

12.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby`:

```
In [32]: grouped = df.groupby('A')

In [33]: 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 [34]: 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:`.

12.3 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. An obvious one is aggregation via the `aggregate` or equivalently `agg` method:

```
In [35]: grouped = df.groupby('A')

In [36]: grouped.aggregate(np.sum)

            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938

In [37]: grouped = df.groupby(['A', 'B'])

In [38]: grouped.aggregate(np.sum)

                  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 [39]: grouped = df.groupby(['A', 'B'], as_index=False)

In [40]: grouped.aggregate(np.sum)

     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 [41]: df.groupby('A', as_index=False).sum()

     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 [42]: df.groupby(['A', 'B']).sum().reset_index()

     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
```

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a Series whose index are the group names and whose values are the sizes of each group.

```
In [43]: grouped.size()

A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

12.3.1 Applying multiple functions at once

With grouped Series you can also pass a list or dict of functions to do aggregation with, outputting a DataFrame:

```
In [44]: grouped = df.groupby('A')

In [45]: grouped['C'].agg([np.sum, np.mean, np.std])

          sum      mean       std
A
bar -1.591710 -0.530570  0.526860
foo -0.752861 -0.150572  1.113308
```

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 [46]: grouped['D'].agg({'result1' : np.sum,
   ....:                   'result2' : np.mean})
   ....:

      result2   result1
A
bar -0.579846 -1.739537
foo -0.280588 -1.402938
```

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 [47]: grouped.agg([np.sum, np.mean, np.std])

            C                              D
          sum      mean       std       sum      mean       std
A
bar -1.591710 -0.530570  0.526860 -1.739537 -0.579846  1.591986
foo -0.752861 -0.150572  1.113308 -1.402938 -0.280588  0.753219
```

Passing a dict of functions has different behavior by default, see the next section.

12.3.2 Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```
In [48]: grouped.agg({'C' : np.sum,
   ....:              'D' : lambda x: np.std(x, ddof=1)})
   ....:

            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 [49]: grouped.agg({'C' : 'sum', 'D' : 'std'})

            C         D
A
bar -1.591710  1.591986
foo -0.752861  0.753219
```

12.3.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, and `std`, have optimized Cython implementations:

```
In [50]: df.groupby('A').sum()

            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938

In [51]: df.groupby(['A', 'B']).mean()

                  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).

12.4 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. Thus, the passed transform function should return a result that is the same size as the group chunk. For example, suppose we wished to standardize the data within each group:

```
In [52]: index = date_range('10/1/1999', periods=1100)

In [53]: ts = Series(np.random.normal(0.5, 2, 1100), index)

In [54]: ts = rolling_mean(ts, 100, 100).dropna()

In [55]: ts.head()

2000-01-08    0.536925
2000-01-09    0.494448
2000-01-10    0.496114
2000-01-11    0.443475
2000-01-12    0.474744
Freq: D, dtype: float64

In [56]: ts.tail()

2002-09-30    0.978859
2002-10-01    0.994704
2002-10-02    0.953789
2002-10-03    0.932345
2002-10-04    0.915581
Freq: D, dtype: float64

In [57]: key = lambda x: x.year

In [58]: zscore = lambda x: (x - x.mean()) / x.std()

In [59]: transformed = ts.groupby(key).transform(zscore)
```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
# Original Data
In [60]: grouped = ts.groupby(key)

In [61]: grouped.mean()

2000    0.416344
2001    0.416987
2002    0.599380
dtype: float64

In [62]: grouped.std()

2000    0.174755
2001    0.309640
2002    0.266172
dtype: float64

# Transformed Data
In [63]: grouped_trans = transformed.groupby(key)

In [64]: grouped_trans.mean()

2000   -3.122696e-16
2001   -2.688869e-16
2002   -1.499001e-16
```

```
dtype: float64
```

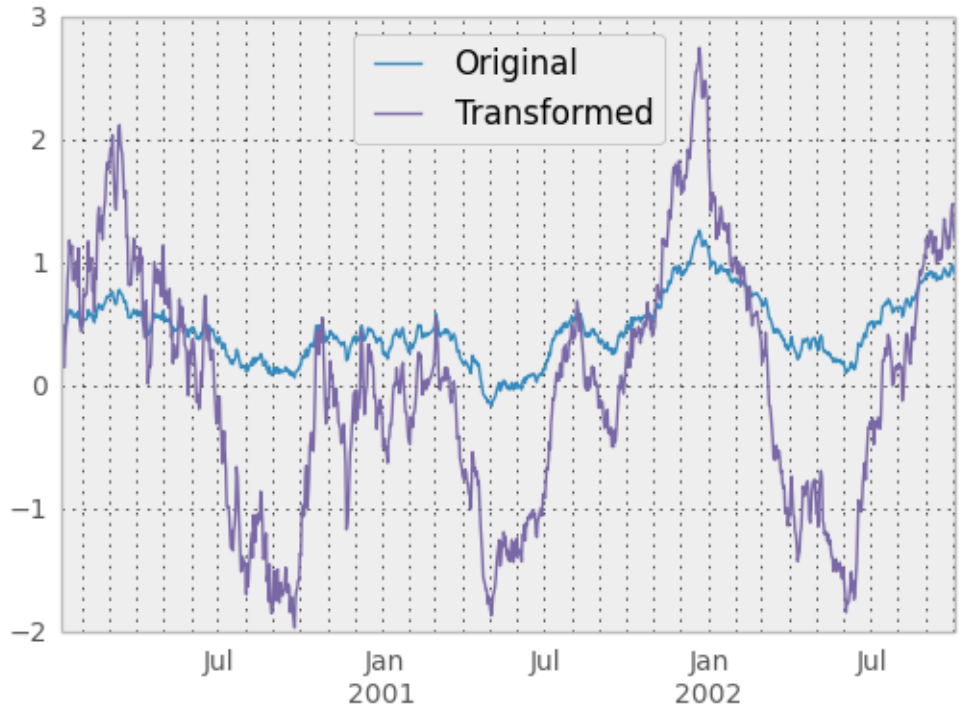
```
In [65]: grouped_trans.std()
```

```
2000    1
2001    1
2002    1
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [66]: compare = DataFrame({'Original': ts, 'Transformed': transformed})
```

```
In [67]: compare.plot()
<matplotlib.axes.AxesSubplot at 0x6eb9810>
```



Another common data transform is to replace missing data with the group mean.

```
In [68]: data_df
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 0 to 999
Data columns (total 3 columns):
A    908  non-null values
B    953  non-null values
C    820  non-null values
dtypes: float64(3)
```

```
In [69]: countries = np.array(['US', 'UK', 'GR', 'JP'])
```

```
In [70]: key = countries[np.random.randint(0, 4, 1000)]
```

```
In [71]: grouped = data_df.groupby(key)
```

```
# Non-NA count in each group
In [72]: grouped.count()

      A    B    C
GR  219  223  194
JP  238  250  211
UK  228  239  213
US  223  241  202

In [73]: f = lambda x: x.fillna(x.mean())

In [74]: transformed = grouped.transform(f)
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.

```
In [75]: grouped_trans = transformed.groupby(key)

In [76]: grouped.mean() # original group means

           A         B         C
GR  0.093655 -0.004978 -0.049883
JP -0.067605  0.025828  0.006752
UK -0.054246  0.031742  0.068974
US  0.084334 -0.013433  0.056589

In [77]: grouped_trans.mean() # transformation did not change group means

           A         B         C
GR  0.093655 -0.004978 -0.049883
JP -0.067605  0.025828  0.006752
UK -0.054246  0.031742  0.068974
US  0.084334 -0.013433  0.056589

In [78]: grouped.count() # original has some missing data points

      A    B    C
GR  219  223  194
JP  238  250  211
UK  228  239  213
US  223  241  202

In [79]: grouped_trans.count() # counts after transformation

      A    B    C
GR  234  234  234
JP  264  264  264
UK  251  251  251
US  251  251  251

In [80]: grouped_trans.size() # Verify non-NA count equals group size

GR    234
JP    264
UK    251
US    251
dtype: int64
```

12.5 Filtration

New in version 0.12. The `filter` method returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [81]: sf = Series([1, 1, 2, 3, 3, 3])

In [82]: sf.groupby(sf).filter(lambda x: x.sum() > 2)

3    3
4    3
5    3
dtype: int64
```

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

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

```
In [83]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [84]: dff.groupby('B').filter(lambda x: len(x) > 2)

   A  B
2  2  b
3  3  b
4  4  b
5  5  b
```

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

```
In [85]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)

    A    B
0 NaN  NaN
1 NaN  NaN
2   2    b
3   3    b
4   4    b
5   5    b
6 NaN  NaN
7 NaN  NaN
```

12.6 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```
In [86]: grouped = df.groupby('A')

In [87]: grouped.agg(lambda x: x.std())

      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 [88]: grouped.std()
```

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

In [90]: tsdf.ix[::2] = np.nan

In [91]: grouped = tsdf.groupby(lambda x: x.year)

In [92]: grouped.fillna(method='pad')
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1000 entries, 2000-01-01 00:00:00 to 2002-09-26 00:00:00
Freq: D
Data columns (total 3 columns):
A    998  non-null values
B    998  non-null values
C    998  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.

12.7 Flexible `apply`

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want GroupBy to infer how to combine the results. For these, use the `apply` function, which can be substituted for both `aggregate` and `transform` in many standard use cases. However, `apply` can handle some exceptional use cases, for example:

```
In [93]: df
```

```
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

```
In [94]: grouped = df.groupby('A')

# could also just call .describe()
In [95]: grouped['C'].apply(lambda x: x.describe())

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
dtype: float64
```

The dimension of the returned result can also change:

```
In [96]: grouped = df.groupby('A')['C']

In [97]: def f(group):
   ....:     return DataFrame({'original' : group,
   ....:                       'demeaned' : group - group.mean()})
   ....:

In [98]: grouped.apply(f)

   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
```

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

```
In [99]: def f(x):
   ....:         return Series([ x, x**2 ], index = ['x', 'x^s'])
   ....:

In [100]: s = Series(np.random.rand(5))

In [101]: s

0    0.785887
1    0.498525
2    0.933703
```

```
3    0.154106
4    0.271779
dtype: float64
```

```
In [102]: s.apply(f)
```

```
          x       x^s
0  0.785887  0.617619
1  0.498525  0.248528
2  0.933703  0.871801
3  0.154106  0.023749
4  0.271779  0.073864
```

12.8 Other useful features

12.8.1 Automatic exclusion of "nuisance" columns

Again consider the example DataFrame we've been looking at:

```
In [103]: df
```

```
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

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 [104]: df.groupby('A').std()
```

```
            C         D
A
bar  0.526860  1.591986
foo  1.113308  0.753219
```

12.8.2 NA group handling

If there are any NaN values in the grouping key, these will be automatically excluded. So there will never be an "NA group". This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

12.8.3 Grouping with ordered factors

Categorical variables represented as instance of pandas's `Categorical` class can be used as group keys. If so, the order of the levels will be preserved:

```
In [105]: data = Series(np.random.randn(100))

In [106]: factor = qcut(data, [0, .25, .5, .75, 1.])

In [107]: data.groupby(factor).mean()

[-3.469, -0.737]   -1.269581
(-0.737, 0.214]    -0.216269
(0.214, 1.0572]     0.680402
(1.0572, 3.0762]    1.629338
dtype: float64
```

CHAPTER

THIRTEEN

MERGE, JOIN, AND CONCATENATE

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

13.1 Concatenating objects

The `concat` function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say "if any" because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

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

In [2]: df

          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914

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

In [4]: concatenated = concat(pieces)

In [5]: concatenated

          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
```

```
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```

Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of "what to do with the other axes":

```
concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
       keys=None, levels=None, names=None, verify_integrity=False)
```

- `objs`: list or dict of Series, DataFrame, or Panel objects. If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below)
- `axis`: {0, 1, ...}, default 0. The axis to concatenate along
- `join`: {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection
- `join_axes`: list of Index objects. Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic
- `keys`: sequence, default None. Construct hierarchical index using the passed keys as the outermost level If multiple levels passed, should contain tuples.
- `levels` : list of sequences, default None. If keys passed, specific levels to use for the resulting MultiIndex. Otherwise they will be inferred from the keys
- `names`: list, default None. Names for the levels in the resulting hierarchical index
- `verify_integrity`: boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation
- `ignore_index` : boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information.

Without a little bit of context and example many of these arguments don't make much sense. Let's take the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the `keys` argument:

```
In [6]: concatenated = concat(pieces, keys=['first', 'second', 'third'])

In [7]: concatenated
```

```
                  0         1         2         3
first  0  0.469112 -0.282863 -1.509059 -1.135632
       1  1.212112 -0.173215  0.119209 -1.044236
       2 -0.861849 -2.104569 -0.494929  1.071804
second 3  0.721555 -0.706771 -1.039575  0.271860
       4 -0.424972  0.567020  0.276232 -1.087401
       5 -0.673690  0.113648 -1.478427  0.524988
       6  0.404705  0.577046 -1.715002 -1.039268
third  7 -0.370647 -1.157892 -1.344312  0.844885
       8  1.075770 -0.109050  1.643563 -1.469388
       9  0.357021 -0.674600 -1.776904 -0.968914
```

As you can see (if you've read the rest of the documentation), the resulting object's index has a *hierarchical index*. This means that we can now do stuff like select out each chunk by key:

```
In [8]: concatenated.ix['second']
```

```
          0         1         2         3
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

13.1.1 Set logic on the other axes

When gluing together multiple DataFrames (or Panels or...), for example, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in three ways:

- Take the (sorted) union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.
- Use a specific index (in the case of DataFrame) or indexes (in the case of Panel or future higher dimensional objects), i.e. the `join_axes` argument

Here is a example of each of these methods. First, the default `join='outer'` behavior:

```
In [9]: from pandas.util.testing import rands

In [10]: df = DataFrame(np.random.randn(10, 4), columns=['a', 'b', 'c', 'd'],
   ....:                index=[rands(5) for _ in xrange(10)])
   ....:

In [11]: df
```

```
              a         b         c         d
kpw8b -1.294524  0.413738  0.276662 -0.472035
4Teki -0.013960 -0.362543 -0.006154 -0.923061
QJBdT  0.895717  0.805244 -1.206412  2.565646
hNBQ3  1.431256  1.340309 -1.170299 -0.226169
6uKmx  0.410835  0.813850  0.132003 -0.827317
UQC83 -0.076467 -1.187678  1.130127 -1.436737
IYhSl -1.413681  1.607920  1.024180  0.569605
EU2TB  0.875906 -2.211372  0.974466 -2.006747
twLhS -0.410001 -0.078638  0.545952 -1.219217
DJeCP -1.226825  0.769804 -1.281247 -0.727707
```

```
In [12]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1)
   ....:
```

```
              a         b         c         d
4Teki -0.013960 -0.362543       NaN       NaN
6uKmx  0.410835  0.813850  0.132003 -0.827317
DJeCP       NaN       NaN       NaN -0.727707
EU2TB       NaN       NaN  0.974466 -2.006747
IYhSl -1.413681  1.607920  1.024180  0.569605
QJBdT  0.895717  0.805244 -1.206412       NaN
UQC83 -0.076467 -1.187678  1.130127 -1.436737
hNBQ3  1.431256  1.340309 -1.170299 -0.226169
```

```
kpw8b -1.294524  0.413738       NaN       NaN
twLhS       NaN       NaN       NaN -1.219217
```

Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```
In [13]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1, join='inner')
   ....:

              a         b         c         d
hNBQ3  1.431256  1.340309 -1.170299 -0.226169
6uKmx  0.410835  0.813850  0.132003 -0.827317
UQC83 -0.076467 -1.187678  1.130127 -1.436737
IYhSl -1.413681  1.607920  1.024180  0.569605
```

Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

```
In [14]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
   ....:

              a         b         c         d
kpw8b -1.294524  0.413738       NaN       NaN
4Teki -0.013960 -0.362543       NaN       NaN
QJBdT  0.895717  0.805244 -1.206412       NaN
hNBQ3  1.431256  1.340309 -1.170299 -0.226169
6uKmx  0.410835  0.813850  0.132003 -0.827317
UQC83 -0.076467 -1.187678  1.130127 -1.436737
IYhSl -1.413681  1.607920  1.024180  0.569605
EU2TB       NaN       NaN  0.974466 -2.006747
twLhS       NaN       NaN       NaN -1.219217
DJeCP       NaN       NaN       NaN -0.727707
```

13.1.2 Concatenating using `append`

A useful shortcut to `concat` are the `append` instance methods on Series and DataFrame. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

```
In [15]: s = Series(randn(10), index=np.arange(10))

In [16]: s1 = s[:5] # note we're slicing with labels here, so 5 is included

In [17]: s2 = s[6:]

In [18]: s1.append(s2)

0   -0.121306
1   -0.097883
2    0.695775
3    0.341734
4    0.959726
6   -0.619976
7    0.149748
8   -0.732339
9    0.687738
dtype: float64
```

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

```
In [19]: df = DataFrame(randn(6, 4), index=date_range('1/1/2000', periods=6),
   ....:                columns=['A', 'B', 'C', 'D'])
   ....:

In [20]: df1 = df.ix[:3]

In [21]: df2 = df.ix[3:, :3]

In [22]: df1

                   A         B         C         D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232

In [23]: df2

                   A         B         C
2000-01-04  0.690579  0.995761  2.396780
2000-01-05  3.357427 -0.317441 -1.236269
2000-01-06 -0.487602 -0.082240 -2.182937

In [24]: df1.append(df2)

                   A         B         C         D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
2000-01-04  0.690579  0.995761  2.396780       NaN
2000-01-05  3.357427 -0.317441 -1.236269       NaN
2000-01-06 -0.487602 -0.082240 -2.182937       NaN
```

`append` may take multiple objects to concatenate:

```
In [25]: df1 = df.ix[:2]

In [26]: df2 = df.ix[2:4]

In [27]: df3 = df.ix[4:]

In [28]: df1.append([df2,df3])

                   A         B         C         D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
2000-01-04  0.690579  0.995761  2.396780  0.014871
2000-01-05  3.357427 -0.317441 -1.236269  0.896171
2000-01-06 -0.487602 -0.082240 -2.182937  0.380396
```

Note: Unlike *list.append* method, which appends to the original list and returns nothing, `append` here **does not** modify `df1` and returns its copy with `df2` appended.

13.1.3 Ignoring indexes on the concatenation axis

For DataFrames which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes:

```
In [29]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])

In [30]: df2 = DataFrame(randn(3, 4), columns=['A', 'B', 'C', 'D'])

In [31]: df1
```

```
          A         B         C         D
0  0.084844  0.432390  1.519970 -0.493662
1  0.600178  0.274230  0.132885 -0.023688
2  2.410179  1.450520  0.206053 -0.251905
3 -2.213588  1.063327  1.266143  0.299368
4 -0.863838  0.408204 -1.048089 -0.025747
5 -0.988387  0.094055  1.262731  1.289997
```

```
In [32]: df2
```

```
          A         B         C         D
0  0.082423 -0.055758  0.536580 -0.489682
1  0.369374 -0.034571 -2.484478 -0.281461
2  0.030711  0.109121  1.126203 -0.977349
```

To do this, use the `ignore_index` argument:

```
In [33]: concat([df1, df2], ignore_index=True)
```

```
          A         B         C         D
0  0.084844  0.432390  1.519970 -0.493662
1  0.600178  0.274230  0.132885 -0.023688
2  2.410179  1.450520  0.206053 -0.251905
3 -2.213588  1.063327  1.266143  0.299368
4 -0.863838  0.408204 -1.048089 -0.025747
5 -0.988387  0.094055  1.262731  1.289997
6  0.082423 -0.055758  0.536580 -0.489682
7  0.369374 -0.034571 -2.484478 -0.281461
8  0.030711  0.109121  1.126203 -0.977349
```

This is also a valid argument to `DataFrame.append`:

```
In [34]: df1.append(df2, ignore_index=True)
```

```
          A         B         C         D
0  0.084844  0.432390  1.519970 -0.493662
1  0.600178  0.274230  0.132885 -0.023688
2  2.410179  1.450520  0.206053 -0.251905
3 -2.213588  1.063327  1.266143  0.299368
4 -0.863838  0.408204 -1.048089 -0.025747
5 -0.988387  0.094055  1.262731  1.289997
6  0.082423 -0.055758  0.536580 -0.489682
7  0.369374 -0.034571 -2.484478 -0.281461
8  0.030711  0.109121  1.126203 -0.977349
```

13.1.4 More concatenating with group keys

Let's consider a variation on the first example presented:

```
In [35]: df = DataFrame(np.random.randn(10, 4))

In [36]: df

          0         1         2         3
0  1.474071 -0.064034 -1.282782  0.781836
1 -1.071357  0.441153  2.353925  0.583787
2  0.221471 -0.744471  0.758527  1.729689
3 -0.964980 -0.845696 -1.340896  1.846883
4 -1.328865  1.682706 -1.717693  0.888782
5  0.228440  0.901805  1.171216  0.520260
6 -1.197071 -1.066969 -0.303421 -0.858447
7  0.306996 -0.028665  0.384316  1.574159
8  1.588931  0.476720  0.473424 -0.242861
9 -0.014805 -0.284319  0.650776 -1.461665

# break it into pieces
In [37]: pieces = [df.ix[:, [0, 1]], df.ix[:, [2]], df.ix[:, [3]]]

In [38]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'])

In [39]: result

        one                   two     three
          0         1         2         3
0  1.474071 -0.064034 -1.282782  0.781836
1 -1.071357  0.441153  2.353925  0.583787
2  0.221471 -0.744471  0.758527  1.729689
3 -0.964980 -0.845696 -1.340896  1.846883
4 -1.328865  1.682706 -1.717693  0.888782
5  0.228440  0.901805  1.171216  0.520260
6 -1.197071 -1.066969 -0.303421 -0.858447
7  0.306996 -0.028665  0.384316  1.574159
8  1.588931  0.476720  0.473424 -0.242861
9 -0.014805 -0.284319  0.650776 -1.461665
```

You can also pass a dict to `concat` in which case the dict keys will be used for the `keys` argument (unless other keys are specified):

```
In [40]: pieces = {'one': df.ix[:, [0, 1]],
   ....:           'two': df.ix[:, [2]],
   ....:           'three': df.ix[:, [3]]}
   ....:

In [41]: concat(pieces, axis=1)

        one               three       two
          0         1         3         2
0  1.474071 -0.064034  0.781836 -1.282782
1 -1.071357  0.441153  0.583787  2.353925
2  0.221471 -0.744471  1.729689  0.758527
3 -0.964980 -0.845696  1.846883 -1.340896
4 -1.328865  1.682706  0.888782 -1.717693
5  0.228440  0.901805  0.520260  1.171216
6 -1.197071 -1.066969 -0.858447 -0.303421
```

```
7  0.306996 -0.028665  1.574159  0.384316
8  1.588931  0.476720 -0.242861  0.473424
9 -0.014805 -0.284319 -1.461665  0.650776
```

```
In [42]: concat(pieces, keys=['three', 'two'])
```

```
                2         3
three 0       NaN  0.781836
      1       NaN  0.583787
      2       NaN  1.729689
      3       NaN  1.846883
      4       NaN  0.888782
      5       NaN  0.520260
      6       NaN -0.858447
      7       NaN  1.574159
      8       NaN -0.242861
      9       NaN -1.461665
two   0 -1.282782       NaN
      1  2.353925       NaN
      2  0.758527       NaN
      3 -1.340896       NaN
      4 -1.717693       NaN
      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 [43]: result.columns.levels
```

```
[Index([u'one', u'two', u'three'], dtype=object),
 Int64Index([0, 1, 2, 3], dtype=int64)]
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

```
In [44]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'],
   ....:                 levels=[['three', 'two', 'one', 'zero']],
   ....:                 names=['group_key'])
   ....:
```

```
In [45]: result
```

```
group_key       one                   two     three
                  0         1         2         3
0          1.474071 -0.064034 -1.282782  0.781836
1         -1.071357  0.441153  2.353925  0.583787
2          0.221471 -0.744471  0.758527  1.729689
3         -0.964980 -0.845696 -1.340896  1.846883
4         -1.328865  1.682706 -1.717693  0.888782
5          0.228440  0.901805  1.171216  0.520260
6         -1.197071 -1.066969 -0.303421 -0.858447
7          0.306996 -0.028665  0.384316  1.574159
8          1.588931  0.476720  0.473424 -0.242861
9         -0.014805 -0.284319  0.650776 -1.461665
```

```
In [46]: result.columns.levels
```

```
[Index([u'three', u'two', u'one', u'zero'], dtype=object),
 Int64Index([0, 1, 2, 3], dtype=int64)]
```

Yes, this is fairly esoteric, but is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

13.1.5 Appending rows to a DataFrame

While not especially efficient (since a new object must be created), you can append a single row to a DataFrame by passing a Series or dict to `append`, which returns a new DataFrame as above.

```
In [47]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [48]: df

          A         B         C         D
0 -1.137707 -0.891060 -0.693921  1.613616
1  0.464000  0.227371 -0.496922  0.306389
2 -2.290613 -1.134623 -1.561819 -0.260838
3  0.281957  1.523962 -0.902937  0.068159
4 -0.057873 -0.368204 -1.144073  0.861209
5  0.800193  0.782098 -1.069094 -1.099248
6  0.255269  0.009750  0.661084  0.379319
7 -0.008434  1.952541 -1.056652  0.533946

In [49]: s = df.xs(3)

In [50]: df.append(s, ignore_index=True)

          A         B         C         D
0 -1.137707 -0.891060 -0.693921  1.613616
1  0.464000  0.227371 -0.496922  0.306389
2 -2.290613 -1.134623 -1.561819 -0.260838
3  0.281957  1.523962 -0.902937  0.068159
4 -0.057873 -0.368204 -1.144073  0.861209
5  0.800193  0.782098 -1.069094 -1.099248
6  0.255269  0.009750  0.661084  0.379319
7 -0.008434  1.952541 -1.056652  0.533946
8  0.281957  1.523962 -0.902937  0.068159
```

You should use `ignore_index` with this method to instruct DataFrame to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed DataFrame and append or concatenate those objects.

You can also pass a list of dicts or Series:

```
In [51]: df = DataFrame(np.random.randn(5, 4),
   ....:                columns=['foo', 'bar', 'baz', 'qux'])
   ....:

In [52]: dicts = [{'foo': 1, 'bar': 2, 'baz': 3, 'peekaboo': 4},
   ....:          {'foo': 5, 'bar': 6, 'baz': 7, 'peekaboo': 8}]
   ....:

In [53]: result = df.append(dicts, ignore_index=True)

In [54]: result

        bar       baz       foo  peekaboo       qux
```

```
0  0.040403 -0.507516 -1.226970       NaN -0.230096
1 -1.934370 -1.652499  0.394500       NaN  1.488753
2  0.576897  1.146000 -0.896484       NaN  1.487349
3  2.121453  0.597701  0.604603       NaN  0.563700
4 -1.057909  1.375020  0.967661       NaN -0.928797
5  2.000000  3.000000  1.000000         4       NaN
6  6.000000  7.000000  5.000000         8       NaN
```

13.2 Database-style DataFrame joining/merging

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R). The reason for this is careful algorithmic design and internal layout of the data in DataFrame.

See the *cookbook* for some advanced strategies

pandas provides a single function, `merge`, as the entry point for all standard database join operations between DataFrame objects:

```
merge(left, right, how='left', on=None, left_on=None, right_on=None,
      left_index=False, right_index=False, sort=True,
      suffixes=('_x', '_y'), copy=True)
```

Here's a description of what each argument is for:

- `left`: A DataFrame object
- `right`: Another DataFrame object
- `on`: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the DataFrames will be inferred to be the join keys
- `left_on`: Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- `right_on`: Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- `left_index`: If `True`, use the index (row labels) from the left DataFrame as its join key(s). In the case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame
- `right_index`: Same usage as `left_index` for the right DataFrame
- `how`: One of `'left'`, `'right'`, `'outer'`, `'inner'`. Defaults to `inner`. See below for more detailed description of each method
- `sort`: Sort the result DataFrame by the join keys in lexicographical order. Defaults to `True`, setting to `False` will improve performance substantially in many cases
- `suffixes`: A tuple of string suffixes to apply to overlapping columns. Defaults to `('_x', '_y')`.
- `copy`: Always copy data (default `True`) from the passed DataFrame objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.

`merge` is a function in the pandas namespace, and it is also available as a DataFrame instance method, with the calling DataFrame being implicitly considered the left object in the join.

The related `DataFrame.join` method, uses `merge` internally for the index-on-index and index-on-column(s) joins, but *joins on indexes* by default rather than trying to join on common columns (the default behavior for `merge`). If you are joining on index, you may wish to use `DataFrame.join` to save yourself some typing.

13.2.1 Brief primer on merge methods (relational algebra)

Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (DataFrame objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two DataFrame objects on their indexes (which must contain unique values)
- **many-to-one** joins: for example when joining an index (unique) to one or more columns in a DataFrame
- **many-to-many** joins: joining columns on columns.

Note: When joining columns on columns (potentially a many-to-many join), any indexes on the passed DataFrame objects **will be discarded**.

It is worth spending some time understanding the result of the **many-to-many** join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the **Cartesian product** of the associated data. Here is a very basic example with one unique key combination:

```
In [55]: left = DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [56]: right = DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [57]: left

   key  lval
0  foo     1
1  foo     2

In [58]: right

   key  rval
0  foo     4
1  foo     5

In [59]: merge(left, right, on='key')

   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Here is a more complicated example with multiple join keys:

```
In [60]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                   'key2': ['one', 'two', 'one'],
   ....:                   'lval': [1, 2, 3]})
   ....:

In [61]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                    'key2': ['one', 'one', 'one', 'two'],
```

```
   ....:                     'rval': [4, 5, 6, 7]})
   ....:

In [62]: merge(left, right, how='outer')

  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  foo  two     2   NaN
3  bar  one     3     6
4  bar  two   NaN     7

In [63]: merge(left, right, how='inner')

  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  bar  one     3     6
```

The `how` argument to `merge` specifies how to determine which keys are to be included in the resulting table. If a key combination **does not appear** in either the left or right tables, the values in the joined table will be `NA`. Here is a summary of the `how` options and their SQL equivalent names:

Merge method	SQL Join Name	Description
`left`	`LEFT OUTER JOIN`	Use keys from left frame only
`right`	`RIGHT OUTER JOIN`	Use keys from right frame only
`outer`	`FULL OUTER JOIN`	Use union of keys from both frames
`inner`	`INNER JOIN`	Use intersection of keys from both frames

13.2.2 Joining on index

`DataFrame.join` is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Here is a very basic example:

```
In [64]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [65]: df1 = df.ix[1:, ['A', 'B']]

In [66]: df2 = df.ix[:5, ['C', 'D']]

In [67]: df1

          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 [68]: df2

          C         D
0  0.377953  0.493672
1  2.015523 -1.833722
2  0.049307 -0.521493
```

```
3  0.146111  1.903247
4  0.393876  1.861468
5 -2.655452  1.219492
```

```
In [69]: df1.join(df2)
```

```
          A         B         C         D
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740 -0.670027  0.049307 -0.521493
3 -3.201750  0.792716  0.146111  1.903247
4 -0.747169 -0.309038  0.393876  1.861468
5  0.936527  1.255746 -2.655452  1.219492
6  0.062297 -0.110388       NaN       NaN
7  0.077849  0.629498       NaN       NaN
```

```
In [70]: df1.join(df2, how='outer')
```

```
          A         B         C         D
0       NaN       NaN  0.377953  0.493672
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740 -0.670027  0.049307 -0.521493
3 -3.201750  0.792716  0.146111  1.903247
4 -0.747169 -0.309038  0.393876  1.861468
5  0.936527  1.255746 -2.655452  1.219492
6  0.062297 -0.110388       NaN       NaN
7  0.077849  0.629498       NaN       NaN
```

```
In [71]: df1.join(df2, how='inner')
```

```
          A         B         C         D
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740 -0.670027  0.049307 -0.521493
3 -3.201750  0.792716  0.146111  1.903247
4 -0.747169 -0.309038  0.393876  1.861468
5  0.936527  1.255746 -2.655452  1.219492
```

The data alignment here is on the indexes (row labels). This same behavior can be achieved using `merge` plus additional arguments instructing it to use the indexes:

```
In [72]: merge(df1, df2, left_index=True, right_index=True, how='outer')
```

```
          A         B         C         D
0       NaN       NaN  0.377953  0.493672
1 -2.461467 -1.553902  2.015523 -1.833722
2  1.771740 -0.670027  0.049307 -0.521493
3 -3.201750  0.792716  0.146111  1.903247
4 -0.747169 -0.309038  0.393876  1.861468
5  0.936527  1.255746 -2.655452  1.219492
6  0.062297 -0.110388       NaN       NaN
7  0.077849  0.629498       NaN       NaN
```

13.2.3 Joining key columns on an index

`join` takes an optional `on` argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
merge(left, right, left_on=key_or_keys, right_index=True,
      how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the DataFrame's is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [73]: df['key'] = ['foo', 'bar'] * 4

In [74]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
   ....:                     columns=['j1', 'j2'])
   ....:

In [75]: df

          A         B         C         D  key
0 -0.308853 -0.681087  0.377953  0.493672  foo
1 -2.461467 -1.553902  2.015523 -1.833722  bar
2  1.771740 -0.670027  0.049307 -0.521493  foo
3 -3.201750  0.792716  0.146111  1.903247  bar
4 -0.747169 -0.309038  0.393876  1.861468  foo
5  0.936527  1.255746 -2.655452  1.219492  bar
6  0.062297 -0.110388 -1.184357 -0.558081  foo
7  0.077849  0.629498 -1.035260 -0.438229  bar

In [76]: to_join

           j1        j2
bar  0.503703  0.413086
foo -1.139050  0.660342

In [77]: df.join(to_join, on='key')

          A         B         C         D  key        j1        j2
0 -0.308853 -0.681087  0.377953  0.493672  foo -1.139050  0.660342
1 -2.461467 -1.553902  2.015523 -1.833722  bar  0.503703  0.413086
2  1.771740 -0.670027  0.049307 -0.521493  foo -1.139050  0.660342
3 -3.201750  0.792716  0.146111  1.903247  bar  0.503703  0.413086
4 -0.747169 -0.309038  0.393876  1.861468  foo -1.139050  0.660342
5  0.936527  1.255746 -2.655452  1.219492  bar  0.503703  0.413086
6  0.062297 -0.110388 -1.184357 -0.558081  foo -1.139050  0.660342
7  0.077849  0.629498 -1.035260 -0.438229  bar  0.503703  0.413086

In [78]: merge(df, to_join, left_on='key', right_index=True,
   ....:       how='left', sort=False)
   ....:

          A         B         C         D  key        j1        j2
0 -0.308853 -0.681087  0.377953  0.493672  foo -1.139050  0.660342
1 -2.461467 -1.553902  2.015523 -1.833722  bar  0.503703  0.413086
2  1.771740 -0.670027  0.049307 -0.521493  foo -1.139050  0.660342
3 -3.201750  0.792716  0.146111  1.903247  bar  0.503703  0.413086
4 -0.747169 -0.309038  0.393876  1.861468  foo -1.139050  0.660342
5  0.936527  1.255746 -2.655452  1.219492  bar  0.503703  0.413086
6  0.062297 -0.110388 -1.184357 -0.558081  foo -1.139050  0.660342
7  0.077849  0.629498 -1.035260 -0.438229  bar  0.503703  0.413086
```

To join on multiple keys, the passed DataFrame must have a `MultiIndex`:

```
In [79]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   ....:                            ['one', 'two', 'three']],
   ....:                    labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   ....:                            [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   ....:                    names=['first', 'second'])
   ....:

In [80]: to_join = DataFrame(np.random.randn(10, 3), index=index,
   ....:                     columns=['j_one', 'j_two', 'j_three'])
   ....:

# a little relevant example with NAs
In [81]: key1 = ['bar', 'bar', 'bar', 'foo', 'foo', 'baz', 'baz', 'qux',
   ....:         'qux', 'snap']
   ....:

In [82]: key2 = ['two', 'one', 'three', 'one', 'two', 'one', 'two', 'two',
   ....:         'three', 'one']
   ....:

In [83]: data = np.random.randn(len(key1))

In [84]: data = DataFrame({'key1' : key1, 'key2' : key2,
   ....:                   'data' : data})
   ....:

In [85]: data

       data  key1   key2
0 -1.004168   bar    two
1 -1.377627   bar    one
2  0.499281   bar  three
3 -1.405256   foo    one
4  0.162565   foo    two
5 -0.067785   baz    one
6 -1.260006   baz    two
7 -1.132896   qux    two
8 -2.006481   qux  three
9  0.301016  snap    one

In [86]: to_join

                 j_one     j_two   j_three
first second
foo   one     0.464794 -0.309337 -0.649593
      two     0.683758 -0.643834  0.421287
      three   1.032814 -1.290493  0.787872
bar   one     1.515707 -0.276487 -0.223762
      two     1.397431  1.503874 -0.478905
baz   two    -0.135950 -0.730327 -0.033277
      three   0.281151 -1.298915 -2.819487
qux   one    -0.851985 -1.106952 -0.937731
      two    -1.537770  0.555759 -2.277282
      three  -0.390201  1.207122  0.178690
```

Now this can be joined by passing the two key column names:

```
In [87]: data.join(to_join, on=['key1', 'key2'])
```

```
       data  key1   key2     j_one     j_two   j_three
0 -1.004168   bar    two  1.397431  1.503874 -0.478905
1 -1.377627   bar    one  1.515707 -0.276487 -0.223762
2  0.499281   bar  three       NaN       NaN       NaN
3 -1.405256   foo    one  0.464794 -0.309337 -0.649593
4  0.162565   foo    two  0.683758 -0.643834  0.421287
5 -0.067785   baz    one       NaN       NaN       NaN
6 -1.260006   baz    two -0.135950 -0.730327 -0.033277
7 -1.132896   qux    two -1.537770  0.555759 -2.277282
8 -2.006481   qux  three -0.390201  1.207122  0.178690
9  0.301016  snap    one       NaN       NaN       NaN
```

The default for `DataFrame.join` is to perform a left join (essentially a "VLOOKUP" operation, for Excel users), which uses only the keys found in the calling DataFrame. Other join types, for example inner join, can be just as easily performed:

```
In [88]: data.join(to_join, on=['key1', 'key2'], how='inner')
```

```
       data key1   key2     j_one     j_two   j_three
0 -1.004168  bar    two  1.397431  1.503874 -0.478905
1 -1.377627  bar    one  1.515707 -0.276487 -0.223762
3 -1.405256  foo    one  0.464794 -0.309337 -0.649593
4  0.162565  foo    two  0.683758 -0.643834  0.421287
6 -1.260006  baz    two -0.135950 -0.730327 -0.033277
7 -1.132896  qux    two -1.537770  0.555759 -2.277282
8 -2.006481  qux  three -0.390201  1.207122  0.178690
```

As you can see, this drops any rows where there was no match.

13.2.4 Overlapping value columns

The merge `suffixes` argument takes a tuple of list of strings to append to overlapping column names in the input DataFrames to disambiguate the result columns:

```
In [89]: left = DataFrame({'key': ['foo', 'foo'], 'value': [1, 2]})

In [90]: right = DataFrame({'key': ['foo', 'foo'], 'value': [4, 5]})

In [91]: merge(left, right, on='key', suffixes=['_left', '_right'])
```

```
   key  value_left  value_right
0  foo           1            4
1  foo           1            5
2  foo           2            4
3  foo           2            5
```

`DataFrame.join` has `lsuffix` and `rsuffix` arguments which behave similarly.

13.2.5 Merging Ordered Data

New in v0.8.0 is the ordered_merge function for combining time series and other ordered data. In particular it has an optional `fill_method` keyword to fill/interpolate missing data:

```
In [92]: A
```

```
  group key  lvalue
0     a   a       1
1     a   c       2
2     a   e       3
3     b   a       1
4     b   c       2
5     b   e       3
```

```
In [93]: B
```

```
  key  rvalue
0   b       1
1   c       2
2   d       3
```

```
In [94]: ordered_merge(A, B, fill_method='ffill', left_by='group')
```

```
  group key  lvalue  rvalue
0     a   a       1     NaN
1     a   b       1       1
2     a   c       2       2
3     a   d       2       3
4     a   e       3       3
5     b   a       1     NaN
6     b   b       1       1
7     b   c       2       2
8     b   d       2       3
9     b   e       3       3
```

13.2.6 Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to `DataFrame.join` to join them together on their indexes. The same is true for `Panel.join`.

```
In [95]: df1 = df.ix[:, ['A', 'B']]
```

```
In [96]: df2 = df.ix[:, ['C', 'D']]
```

```
In [97]: df3 = df.ix[:, ['key']]
```

```
In [98]: df1
```

```
          A         B
0 -0.308853 -0.681087
1 -2.461467 -1.553902
2  1.771740 -0.670027
3 -3.201750  0.792716
4 -0.747169 -0.309038
5  0.936527  1.255746
6  0.062297 -0.110388
7  0.077849  0.629498
```

```
In [99]: df1.join([df2, df3])
```

```
          A         B         C         D  key
```

```
0 -0.308853 -0.681087  0.377953  0.493672  foo
1 -2.461467 -1.553902  2.015523 -1.833722  bar
2  1.771740 -0.670027  0.049307 -0.521493  foo
3 -3.201750  0.792716  0.146111  1.903247  bar
4 -0.747169 -0.309038  0.393876  1.861468  foo
5  0.936527  1.255746 -2.655452  1.219492  bar
6  0.062297 -0.110388 -1.184357 -0.558081  foo
7  0.077849  0.629498 -1.035260 -0.438229  bar
```

13.2.7 Merging together values within Series or DataFrame columns

Another fairly common situation is to have two like-indexed (or similarly indexed) Series or DataFrame objects and wanting to "patch" values in one object from values for matching indices in the other. Here is an example:

```
In [100]: df1 = DataFrame([[nan, 3., 5.], [-4.6, np.nan, nan],
   .....:                  [nan, 7., nan]])
   .....:

In [101]: df2 = DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4]],
   .....:                 index=[1, 2])
   .....:
```

For this, use the `combine_first` method:

```
In [102]: df1.combine_first(df2)

     0   1    2
0  NaN   3  5.0
1 -4.6 NaN -8.2
2 -5.0   7  4.0
```

Note that this method only takes values from the right DataFrame if they are missing in the left DataFrame. A related method, `update`, alters non-NA values inplace:

```
In [103]: df1.update(df2)

In [104]: df1

      0    1    2
0   NaN  3.0  5.0
1 -42.6  NaN -8.2
2  -5.0  1.6  4.0
```

CHAPTER

FOURTEEN

RESHAPING AND PIVOT TABLES

14.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called "stacked" or "record" format:

```
In [1]: df
```

```
                  date variable     value
0  2000-01-03 00:00:00        A  0.469112
1  2000-01-04 00:00:00        A -0.282863
2  2000-01-05 00:00:00        A -1.509059
3  2000-01-03 00:00:00        B -1.135632
4  2000-01-04 00:00:00        B  1.212112
5  2000-01-05 00:00:00        B -0.173215
6  2000-01-03 00:00:00        C  0.119209
7  2000-01-04 00:00:00        C -1.044236
8  2000-01-05 00:00:00        C -0.861849
9  2000-01-03 00:00:00        D -2.104569
10 2000-01-04 00:00:00        D -0.494929
11 2000-01-05 00:00:00        D  1.071804
```

For the curious here is how the above DataFrame was created:

```
import pandas.util.testing as tm; tm.N = 3
def unpivot(frame):
    N, K = frame.shape
    data = {'value' : frame.values.ravel('F'),
            'variable' : np.asarray(frame.columns).repeat(N),
            'date' : np.tile(np.asarray(frame.index), K)}
    return DataFrame(data, columns=['date', 'variable', 'value'])
df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable `A` we could do:

```
In [2]: df[df['variable'] == 'A']
```

```
                 date variable     value
0 2000-01-03 00:00:00        A  0.469112
1 2000-01-04 00:00:00        A -0.282863
2 2000-01-05 00:00:00        A -1.509059
```

But suppose we wish to do time series operations with the variables. A better representation would be where the `columns` are the unique variables and an `index` of dates identifies individual observations. To reshape the data into this form, use the `pivot` function:

```
In [3]: df.pivot(index='date', columns='variable', values='value')

variable           A         B         C         D
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804
```

If the `values` argument is omitted, and the input DataFrame has more than one column of values which are not used as column or index inputs to `pivot`, then the resulting "pivoted" DataFrame will have *hierarchical columns* whose topmost level indicates the respective value column:

```
In [4]: df['value2'] = df['value'] * 2

In [5]: pivoted = df.pivot('date', 'variable')

In [6]: pivoted

               value                                     value2            \
variable           A         B         C         D         A         B
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569  0.938225 -2.271265
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429

variable           C         D
date
2000-01-03  0.238417 -4.209138
2000-01-04 -2.088472 -0.989859
2000-01-05 -1.723698  2.143608
```

You of course can then select subsets from the pivoted DataFrame:

```
In [7]: pivoted['value2']

variable           A         B         C         D
date
2000-01-03  0.938225 -2.271265  0.238417 -4.209138
2000-01-04 -0.565727  2.424224 -2.088472 -0.989859
2000-01-05 -3.018117 -0.346429 -1.723698  2.143608
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

14.2 Reshaping by stacking and unstacking

Closely related to the `pivot` function are the related `stack` and `unstack` functions currently available on Series and DataFrame. These functions are designed to work together with `MultiIndex` objects (see the section on *hierarchical indexing*). Here are essentially what these functions do:

- `stack`: "pivot" a level of the (possibly hierarchical) column labels, returning a DataFrame with an index with a new inner-most level of row labels.
- `unstack`: inverse operation from `stack`: "pivot" a level of the (possibly hierarchical) row index to the column axis, producing a reshaped DataFrame with a new inner-most level of column labels.

The clearest way to explain is by example. Let's take a prior example data set from the hierarchical indexing section:

```
In [8]: tuples = zip(*[['bar', 'bar', 'baz', 'baz',
   ...:                 'foo', 'foo', 'qux', 'qux'],
   ...:                ['one', 'two', 'one', 'two',
   ...:                 'one', 'two', 'one', 'two']])
   ...:

In [9]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [10]: df = DataFrame(randn(8, 2), index=index, columns=['A', 'B'])

In [11]: df2 = df[:4]

In [12]: df2
```

```
                     A         B
first second
bar   one      0.721555 -0.706771
      two     -1.039575  0.271860
baz   one     -0.424972  0.567020
      two      0.276232 -1.087401
```

The `stack` function "compresses" a level in the DataFrame's columns to produce either:

- A Series, in the case of a simple column Index
- A DataFrame, in the case of a `MultiIndex` in the columns

If the columns have a `MultiIndex`, you can choose which level to stack. The stacked level becomes the new lowest level in a `MultiIndex` on the columns:

```
In [13]: stacked = df2.stack()

In [14]: stacked
```

```
first  second
bar    one     A    0.721555
               B   -0.706771
       two     A   -1.039575
               B    0.271860
baz    one     A   -0.424972
               B    0.567020
       two     A    0.276232
               B   -1.087401
dtype: float64
```

With a "stacked" DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack` is `unstack`, which by default unstacks the **last level**:

```
In [15]: stacked.unstack()
```

```
                     A         B
first second
bar   one      0.721555 -0.706771
      two     -1.039575  0.271860
baz   one     -0.424972  0.567020
      two      0.276232 -1.087401
```

```
In [16]: stacked.unstack(1)
```

```
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 [17]: stacked.unstack(0)
```

```
first          bar       baz
second
one    A  0.721555 -0.424972
       B -0.706771  0.567020
two    A -1.039575  0.276232
       B  0.271860 -1.087401
```

If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [18]: stacked.unstack('second')
```

```
second        one       two
first
bar   A  0.721555 -1.039575
      B -0.706771  0.271860
baz   A -0.424972  0.276232
      B  0.567020 -1.087401
```

You may also stack or unstack more than one level at a time by passing a list of levels, in which case the end result is as if each level in the list were processed individually.

These functions are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling `sortlevel`, of course). Here is a more complex example:

```
In [19]: columns = MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
   ....:                                   ('B', 'cat'), ('A', 'dog')],
   ....:                                  names=['exp', 'animal'])
   ....:

In [20]: df = DataFrame(randn(8, 4), index=index, columns=columns)

In [21]: df2 = df.ix[[0, 1, 2, 4, 5, 7]]

In [22]: df2
```

```
exp                 A         B                   A
animal            cat       dog       cat       dog
first second
bar   one    -0.370647 -1.157892 -1.344312  0.844885
      two     1.075770 -0.109050  1.643563 -1.469388
baz   one     0.357021 -0.674600 -1.776904 -0.968914
foo   one    -0.013960 -0.362543 -0.006154 -0.923061
      two     0.895717  0.805244 -1.206412  2.565646
qux   two     0.410835  0.813850  0.132003 -0.827317
```

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [23]: df2.stack('exp')
```

```
animal                cat       dog
```

```
first second exp
bar   one    A   -0.370647  0.844885
             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   one    A   -0.013960 -0.923061
             B   -0.006154 -0.362543
      two    A    0.895717  2.565646
             B   -1.206412  0.805244
qux   two    A    0.410835 -0.827317
             B    0.132003  0.813850
```

```
In [24]: df2.stack('animal')
```

```
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   one    cat    -0.013960 -0.006154
             dog    -0.923061 -0.362543
      two    cat     0.895717 -1.206412
             dog     2.565646  0.805244
qux   two    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 [25]: df[:3].unstack(0)
```

```
exp            A                   B                                     A  \
animal       cat                 dog                 cat                 dog
first        bar       baz       bar     baz       bar       baz       bar
second
one    -0.370647  0.357021 -1.157892 -0.6746 -1.344312 -1.776904  0.844885
two     1.075770       NaN -0.109050     NaN  1.643563       NaN -1.469388
exp
animal
first        baz
second
one    -0.968914
two          NaN
```

```
In [26]: df2.unstack(1)
```

```
exp            A                   B                                       A  \
animal       cat                 dog                   cat                 dog
second       one       two       one       two       one       two       one
first
bar    -0.370647  1.075770 -1.157892 -0.109050 -1.344312  1.643563  0.844885
baz     0.357021       NaN -0.674600       NaN -1.776904       NaN -0.968914
foo    -0.013960  0.895717 -0.362543  0.805244 -0.006154 -1.206412 -0.923061
qux          NaN  0.410835       NaN  0.813850       NaN  0.132003       NaN
```

```
exp
animal
second       two
first
bar    -1.469388
baz          NaN
foo     2.565646
qux    -0.827317
```

14.3 Reshaping by Melt

The `melt` function found in `pandas.core.reshape` is useful to massage a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are "pivoted" to the row axis, leaving just two non-identifier columns, "variable" and "value". The names of those columns can be customized by supplying the `var_name` and `value_name` parameters.

For instance,

```
In [27]: cheese = DataFrame({'first' : ['John', 'Mary'],
   ....:                     'last' : ['Doe', 'Bo'],
   ....:                     'height' : [5.5, 6.0],
   ....:                     'weight' : [130, 150]})
   ....:

In [28]: cheese

  first  height last  weight
0  John     5.5  Doe     130
1  Mary     6.0   Bo     150

In [29]: melt(cheese, id_vars=['first', 'last'])

  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0

In [30]: melt(cheese, id_vars=['first', 'last'], var_name='quantity')

  first last quantity  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

14.4 Combining with stats and GroupBy

It should be no shock that combining `pivot` / `stack` / `unstack` with GroupBy and the basic Series and DataFrame statistical functions can produce some very expressive and fast data manipulations.

```
In [31]: df

exp                 A         B                   A
```

```
animal             cat       dog       cat       dog
first second
bar   one    -0.370647 -1.157892 -1.344312  0.844885
      two     1.075770 -0.109050  1.643563 -1.469388
baz   one     0.357021 -0.674600 -1.776904 -0.968914
      two    -1.294524  0.413738  0.276662 -0.472035
foo   one    -0.013960 -0.362543 -0.006154 -0.923061
      two     0.895717  0.805244 -1.206412  2.565646
qux   one     1.431256  1.340309 -1.170299 -0.226169
      two     0.410835  0.813850  0.132003 -0.827317

In [32]: df.stack().mean(1).unstack()

animal             cat       dog
first second
bar   one    -0.857479 -0.156504
      two     1.359666 -0.789219
baz   one    -0.709942 -0.821757
      two    -0.508931 -0.029148
foo   one    -0.010057 -0.642802
      two    -0.155347  1.685445
qux   one     0.130479  0.557070
      two     0.271419 -0.006733

# same result, another way
In [33]: df.groupby(level=1, axis=1).mean()

animal             cat       dog
first second
bar   one    -0.857479 -0.156504
      two     1.359666 -0.789219
baz   one    -0.709942 -0.821757
      two    -0.508931 -0.029148
foo   one    -0.010057 -0.642802
      two    -0.155347  1.685445
qux   one     0.130479  0.557070
      two     0.271419 -0.006733

In [34]: df.stack().groupby(level=1).mean()

exp            A         B
second
one     0.016301 -0.644049
two     0.110588  0.346200

In [35]: df.mean().unstack(0)

exp            A         B
animal
cat     0.311433 -0.431481
dog    -0.184544  0.133632
```

14.5 Pivot tables and cross-tabulations

The function `pandas.pivot_table` can be used to create spreadsheet-style pivot tables. See the *cookbook* for some advanced strategies

It takes a number of arguments

- `data`: A DataFrame object
- `values`: a column or a list of columns to aggregate
- `rows`: list of columns to group by on the table rows
- `cols`: list of columns to group by on the table columns
- `aggfunc`: function to use for aggregation, defaulting to `numpy.mean`

Consider a data set like this:

```
In [36]: 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 [37]: df
```

```
        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 [38]: pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
```

```
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 [39]: pivot_table(df, values='D', rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
```

```
A       one               three                 two
C       bar       foo       bar       foo       bar       foo
B
A -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 [40]: pivot_table(df, values=['D','E'], rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
```

```
          D                                                     E  \
A       one               three                 two                one
C       bar       foo       bar       foo       bar       foo       bar
B
A -2.309255 -0.486468 -2.655954       NaN       NaN -0.257067  0.316495
B -2.640506 -1.266315       NaN -0.158102  1.670241       NaN -1.077692
C  2.377724  0.754600 -1.665013       NaN       NaN  1.676079  2.001971

A              three                 two
C       foo       bar      foo       bar       foo
B
A  0.005518 -0.676843      NaN       NaN  0.867024
B  0.352360       NaN  1.39907  1.177566       NaN
C  2.241830 -1.687290      NaN       NaN -2.363137
```

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 [41]: pivot_table(df, rows=['A', 'B'], cols=['C'])
```

```
                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 [42]: table = pivot_table(df, rows=['A', 'B'], cols=['C'])

In [43]: 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.

14.5.1 Cross tabulations

Use the `crosstab` function to compute a cross-tabulation of two (or more) factors. By default `crosstab` computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- `rows`: array-like, values to group by in the rows
- `cols`: array-like, values to group by in the columns
- `values`: array-like, optional, array of values to aggregate according to the factors
- `aggfunc`: function, optional, If no values array is passed, computes a frequency table
- `rownames`: sequence, default None, must match number of row arrays passed
- `colnames`: sequence, default None, if passed, must match number of column arrays passed
- `margins`: boolean, default False, Add row/column margins (subtotals)

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

```
In [44]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [45]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [46]: b = np.array([one, one, two, one, two, one], dtype=object)

In [47]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [48]: crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])

b     one          two
c    dull  shiny  dull  shiny
a
bar     1      0     0      1
foo     2      1     1      0
```

14.5.2 Adding margins (partial aggregates)

If you pass `margins=True` to `pivot_table`, special `All` columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
In [49]: df.pivot_table(rows=['A', 'B'], cols='C', margins=True, aggfunc=np.std)
```

```
                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
```

14.6 Tiling

The `cut` function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

```
In [50]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [51]: cut(ages, bins=3)
```

```
Categorical:
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (26.
Levels (3): Index(['(9.95, 26.667]', '(26.667, 43.333]', '(43.333, 60]'], dtype=object)
```

If the `bins` keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [52]: cut(ages, bins=[0, 18, 35, 70])
```

```
Categorical:
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]]
Levels (3): Index(['(0, 18]', '(18, 35]', '(35, 70]'], dtype=object)
```

CHAPTER

FIFTEEN

TIME SERIES / DATE FUNCTIONALITY

pandas has proven very successful as a tool for working with time series data, especially in the financial data analysis space. With the 0.8 release, we have further improved the time series API in pandas by leaps and bounds. Using the new NumPy `datetime64` dtype, we have consolidated a large number of features from other Python libraries like `scikits.timeseries` as well as created a tremendous amount of new functionality for manipulating time series data.

In working with time series data, we will frequently seek to:

- generate sequences of fixed-frequency dates and time spans
- conform or convert time series to a particular frequency
- compute "relative" dates based on various non-standard time increments (e.g. 5 business days before the last business day of the year), or "roll" dates forward or backward

pandas provides a relatively compact and self-contained set of tools for performing the above tasks.

Create a range of dates:

```
# 72 hours starting with midnight Jan 1st, 2011
In [1]: rng = date_range('1/1/2011', periods=72, freq='H')

In [2]: rng[:5]

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 04:00:00]
Length: 5, Freq: H, Timezone: None
```

Index pandas objects with dates:

```
In [3]: ts = Series(randn(len(rng)), index=rng)

In [4]: ts.head()

2011-01-01 00:00:00    0.469112
2011-01-01 01:00:00   -0.282863
2011-01-01 02:00:00   -1.509059
2011-01-01 03:00:00   -1.135632
2011-01-01 04:00:00    1.212112
Freq: H, dtype: float64
```

Change frequency and fill gaps:

```
# to 45 minute frequency and forward fill
In [5]: converted = ts.asfreq('45Min', method='pad')
```

```
In [6]: converted.head()

2011-01-01 00:00:00    0.469112
2011-01-01 00:45:00    0.469112
2011-01-01 01:30:00   -0.282863
2011-01-01 02:15:00   -1.509059
2011-01-01 03:00:00   -1.135632
Freq: 45T, dtype: float64
```

Resample:

```
# Daily means
In [7]: ts.resample('D', how='mean')

2011-01-01   -0.319569
2011-01-02   -0.337703
2011-01-03    0.117258
Freq: D, dtype: float64
```

15.1 Time Stamps vs. Time Spans

Time-stamped data is the most basic type of timeseries data that associates values with points in time. For pandas objects it means using the points in time to create the index

```
In [8]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]

In [9]: ts = Series(np.random.randn(3), dates)

In [10]: type(ts.index)
pandas.tseries.index.DatetimeIndex

In [11]: ts

2012-05-01   -0.410001
2012-05-02   -0.078638
2012-05-03    0.545952
dtype: float64
```

However, in many cases it is more natural to associate things like change variables with a time span instead.

For example:

```
In [12]: periods = PeriodIndex([Period('2012-01'), Period('2012-02'),
   ....:                        Period('2012-03')])
   ....:

In [13]: ts = Series(np.random.randn(3), periods)

In [14]: type(ts.index)
pandas.tseries.period.PeriodIndex

In [15]: ts

2012-01   -1.219217
2012-02   -1.226825
2012-03    0.769804
Freq: M, dtype: float64
```

Starting with 0.8, pandas allows you to capture both representations and convert between them. Under the hood, pandas represents timestamps using instances of `Timestamp` and sequences of timestamps using instances of `DatetimeIndex`. For regular time spans, pandas uses `Period` objects for scalar values and `PeriodIndex` for sequences of spans. Better support for irregular intervals with arbitrary start and end points are forth-coming in future releases.

15.2 Converting to Timestamps

To convert a Series or list-like object of date-like objects e.g. strings, epochs, or a mixture, you can use the `to_datetime` function. When passed a Series, this returns a Series (with the same index), while a list-like is converted to a DatetimeIndex:

```
In [16]: to_datetime(Series(['Jul 31, 2009', '2010-01-10', None]))

0   2009-07-31 00:00:00
1   2010-01-10 00:00:00
2                   NaT
dtype: datetime64[ns]

In [17]: to_datetime(['2005/11/23', '2010.12.31'])

<class 'pandas.tseries.index.DatetimeIndex'>
[2005-11-23 00:00:00, 2010-12-31 00:00:00]
Length: 2, Freq: None, Timezone: None
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```
In [18]: to_datetime(['04-01-2012 10:00'], dayfirst=True)

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-04 10:00:00]
Length: 1, Freq: None, Timezone: None

In [19]: to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-14 00:00:00, 2012-01-14 00:00:00]
Length: 2, Freq: None, Timezone: None
```

> **Warning:** You see in the above example that `dayfirst` isn't strict, so if a date can't be parsed with the day being first it will be parsed as if `dayfirst` were False.

Pass `coerce=True` to convert bad data to `NaT` (not a time):

```
In [20]: to_datetime(['2009-07-31', 'asd'])
array(['2009-07-31', 'asd'], dtype=object)

In [21]: to_datetime(['2009-07-31', 'asd'], coerce=True)

<class 'pandas.tseries.index.DatetimeIndex'>
[2009-07-31 00:00:00, NaT]
Length: 2, Freq: None, Timezone: None
```

It's also possible to convert integer or float epoch times. The default unit for these is nanoseconds (since these are how Timestamps are stored). However, often epochs are stored in another `unit` which can be specified:

```
In [22]: to_datetime([1])

<class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:00.000000001]
Length: 1, Freq: None, Timezone: None

In [23]: to_datetime([1, 3.14], unit='s')

<class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:01, 1970-01-01 00:00:03.140000]
Length: 2, Freq: None, Timezone: None
```

Note: Epoch times will be rounded to the nearest nanosecond.

Take care, `to_datetime` may not act as you expect on mixed data:

```
In [24]: pd.to_datetime([1, '1'])

<class 'pandas.tseries.index.DatetimeIndex'>
[1970-01-01 00:00:00.000000001, 2014-01-01 00:00:00]
Length: 2, Freq: None, Timezone: None
```

15.3 Generating Ranges of Timestamps

To generate an index with time stamps, you can use either the DatetimeIndex or Index constructor and pass in a list of datetime objects:

```
In [25]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]

In [26]: index = DatetimeIndex(dates)

In [27]: index # Note the frequency information

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01 00:00:00, ..., 2012-05-03 00:00:00]
Length: 3, Freq: None, Timezone: None

In [28]: index = Index(dates)

In [29]: index # Automatically converted to DatetimeIndex

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01 00:00:00, ..., 2012-05-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

Practically, this becomes very cumbersome because we often need a very long index with a large number of timestamps. If we need timestamps on a regular frequency, we can use the pandas functions `date_range` and `bdate_range` to create timestamp indexes.

```
In [30]: index = date_range('2000-1-1', periods=1000, freq='M')

In [31]: index

<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31 00:00:00, ..., 2083-04-30 00:00:00]
```

```
Length: 1000, Freq: M, Timezone: None

In [32]: index = bdate_range('2012-1-1', periods=250)

In [33]: index

<class 'pandas.tseries.index.DatetimeIndex'>
[2012-01-02 00:00:00, ..., 2012-12-14 00:00:00]
Length: 250, Freq: B, Timezone: None
```

Convenience functions like `date_range` and `bdate_range` utilize a variety of frequency aliases. The default frequency for `date_range` is a **calendar day** while the default for `bdate_range` is a **business day**

```
In [34]: start = datetime(2011, 1, 1)

In [35]: end = datetime(2012, 1, 1)

In [36]: rng = date_range(start, end)

In [37]: rng

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2012-01-01 00:00:00]
Length: 366, Freq: D, Timezone: None

In [38]: rng = bdate_range(start, end)

In [39]: rng

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-12-30 00:00:00]
Length: 260, Freq: B, Timezone: None
```

`date_range` and `bdate_range` makes it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`:

```
In [40]: date_range(start, end, freq='BM')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None

In [41]: date_range(start, end, freq='W')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-02 00:00:00, ..., 2012-01-01 00:00:00]
Length: 53, Freq: W-SUN, Timezone: None

In [42]: bdate_range(end=end, periods=20)

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-12-05 00:00:00, ..., 2011-12-30 00:00:00]
Length: 20, Freq: B, Timezone: None

In [43]: bdate_range(start=start, periods=20)

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-28 00:00:00]
Length: 20, Freq: B, Timezone: None
```

The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

15.4 DatetimeIndex

One of the main uses for `DatetimeIndex` is as an index for pandas objects. The `DatetimeIndex` class contains many timeseries related optimizations:

- A large range of dates for various offsets are pre-computed and cached under the hood in order to make generating subsequent date ranges very fast (just have to grab a slice)
- Fast shifting using the `shift` and `tshift` method on pandas objects
- Unioning of overlapping DatetimeIndex objects with the same frequency is very fast (important for fast data alignment)
- Quick access to date fields via properties such as `year`, `month`, etc.
- Regularization functions like `snap` and very fast `asof` logic

DatetimeIndex objects has all the basic functionality of regular Index objects and a smorgasbord of advanced timeseries-specific methods for easy frequency processing.

See Also:

Reindexing methods

Note: While pandas does not force you to have a sorted date index, some of these methods may have unexpected or incorrect behavior if the dates are unsorted. So please be careful.

`DatetimeIndex` can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```
In [44]: rng = date_range(start, end, freq='BM')

In [45]: ts = Series(randn(len(rng)), index=rng)

In [46]: ts.index

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None

In [47]: ts[:5].index

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-05-31 00:00:00]
Length: 5, Freq: BM, Timezone: None

In [48]: ts[::2].index

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-11-30 00:00:00]
Length: 6, Freq: 2BM, Timezone: None
```

15.4.1 Partial String Indexing

You can pass in dates and strings that parse to dates as indexing parameters:

```
In [49]: ts['1/31/2011']
-1.2812473076599531

In [50]: ts[datetime(2011, 12, 25):]

2011-12-30    0.687738
Freq: BM, dtype: float64

In [51]: ts['10/31/2011':'12/31/2011']

2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
Freq: BM, dtype: float64
```

To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

```
In [52]: ts['2011']

2011-01-31   -1.281247
2011-02-28   -0.727707
2011-03-31   -0.121306
2011-04-29   -0.097883
2011-05-31    0.695775
2011-06-30    0.341734
2011-07-29    0.959726
2011-08-31   -1.110336
2011-09-30   -0.619976
2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
Freq: BM, dtype: float64

In [53]: ts['2011-6']

2011-06-30    0.341734
Freq: BM, dtype: float64
```

This type of slicing will work on a DataFrame with a `DateTimeIndex` as well. Since the partial string selection is a form of label slicing, the endpoints **will be** included. This would include matching times on an included date. Here's an example:

```
In [54]: dft = DataFrame(randn(100000,1),columns=['A'],index=date_range('20130101',periods=100000,fre

In [55]: dft

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 100000 entries, 2013-01-01 00:00:00 to 2013-03-11 10:39:00
Freq: T
Data columns (total 1 columns):
A    100000  non-null values
dtypes: float64(1)

In [56]: dft['2013']

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 100000 entries, 2013-01-01 00:00:00 to 2013-03-11 10:39:00
Freq: T
Data columns (total 1 columns):
```

```
A    100000  non-null values
dtypes: float64(1)
```

This starts on the very first time in the month, and includes the last date & time for the month

```
In [57]: dft['2013-1':'2013-2']
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 84960 entries, 2013-01-01 00:00:00 to 2013-02-28 23:59:00
Freq: T
Data columns (total 1 columns):
A    84960  non-null values
dtypes: float64(1)
```

This specifies a stop time **that includes all of the times on the last day**

```
In [58]: dft['2013-1':'2013-2-28']
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 84960 entries, 2013-01-01 00:00:00 to 2013-02-28 23:59:00
Freq: T
Data columns (total 1 columns):
A    84960  non-null values
dtypes: float64(1)
```

This specifies an **exact** stop time (and is not the same as the above)

```
In [59]: dft['2013-1':'2013-2-28 00:00:00']
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 83521 entries, 2013-01-01 00:00:00 to 2013-02-28 00:00:00
Freq: T
Data columns (total 1 columns):
A    83521  non-null values
dtypes: float64(1)
```

We are stopping on the included end-point as its part of the index

```
In [60]: dft['2013-1-15':'2013-1-15 12:30:00']
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 751 entries, 2013-01-15 00:00:00 to 2013-01-15 12:30:00
Freq: T
Data columns (total 1 columns):
A    751  non-null values
dtypes: float64(1)
```

Warning: The following selection will raises a `KeyError`; otherwise this selection methodology would be inconsistent with other selection methods in pandas (as this is not a *slice*, nor does it resolve to one)

```
dft['2013-1-15 12:30:00']
```

To select a single row, use `.loc`

```
In [61]: dft.loc['2013-1-15 12:30:00']
```

```
A    0.193284
Name: 2013-01-15 12:30:00, dtype: float64
```

15.4.2 Datetime Indexing

Indexing a `DateTimeIndex` with a partial string depends on the "accuracy" of the period, in other words how specific the interval is in relation to the frequency of the index. In contrast, indexing with datetime objects is exact, because the objects have exact meaning. These also follow the sematics of *including both endpoints*.

These `datetime` objects are specific `hours, minutes,` and `seconds` even though they were not explicity specified (they are `0`).

```
In [62]: dft[datetime(2013, 1, 1):datetime(2013,2,28)]
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 83521 entries, 2013-01-01 00:00:00 to 2013-02-28 00:00:00
Freq: T
Data columns (total 1 columns):
A    83521  non-null values
dtypes: float64(1)
```

With no defaults.

```
In [63]: dft[datetime(2013, 1, 1, 10, 12, 0):datetime(2013, 2, 28, 10, 12, 0)]
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 83521 entries, 2013-01-01 10:12:00 to 2013-02-28 10:12:00
Freq: T
Data columns (total 1 columns):
A    83521  non-null values
dtypes: float64(1)
```

15.4.3 Truncating & Fancy Indexing

A `truncate` convenience function is provided that is equivalent to slicing:

```
In [64]: ts.truncate(before='10/31/2011', after='12/31/2011')
```

```
2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
Freq: BM, dtype: float64
```

Even complicated fancy indexing that breaks the DatetimeIndex's frequency regularity will result in a `DatetimeIndex` (but frequency is lost):

```
In [65]: ts[[0, 2, 6]].index
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-07-29 00:00:00]
Length: 3, Freq: None, Timezone: None
```

15.5 DateOffset objects

In the preceding examples, we created DatetimeIndex objects at various frequencies by passing in frequency strings like 'M', 'W', and 'BM to the `freq` keyword. Under the hood, these frequency strings are being translated into an instance of pandas `DateOffset`, which represents a regular frequency increment. Specific offset logic like "month", "business day", or "one hour" is represented in its various subclasses.

Class name	Description
DateOffset	Generic offset class, defaults to 1 calendar day
BDay	business day (weekday)
CDay	custom business day (experimental)
Week	one week, optionally anchored on a day of the week
WeekOfMonth	the x-th day of the y-th week of each month
MonthEnd	calendar month end
MonthBegin	calendar month begin
BMonthEnd	business month end
BMonthBegin	business month begin
QuarterEnd	calendar quarter end
QuarterBegin	calendar quarter begin
BQuarterEnd	business quarter end
BQuarterBegin	business quarter begin
YearEnd	calendar year end
YearBegin	calendar year begin
BYearEnd	business year end
BYearBegin	business year begin
Hour	one hour
Minute	one minute
Second	one second
Milli	one millisecond
Micro	one microsecond

The basic `DateOffset` takes the same arguments as `dateutil.relativedelta`, which works like:

```
In [66]: d = datetime(2008, 8, 18)

In [67]: d + relativedelta(months=4, days=5)
datetime.datetime(2008, 12, 23, 0, 0)
```

We could have done the same thing with `DateOffset`:

```
In [68]: from pandas.tseries.offsets import *

In [69]: d + DateOffset(months=4, days=5)
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 [70]: d - 5 * BDay()
datetime.datetime(2008, 8, 11, 0, 0)
```

```
In [71]: d + BMonthEnd()
datetime.datetime(2008, 8, 29, 0, 0)
```

The `rollforward` and `rollback` methods do exactly what you would expect:

```
In [72]: d
datetime.datetime(2008, 8, 18, 0, 0)

In [73]: offset = BMonthEnd()

In [74]: offset.rollforward(d)
datetime.datetime(2008, 8, 29, 0, 0)

In [75]: offset.rollback(d)
datetime.datetime(2008, 7, 31, 0, 0)
```

It's definitely worth exploring the `pandas.tseries.offsets` module and the various docstrings for the classes.

15.5.1 Parametric offsets

Some of the offsets can be "parameterized" when created to result in different behavior. For example, the `Week` offset for generating weekly data accepts a `weekday` parameter which results in the generated dates always lying on a particular day of the week:

```
In [76]: d + Week()
datetime.datetime(2008, 8, 25, 0, 0)

In [77]: d + Week(weekday=4)
datetime.datetime(2008, 8, 22, 0, 0)

In [78]: (d + Week(weekday=4)).weekday()
4
```

Another example is parameterizing `YearEnd` with the specific ending month:

```
In [79]: d + YearEnd()
datetime.datetime(2008, 12, 31, 0, 0)

In [80]: d + YearEnd(month=6)
datetime.datetime(2009, 6, 30, 0, 0)
```

15.5.2 Custom Business Days (Experimental)

The `CDay` or `CustomBusinessDay` class provides a parametric `BusinessDay` class which can be used to create customized business day calendars which account for local holidays and local weekend conventions.

```
In [81]: from pandas.tseries.offsets import CustomBusinessDay

# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [82]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
# add that for a couple of years
In [83]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]
```

```
In [84]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [85]: dt = datetime(2013, 4, 30)

In [86]: print dt + 2 * bday_egypt
2013-05-05 00:00:00

In [87]: dts = date_range(dt, periods=5, freq=bday_egypt).to_series()

In [88]: print dts
2013-04-30   2013-04-30 00:00:00
2013-05-02   2013-05-02 00:00:00
2013-05-05   2013-05-05 00:00:00
2013-05-06   2013-05-06 00:00:00
2013-05-07   2013-05-07 00:00:00
Freq: C, dtype: datetime64[ns]

In [89]: print Series(dts.weekday, dts).map(Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
dtype: object
```

Note: The frequency string 'C' is used to indicate that a CustomBusinessDay DateOffset is used, it is important to note that since CustomBusinessDay is a parameterised type, instances of CustomBusinessDay may differ and this is not detectable from the 'C' frequency string. The user therefore needs to ensure that the 'C' frequency string is used consistently within the user's application.

Note: This uses the `numpy.busdaycalendar` API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.

Warning: There are known problems with the timezone handling in Numpy 1.7 and users should therefore use this **experimental(!)** feature with caution and at their own risk.
To the extent that the `datetime64` and `busdaycalendar` APIs in Numpy have to change to fix the timezone issues, the behaviour of the `CustomBusinessDay` class may have to change in future versions.

15.5.3 Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as *offset aliases* (referred to as *time rules* prior to v0.8.0).

Alias	Description
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
BM	business month end frequency
MS	month start frequency
BMS	business month start frequency
Q	quarter end frequency
BQ	business quarter endfrequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
H	hourly frequency
T	minutely frequency
S	secondly frequency
L	milliseonds
U	microseconds

15.5.4 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```
In [90]: date_range(start, periods=5, freq='B')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None

In [91]: date_range(start, periods=5, freq=BDay())

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None
```

You can combine together day and intraday offsets:

```
In [92]: date_range(start, periods=10, freq='2h20min')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 21:00:00]
Length: 10, Freq: 140T, Timezone: None

In [93]: date_range(start, periods=10, freq='1D10U')

<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-10 00:00:00.000090]
Length: 10, Freq: 86400000010U, Timezone: None
```

15.5.5 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

Alias	Description
W-SUN	weekly frequency (sundays). Same as 'W'
W-MON	weekly frequency (mondays)
W-TUE	weekly frequency (tuesdays)
W-WED	weekly frequency (wednesdays)
W-THU	weekly frequency (thursdays)
W-FRI	weekly frequency (fridays)
W-SAT	weekly frequency (saturdays)
(B)Q(S)-DEC	quarterly frequency, year ends in December. Same as 'Q'
(B)Q(S)-JAN	quarterly frequency, year ends in January
(B)Q(S)-FEB	quarterly frequency, year ends in February
(B)Q(S)-MAR	quarterly frequency, year ends in March
(B)Q(S)-APR	quarterly frequency, year ends in April
(B)Q(S)-MAY	quarterly frequency, year ends in May
(B)Q(S)-JUN	quarterly frequency, year ends in June
(B)Q(S)-JUL	quarterly frequency, year ends in July
(B)Q(S)-AUG	quarterly frequency, year ends in August
(B)Q(S)-SEP	quarterly frequency, year ends in September
(B)Q(S)-OCT	quarterly frequency, year ends in October
(B)Q(S)-NOV	quarterly frequency, year ends in November
(B)A(S)-DEC	annual frequency, anchored end of December. Same as 'A'
(B)A(S)-JAN	annual frequency, anchored end of January
(B)A(S)-FEB	annual frequency, anchored end of February
(B)A(S)-MAR	annual frequency, anchored end of March
(B)A(S)-APR	annual frequency, anchored end of April
(B)A(S)-MAY	annual frequency, anchored end of May
(B)A(S)-JUN	annual frequency, anchored end of June
(B)A(S)-JUL	annual frequency, anchored end of July
(B)A(S)-AUG	annual frequency, anchored end of August
(B)A(S)-SEP	annual frequency, anchored end of September
(B)A(S)-OCT	annual frequency, anchored end of October
(B)A(S)-NOV	annual frequency, anchored end of November

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

15.5.6 Legacy Aliases

Note that prior to v0.8.0, time rules had a slightly different look. Pandas will continue to support the legacy time rules for the time being but it is strongly recommended that you switch to using the new offset aliases.

Legacy Time Rule	Offset Alias
WEEKDAY	B
EOM	BM
W@MON	W-MON
W@TUE	W-TUE
W@WED	W-WED
W@THU	W-THU
W@FRI	W-FRI
W@SAT	W-SAT
W@SUN	W-SUN
Q@JAN	BQ-JAN
Q@FEB	BQ-FEB
Q@MAR	BQ-MAR
A@JAN	BA-JAN
A@FEB	BA-FEB
A@MAR	BA-MAR
A@APR	BA-APR
A@MAY	BA-MAY
A@JUN	BA-JUN
A@JUL	BA-JUL
A@AUG	BA-AUG
A@SEP	BA-SEP
A@OCT	BA-OCT
A@NOV	BA-NOV
A@DEC	BA-DEC
min	T
ms	L
us	U

As you can see, legacy quarterly and annual frequencies are business quarter and business year ends. Please also note the legacy time rule for milliseconds `ms` versus the new offset alias for month start `MS`. This means that offset alias parsing is case sensitive.

15.6 Time series-related instance methods

15.6.1 Shifting / lagging

One may want to *shift* or *lag* the values in a TimeSeries back and forward in time. The method for this is `shift`, which is available on all of the pandas objects. In DataFrame, `shift` will currently only shift along the `index` and in Panel along the `major_axis`.

```
In [94]: ts = ts[:5]

In [95]: ts.shift(1)

2011-01-31         NaN
2011-02-28   -1.281247
2011-03-31   -0.727707
2011-04-29   -0.121306
2011-05-31   -0.097883
Freq: BM, dtype: float64
```

The shift method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object

or also a *offset alias*:

```
In [96]: ts.shift(5, freq=datetools.bday)

2011-02-07   -1.281247
2011-03-07   -0.727707
2011-04-07   -0.121306
2011-05-06   -0.097883
2011-06-07    0.695775
dtype: float64

In [97]: ts.shift(5, freq='BM')

2011-06-30   -1.281247
2011-07-29   -0.727707
2011-08-31   -0.121306
2011-09-30   -0.097883
2011-10-31    0.695775
Freq: BM, dtype: float64
```

Rather than changing the alignment of the data and the index, `DataFrame` and `TimeSeries` objects also have a `tshift` convenience method that changes all the dates in the index by a specified number of offsets:

```
In [98]: ts.tshift(5, freq='D')

2011-02-05   -1.281247
2011-03-05   -0.727707
2011-04-05   -0.121306
2011-05-04   -0.097883
2011-06-05    0.695775
dtype: float64
```

Note that with `tshift`, the leading entry is no longer NaN because the data is not being realigned.

15.6.2 Frequency conversion

The primary function for changing frequencies is the `asfreq` function. For a `DatetimeIndex`, this is basically just a thin, but convenient wrapper around `reindex` which generates a `date_range` and calls `reindex`.

```
In [99]: dr = date_range('1/1/2010', periods=3, freq=3 * datetools.bday)

In [100]: ts = Series(randn(3), index=dr)

In [101]: ts

2010-01-01   -0.659574
2010-01-06    1.494522
2010-01-11   -0.778425
Freq: 3B, dtype: float64

In [102]: ts.asfreq(BDay())

2010-01-01   -0.659574
2010-01-04         NaN
2010-01-05         NaN
2010-01-06    1.494522
2010-01-07         NaN
2010-01-08         NaN
```

```
2010-01-11   -0.778425
Freq: B, dtype: float64
```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion

```
In [103]: ts.asfreq(BDay(), method='pad')
```

```
2010-01-01   -0.659574
2010-01-04   -0.659574
2010-01-05   -0.659574
2010-01-06    1.494522
2010-01-07    1.494522
2010-01-08    1.494522
2010-01-11   -0.778425
Freq: B, dtype: float64
```

15.6.3 Filling forward / backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the *missing data section*.

15.6.4 Converting to Python datetimes

`DatetimeIndex` can be converted to an array of Python native datetime.datetime objects using the `to_pydatetime` method.

15.7 Up- and downsampling

With 0.8, pandas introduces simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications.

See some *cookbook examples* for some advanced strategies

```
In [104]: rng = date_range('1/1/2012', periods=100, freq='S')

In [105]: ts = Series(randint(0, 500, len(rng)), index=rng)

In [106]: ts.resample('5Min', how='sum')
```

```
2012-01-01    25103
Freq: 5T, dtype: int64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency conversion and resampling operation.

The `how` parameter can be a function name or numpy array function that takes an array and produces aggregated values:

```
In [107]: ts.resample('5Min') # default is mean
```

```
2012-01-01    251.03
Freq: 5T, dtype: float64
```

```
In [108]: ts.resample('5Min', how='ohlc')

            open  high  low  close
2012-01-01   308   460    9    205

In [109]: ts.resample('5Min', how=np.max)

2012-01-01   NaN
Freq: 5T, dtype: float64
```

Any function available via *dispatching* can be given to the `how` parameter by name, including `sum`, `mean`, `std`, `max`, `min`, `median`, `first`, `last`, `ohlc`.

For downsampling, `closed` can be set to 'left' or 'right' to specify which end of the interval is closed:

```
In [110]: ts.resample('5Min', closed='right')

2011-12-31 23:55:00    308.000000
2012-01-01 00:00:00    250.454545
Freq: 5T, dtype: float64

In [111]: ts.resample('5Min', closed='left')

2012-01-01    251.03
Freq: 5T, dtype: float64
```

For upsampling, the `fill_method` and `limit` parameters can be specified to interpolate over the gaps that are created:

```
# from secondly to every 250 milliseconds
In [112]: ts[:2].resample('250L')

2012-01-01 00:00:00           308
2012-01-01 00:00:00.250000    NaN
2012-01-01 00:00:00.500000    NaN
2012-01-01 00:00:00.750000    NaN
2012-01-01 00:00:01           204
Freq: 250L, dtype: float64

In [113]: ts[:2].resample('250L', fill_method='pad')

2012-01-01 00:00:00           308
2012-01-01 00:00:00.250000    308
2012-01-01 00:00:00.500000    308
2012-01-01 00:00:00.750000    308
2012-01-01 00:00:01           204
Freq: 250L, dtype: int64

In [114]: ts[:2].resample('250L', fill_method='pad', limit=2)

2012-01-01 00:00:00           308
2012-01-01 00:00:00.250000    308
2012-01-01 00:00:00.500000    308
2012-01-01 00:00:00.750000    NaN
2012-01-01 00:00:01           204
Freq: 250L, dtype: float64
```

Parameters like `label` and `loffset` are used to manipulate the resulting labels. `label` specifies whether the result is labeled with the beginning or the end of the interval. `loffset` performs a time adjustment on the output labels.

```
In [115]: ts.resample('5Min') # by default label='right'

2012-01-01    251.03
Freq: 5T, dtype: float64

In [116]: ts.resample('5Min', label='left')

2012-01-01    251.03
Freq: 5T, dtype: float64

In [117]: ts.resample('5Min', label='left', loffset='1s')

2012-01-01 00:00:01    251.03
dtype: float64
```

The `axis` parameter can be set to 0 or 1 and allows you to resample the specified axis for a DataFrame.

`kind` can be set to 'timestamp' or 'period' to convert the resulting index to/from time-stamp and time-span representations. By default `resample` retains the input representation.

`convention` can be set to 'start' or 'end' when resampling period data (detail below). It specifies how low frequency periods are converted to higher frequency periods.

Note that 0.8 marks a watershed in the timeseries functionality in pandas. In previous versions, resampling had to be done using a combination of `date_range`, `groupby` with `asof`, and then calling an aggregation function on the grouped object. This was not nearly convenient or performant as the new pandas timeseries API.

15.8 Time Span Representation

Regular intervals of time are represented by `Period` objects in pandas while sequences of `Period` objects are collected in a `PeriodIndex`, which can be created with the convenience function `period_range`.

15.8.1 Period

A `Period` represents a span of time (e.g., a day, a month, a quarter, etc). It can be created using a frequency alias:

```
In [118]: Period('2012', freq='A-DEC')
Period('2012', 'A-DEC')

In [119]: Period('2012-1-1', freq='D')
Period('2012-01-01', 'D')

In [120]: Period('2012-1-1 19:00', freq='H')
Period('2012-01-01 19:00', 'H')
```

Unlike time stamped data, pandas does not support frequencies at multiples of DateOffsets (e.g., '3Min') for periods.

Adding and subtracting integers from periods shifts the period by its own frequency.

```
In [121]: p = Period('2012', freq='A-DEC')

In [122]: p + 1
Period('2013', 'A-DEC')

In [123]: p - 3
Period('2009', 'A-DEC')
```

Taking the difference of Period instances with the same frequency will return the number of frequency units between them:

```
In [124]: Period('2012', freq='A-DEC') - Period('2002', freq='A-DEC')
10
```

15.8.2 PeriodIndex and period_range

Regular sequences of Period objects can be collected in a PeriodIndex, which can be constructed using the period_range convenience function:

```
In [125]: prng = period_range('1/1/2011', '1/1/2012', freq='M')

In [126]: prng

<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2012-01]
length: 13
```

The PeriodIndex constructor can also be used directly:

```
In [127]: PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')

<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2011-03]
length: 3
```

Just like DatetimeIndex, a PeriodIndex can also be used to index pandas objects:

```
In [128]: Series(randn(len(prng)), prng)

2011-01   -0.253355
2011-02   -1.426908
2011-03    1.548971
2011-04   -0.088718
2011-05   -1.771348
2011-06   -0.989328
2011-07   -1.584789
2011-08   -0.288786
2011-09   -2.029806
2011-10   -0.761200
2011-11   -1.603608
2011-12    1.756171
2012-01    0.256502
Freq: M, dtype: float64
```

15.8.3 Frequency Conversion and Resampling with PeriodIndex

The frequency of Periods and PeriodIndex can be converted via the asfreq method. Let's start with the fiscal year 2011, ending in December:

```
In [129]: p = Period('2011', freq='A-DEC')

In [130]: p
Period('2011', 'A-DEC')
```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```
In [131]: p.asfreq('M', how='start')
Period('2011-01', 'M')

In [132]: p.asfreq('M', how='end')
Period('2011-12', 'M')
```

The shorthands 's' and 'e' are provided for convenience:

```
In [133]: p.asfreq('M', 's')
Period('2011-01', 'M')

In [134]: p.asfreq('M', 'e')
Period('2011-12', 'M')
```

Converting to a "super-period" (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```
In [135]: p = Period('2011-12', freq='M')

In [136]: p.asfreq('A-NOV')
Period('2012', 'A-NOV')
```

Note that since we converted to an annual frequency that ends the year in November, the monthly period of December 2011 is actually in the 2012 A-NOV period. Period conversions with anchored frequencies are particularly useful for working with various quarterly data common to economics, business, and other fields. Many organizations define quarters relative to the month in which their fiscal year start and ends. Thus, first quarter of 2011 could start in 2010 or a few months into 2011. Via anchored frequencies, pandas works all quarterly frequencies `Q-JAN` through `Q-DEC`.

`Q-DEC` define regular calendar quarters:

```
In [137]: p = Period('2012Q1', freq='Q-DEC')

In [138]: p.asfreq('D', 's')
Period('2012-01-01', 'D')

In [139]: p.asfreq('D', 'e')
Period('2012-03-31', 'D')
```

`Q-MAR` defines fiscal year end in March:

```
In [140]: p = Period('2011Q4', freq='Q-MAR')

In [141]: p.asfreq('D', 's')
Period('2011-01-01', 'D')

In [142]: p.asfreq('D', 'e')
Period('2011-03-31', 'D')
```

15.9 Converting between Representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```
In [143]: rng = date_range('1/1/2012', periods=5, freq='M')

In [144]: ts = Series(randn(len(rng)), index=rng)

In [145]: ts

2012-01-31    0.020601
2012-02-29   -0.411719
2012-03-31    2.079413
2012-04-30   -1.077911
2012-05-31    0.099258
Freq: M, dtype: float64

In [146]: ps = ts.to_period()

In [147]: ps

2012-01    0.020601
2012-02   -0.411719
2012-03    2.079413
2012-04   -1.077911
2012-05    0.099258
Freq: M, dtype: float64

In [148]: ps.to_timestamp()

2012-01-01    0.020601
2012-02-01   -0.411719
2012-03-01    2.079413
2012-04-01   -1.077911
2012-05-01    0.099258
Freq: MS, dtype: float64
```

Remember that 's' and 'e' can be used to return the timestamps at the start or end of the period:

```
In [149]: ps.to_timestamp('D', how='s')

2012-01-01    0.020601
2012-02-01   -0.411719
2012-03-01    2.079413
2012-04-01   -1.077911
2012-05-01    0.099258
Freq: MS, dtype: float64
```

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

```
In [150]: prng = period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [151]: ts = Series(randn(len(prng)), prng)

In [152]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [153]: ts.head()

1990-03-01 09:00   -0.089851
1990-06-01 09:00    0.711329
1990-09-01 09:00    0.531761
```

```
1990-12-01 09:00    0.265615
1991-03-01 09:00   -0.174462
Freq: H, dtype: float64
```

15.10 Time Zone Handling

Using `pytz`, pandas provides rich support for working with timestamps in different time zones. By default, pandas objects are time zone unaware:

```
In [154]: rng = date_range('3/6/2012 00:00', periods=15, freq='D')

In [155]: print(rng.tz)
None
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions:

```
In [156]: rng_utc = date_range('3/6/2012 00:00', periods=10, freq='D', tz='UTC')

In [157]: print(rng_utc.tz)
UTC
```

Timestamps, like Python's `datetime.datetime` object can be either time zone naive or time zone aware. Naive time series and DatetimeIndex objects can be *localized* using `tz_localize`:

```
In [158]: ts = Series(randn(len(rng)), rng)

In [159]: ts_utc = ts.tz_localize('UTC')

In [160]: ts_utc

2012-03-06 00:00:00+00:00   -2.189293
2012-03-07 00:00:00+00:00   -1.819506
2012-03-08 00:00:00+00:00    0.229798
2012-03-09 00:00:00+00:00    0.119425
2012-03-10 00:00:00+00:00    1.808966
2012-03-11 00:00:00+00:00    1.015841
2012-03-12 00:00:00+00:00   -1.651784
2012-03-13 00:00:00+00:00    0.347674
2012-03-14 00:00:00+00:00   -0.773688
2012-03-15 00:00:00+00:00    0.425863
2012-03-16 00:00:00+00:00    0.579486
2012-03-17 00:00:00+00:00   -0.745396
2012-03-18 00:00:00+00:00    0.141880
2012-03-19 00:00:00+00:00   -1.077754
2012-03-20 00:00:00+00:00   -1.301174
Freq: D, dtype: float64
```

You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```
In [161]: ts_utc.tz_convert('US/Eastern')

2012-03-05 19:00:00-05:00   -2.189293
2012-03-06 19:00:00-05:00   -1.819506
2012-03-07 19:00:00-05:00    0.229798
2012-03-08 19:00:00-05:00    0.119425
2012-03-09 19:00:00-05:00    1.808966
2012-03-10 19:00:00-05:00    1.015841
```

```
2012-03-11 20:00:00-04:00   -1.651784
2012-03-12 20:00:00-04:00    0.347674
2012-03-13 20:00:00-04:00   -0.773688
2012-03-14 20:00:00-04:00    0.425863
2012-03-15 20:00:00-04:00    0.579486
2012-03-16 20:00:00-04:00   -0.745396
2012-03-17 20:00:00-04:00    0.141880
2012-03-18 20:00:00-04:00   -1.077754
2012-03-19 20:00:00-04:00   -1.301174
Freq: D, dtype: float64
```

Under the hood, all timestamps are stored in UTC. Scalar values from a `DatetimeIndex` with a time zone will have their fields (day, hour, minute) localized to the time zone. However, timestamps with the same UTC value are still considered to be equal even if they are in different time zones:

```
In [162]: rng_eastern = rng_utc.tz_convert('US/Eastern')

In [163]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [164]: rng_eastern[5]
Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [165]: rng_berlin[5]
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

In [166]: rng_eastern[5] == rng_berlin[5]
True
```

Like Series, DataFrame, and DatetimeIndex, Timestamps can be converted to other time zones using `tz_convert`:

```
In [167]: rng_eastern[5]
Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern')

In [168]: rng_berlin[5]
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')

In [169]: rng_eastern[5].tz_convert('Europe/Berlin')
Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of Timestamps functions just like DatetimeIndex and TimeSeries:

```
In [170]: rng[5]
Timestamp('2012-03-11 00:00:00', tz=None)

In [171]: rng[5].tz_localize('Asia/Shanghai')
Timestamp('2012-03-11 00:00:00+0800', tz='Asia/Shanghai')
```

Operations between TimeSeries in difficult time zones will yield UTC TimeSeries, aligning the data on the UTC timestamps:

```
In [172]: eastern = ts_utc.tz_convert('US/Eastern')

In [173]: berlin = ts_utc.tz_convert('Europe/Berlin')

In [174]: result = eastern + berlin

In [175]: result

2012-03-06 00:00:00+00:00   -4.378586
```

```
2012-03-07 00:00:00+00:00   -3.639011
2012-03-08 00:00:00+00:00    0.459596
2012-03-09 00:00:00+00:00    0.238849
2012-03-10 00:00:00+00:00    3.617932
2012-03-11 00:00:00+00:00    2.031683
2012-03-12 00:00:00+00:00   -3.303568
2012-03-13 00:00:00+00:00    0.695349
2012-03-14 00:00:00+00:00   -1.547376
2012-03-15 00:00:00+00:00    0.851726
2012-03-16 00:00:00+00:00    1.158971
2012-03-17 00:00:00+00:00   -1.490793
2012-03-18 00:00:00+00:00    0.283760
2012-03-19 00:00:00+00:00   -2.155508
2012-03-20 00:00:00+00:00   -2.602348
Freq: D, dtype: float64
```

```
In [176]: result.index
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-06 00:00:00, ..., 2012-03-20 00:00:00]
Length: 15, Freq: D, Timezone: UTC
```

15.11 Time Deltas

Timedeltas are differences in times, expressed in difference units, e.g. days,hours,minutes,seconds. They can be both positive and negative.

```
In [177]: from datetime import datetime, timedelta

In [178]: s  = Series(date_range('2012-1-1', periods=3, freq='D'))

In [179]: td = Series([ timedelta(days=i) for i in range(3) ])

In [180]: df = DataFrame(dict(A = s, B = td))

In [181]: df
```

```
                   A               B
0 2012-01-01 00:00:00         00:00:00
1 2012-01-02 00:00:00 1 days, 00:00:00
2 2012-01-03 00:00:00 2 days, 00:00:00
```

```
In [182]: df['C'] = df['A'] + df['B']

In [183]: df
```

```
                   A               B                   C
0 2012-01-01 00:00:00         00:00:00 2012-01-01 00:00:00
1 2012-01-02 00:00:00 1 days, 00:00:00 2012-01-03 00:00:00
2 2012-01-03 00:00:00 2 days, 00:00:00 2012-01-05 00:00:00
```

```
In [184]: df.dtypes
```

```
A     datetime64[ns]
B    timedelta64[ns]
C     datetime64[ns]
```

```
dtype: object

In [185]: s - s.max()

0   -2 days, 00:00:00
1   -1 days, 00:00:00
2          00:00:00
dtype: timedelta64[ns]

In [186]: s - datetime(2011,1,1,3,5)

0   364 days, 20:55:00
1   365 days, 20:55:00
2   366 days, 20:55:00
dtype: timedelta64[ns]

In [187]: s + timedelta(minutes=5)

0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

Getting scalar results from a `timedelta64[ns]` series

```
In [188]: y = s - s[0]

In [189]: y

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

if LooseVersion(np.__version__) <= '1.6.2':
    y.apply(lambda x: x.item().total_seconds())
    y.apply(lambda x: x.item().days)
else:
    y.apply(lambda x: x / np.timedelta64(1, 's'))
    y.apply(lambda x: x / np.timedelta64(1, 'D'))
```

Note: As you can see from the conditional statement above, these operations are different in numpy 1.6.2 and in numpy >= 1.7. The `timedelta64[ns]` scalar type in 1.6.2 is much like a `datetime.timedelta`, while in 1.7 it is a nanosecond based integer. A future version of pandas will make this transparent.

Note: In numpy >= 1.7 dividing a `timedelta64` array by another `timedelta64` array will yield an array with dtype `np.float64`.

Series of timedeltas with `NaT` values are supported

```
In [190]: y = s - s.shift()

In [191]: y

0                 NaT
1   1 days, 00:00:00
```

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

Elements can be set to `NaT` using `np.nan` analagously to datetimes

```
In [192]: y[1] = np.nan

In [193]: y

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

Operands can also appear in a reversed order (a singluar object operated with a Series)

```
In [194]: s.max() - s

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

In [195]: datetime(2011,1,1,3,5) - s

0   -364 days, 20:55:00
1   -365 days, 20:55:00
2   -366 days, 20:55:00
dtype: timedelta64[ns]

In [196]: timedelta(minutes=5) + s

0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

Some timedelta numeric like operations are supported.

```
In [197]: td - timedelta(minutes=5, seconds=5, microseconds=5)

0         -00:05:05.000005
1          23:54:54.999995
2   1 days, 23:54:54.999995
dtype: timedelta64[ns]
```

`min, max` and the corresponding `idxmin, idxmax` operations are supported on frames

```
In [198]: A = s - Timestamp('20120101') - timedelta(minutes=5, seconds=5)

In [199]: B = s - Series(date_range('2012-1-2', periods=3, freq='D'))

In [200]: df = DataFrame(dict(A=A, B=B))

In [201]: df

                A                   B
0       -00:05:05 -1 days, 00:00:00
1        23:54:55 -1 days, 00:00:00
```

```
2 1 days, 23:54:55 -1 days, 00:00:00

In [202]: df.min()

A           -00:05:05
B   -1 days, 00:00:00
dtype: timedelta64[ns]

In [203]: df.min(axis=1)

0   -1 days, 00:00:00
1   -1 days, 00:00:00
2   -1 days, 00:00:00
dtype: timedelta64[ns]

In [204]: df.idxmin()

A    0
B    0
dtype: int64

In [205]: df.idxmax()

A    2
B    0
dtype: int64
```

`min, max` operations are supported on series; these return a single element `timedelta64[ns]` Series (this avoids having to deal with numpy timedelta64 issues). `idxmin, idxmax` are supported as well.

```
In [206]: df.min().max()

0   -00:05:05
dtype: timedelta64[ns]

In [207]: df.min(axis=1).min()

0   -1 days, 00:00:00
dtype: timedelta64[ns]

In [208]: df.min().idxmax()
'A'

In [209]: df.min(axis=1).idxmin()
0
```

CHAPTER

SIXTEEN

PLOTTING WITH MATPLOTLIB

Note: We intend to build more plotting integration with matplotlib as time goes on.

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt
```

16.1 Basic plotting: `plot`

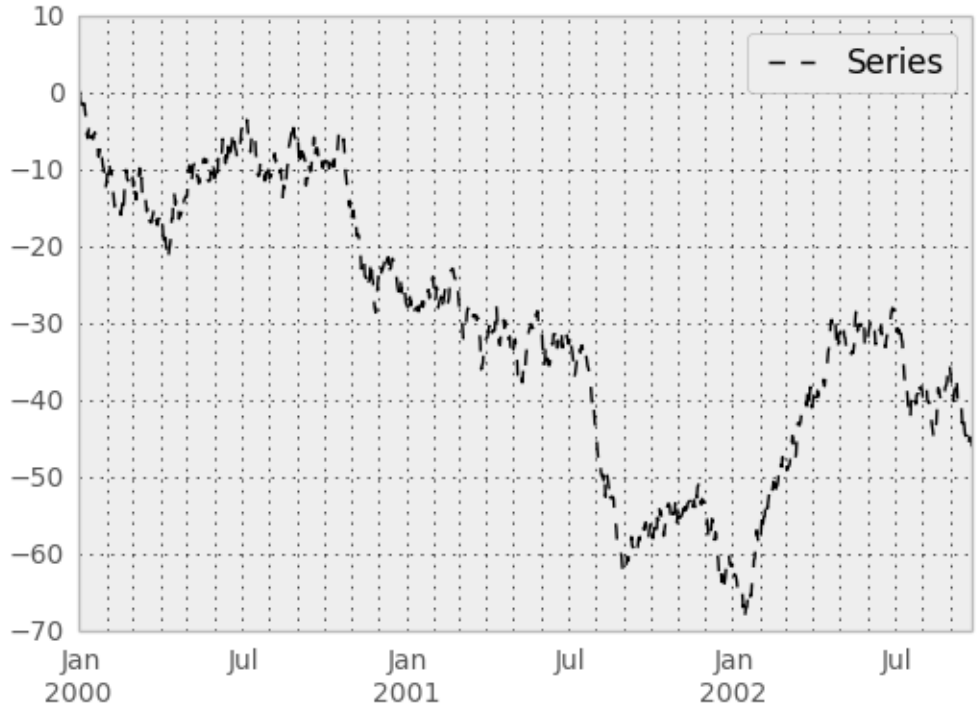
See the *cookbook* for some advanced strategies

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot`:

```
In [2]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [3]: ts = ts.cumsum()

In [4]: ts.plot()
<matplotlib.axes.AxesSubplot at 0x927d490>
```

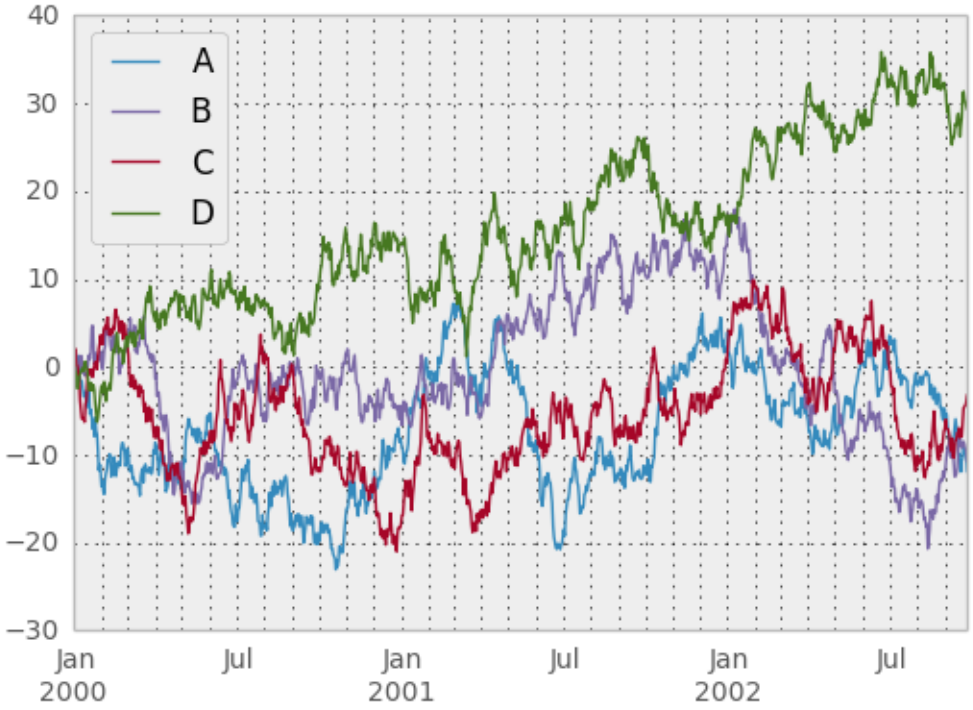
If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above. The method takes a number of arguments for controlling the look of the plot:

```
In [5]: plt.figure(); ts.plot(style='k--', label='Series'); plt.legend()
<matplotlib.legend.Legend at 0x6832bd0>
```



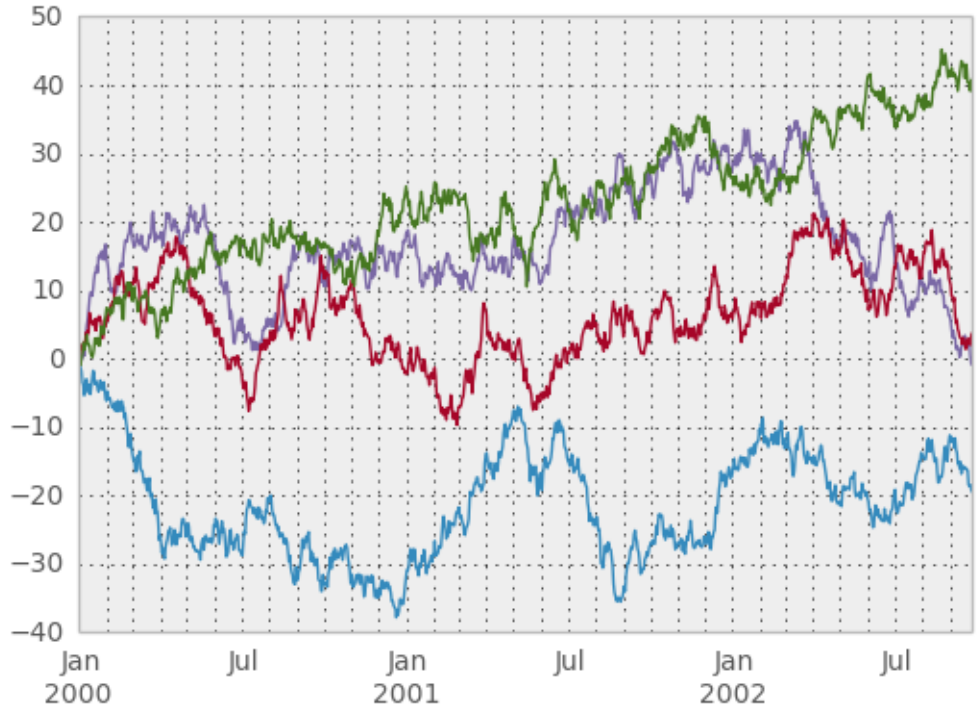
On DataFrame, `plot` is a convenience to plot all of the columns with labels:

```
In [6]: df = DataFrame(randn(1000, 4), index=ts.index, columns=list('ABCD'))

In [7]: df = df.cumsum()

In [8]: plt.figure(); df.plot(); plt.legend(loc='best')
<matplotlib.legend.Legend at 0x78d8b10>
```



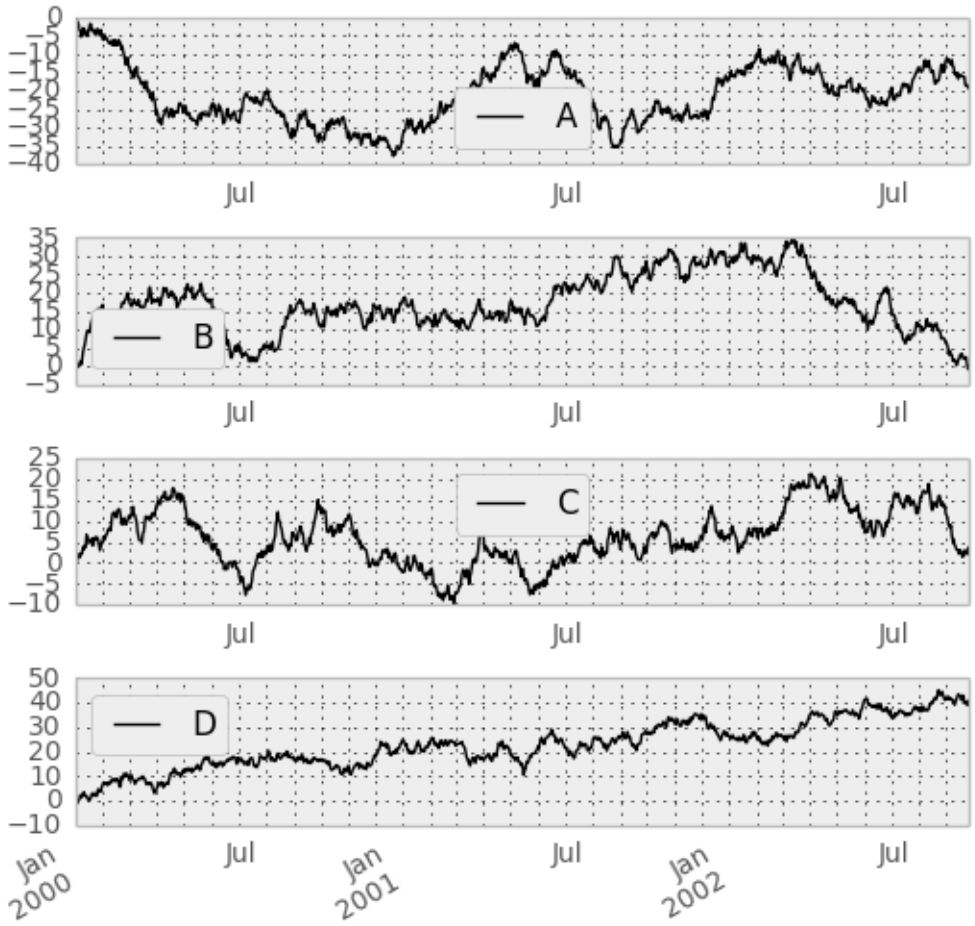
You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```
In [9]: df.plot(legend=False)
<matplotlib.axes.AxesSubplot at 0x78eca90>
```



Some other options are available, like plotting each Series on a different axis:

```
In [10]: df.plot(subplots=True, figsize=(6, 6)); plt.legend(loc='best')
<matplotlib.legend.Legend at 0xaa1a810>
```

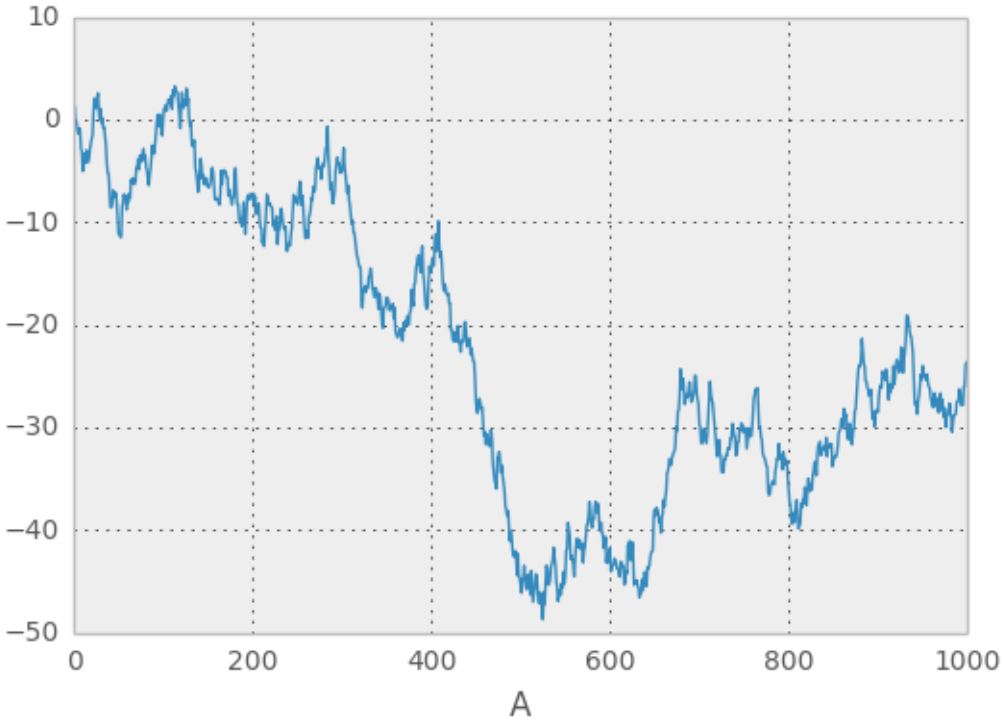


You may pass `logy` to get a log-scale Y axis.

```
In [11]: plt.figure();
In [11]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [12]: ts = np.exp(ts.cumsum())

In [13]: ts.plot(logy=True)
<matplotlib.axes.AxesSubplot at 0x907da10>
```

You can plot one column versus another using the *x* and *y* keywords in *DataFrame.plot*:

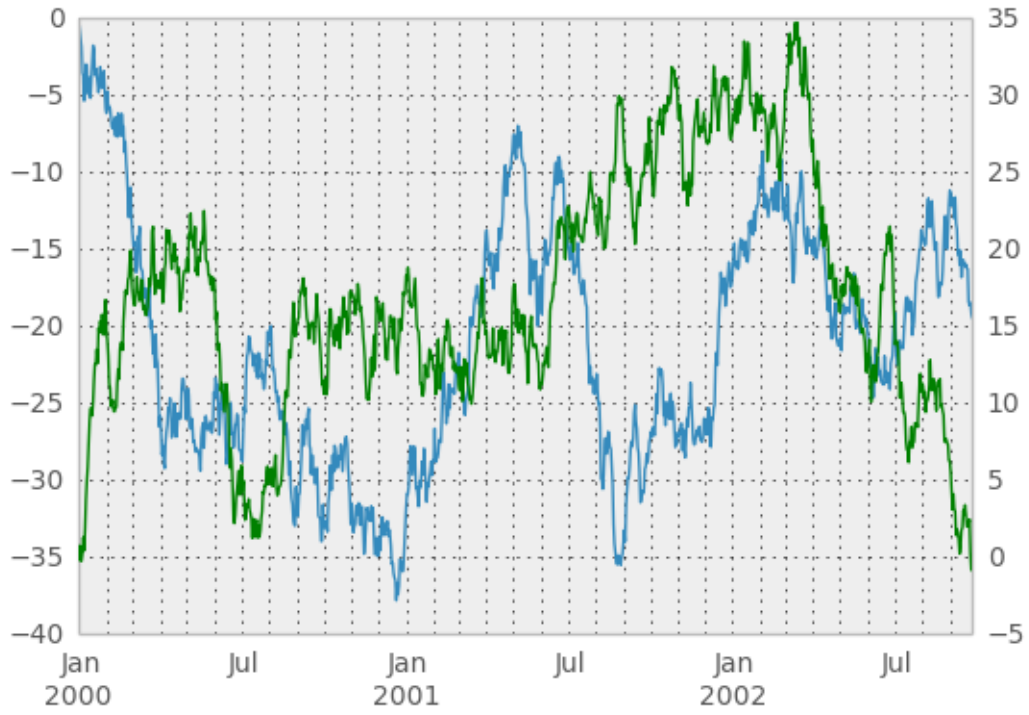
```
In [14]: plt.figure()
<matplotlib.figure.Figure at 0x8de3450>

In [15]: df3 = DataFrame(randn(1000, 2), columns=['B', 'C']).cumsum()

In [16]: df3['A'] = Series(range(len(df)))

In [17]: df3.plot(x='A', y='B')
<matplotlib.axes.AxesSubplot at 0x879d850>
```



16.1.1 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

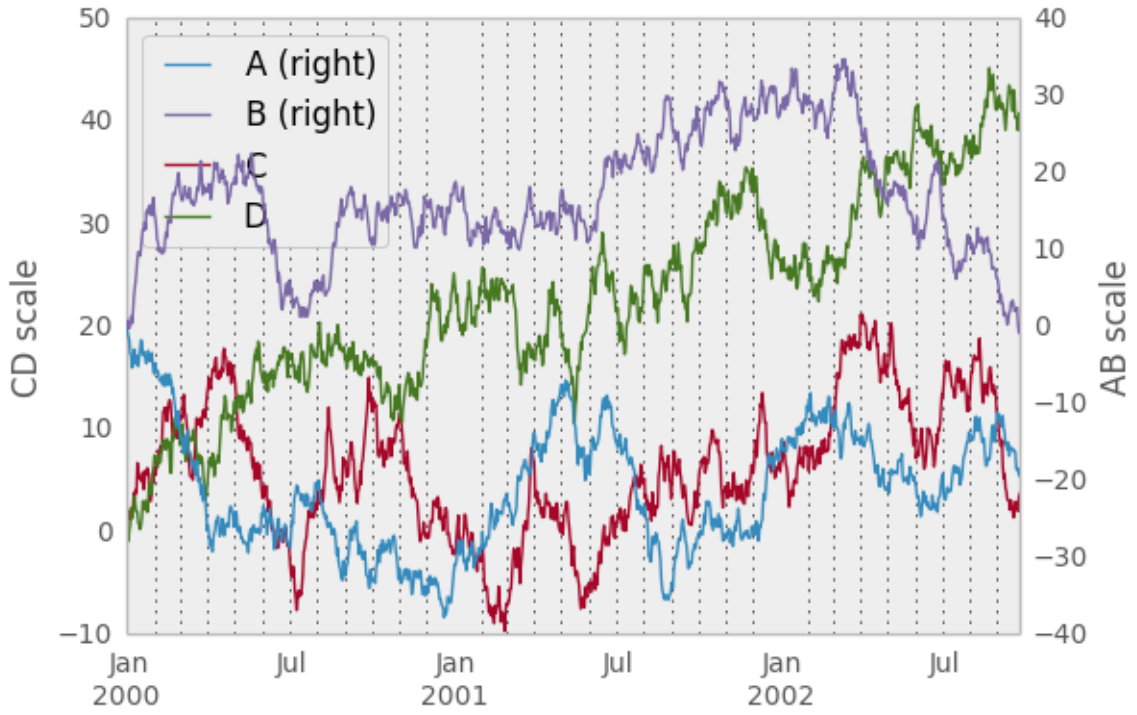
```
In [18]: plt.figure()
<matplotlib.figure.Figure at 0x75ea810>

In [19]: df.A.plot()
<matplotlib.axes.AxesSubplot at 0x7611190>

In [20]: df.B.plot(secondary_y=True, style='g')
<matplotlib.axes.AxesSubplot at 0x9c86510>
```



16.1.2 Selective Plotting on Secondary Y-axis

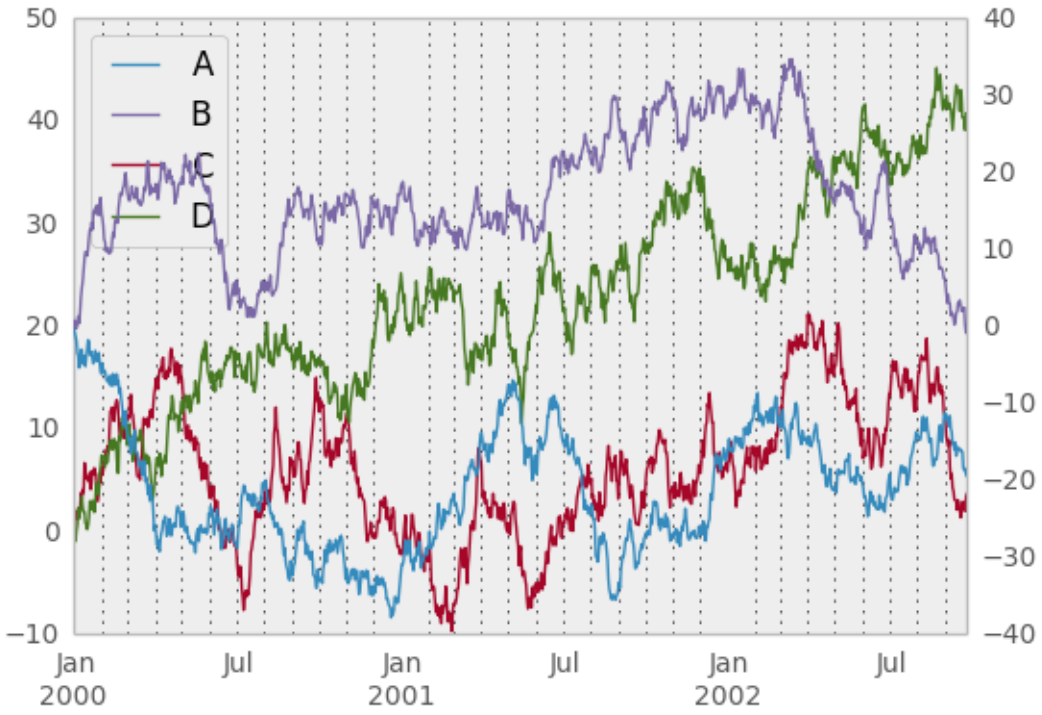
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```
In [21]: plt.figure()
<matplotlib.figure.Figure at 0x907c950>

In [22]: ax = df.plot(secondary_y=['A', 'B'])

In [23]: ax.set_ylabel('CD scale')
<matplotlib.text.Text at 0x9c67290>

In [24]: ax.right_ax.set_ylabel('AB scale')
<matplotlib.text.Text at 0x9f449d0>
```



Note that the columns plotted on the secondary y-axis is automatically marked with "(right)" in the legend. To turn off the automatic marking, use the `mark_right=False` keyword:

```
In [25]: plt.figure()
<matplotlib.figure.Figure at 0x9c55290>

In [26]: df.plot(secondary_y=['A', 'B'], mark_right=False)
<matplotlib.axes.AxesSubplot at 0x961c790>
```

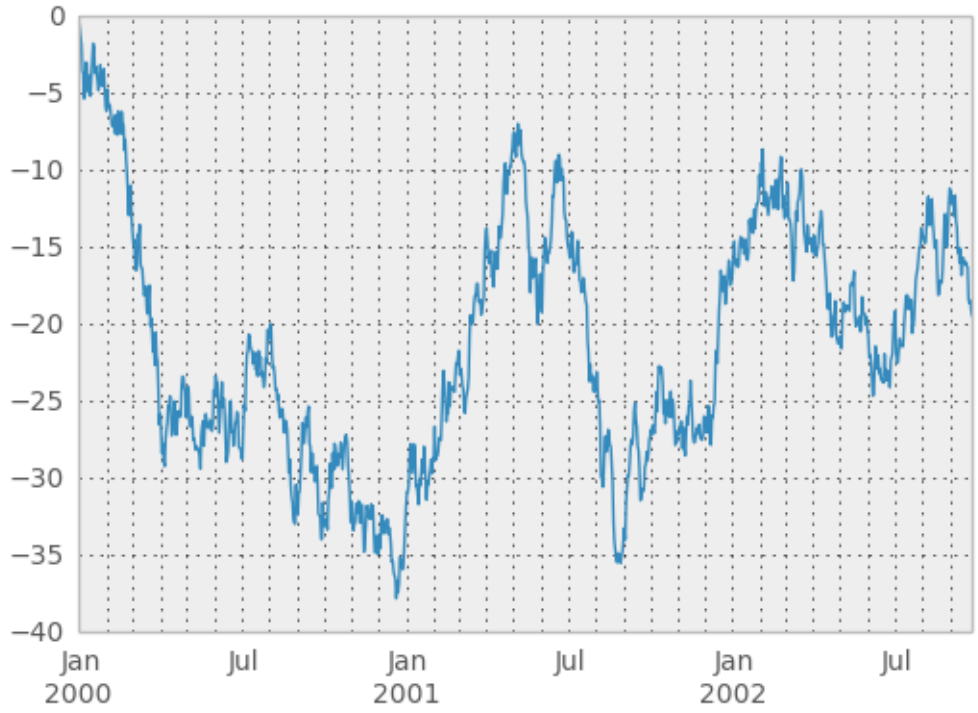


16.1.3 Suppressing tick resolution adjustment

Pandas includes automatically tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created `twinx`), you can choose to suppress this behavior for alignment purposes.

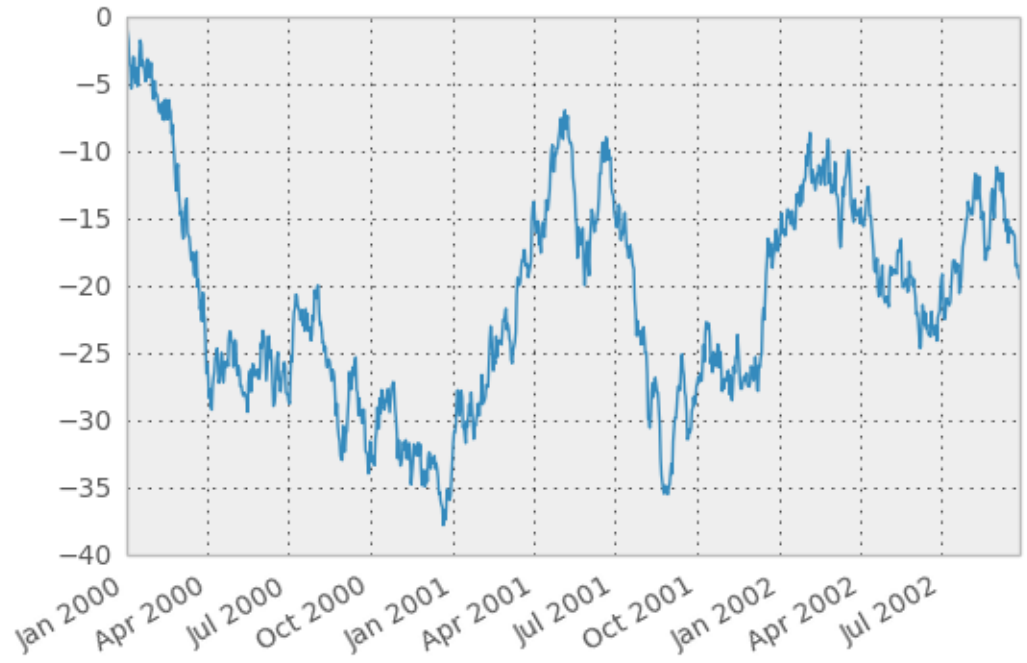
Here is the default behavior, notice how the x-axis tick labelling is performed:

```
In [27]: plt.figure()
<matplotlib.figure.Figure at 0x9617ad0>

In [28]: df.A.plot()
<matplotlib.axes.AxesSubplot at 0x961e210>
```



Using the `x_compat` parameter, you can suppress this behavior:

```
In [29]: plt.figure()
<matplotlib.figure.Figure at 0x9765ed0>

In [30]: df.A.plot(x_compat=True)
<matplotlib.axes.AxesSubplot at 0x9755f10>
```



If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plot_params` can be used

in a *with statement*:

```
In [31]: import pandas as pd

In [32]: plt.figure()
<matplotlib.figure.Figure at 0x8170ad0>

In [33]: with pd.plot_params.use('x_compat', True):
   ....:     df.A.plot(color='r')
   ....:     df.B.plot(color='g')
   ....:     df.C.plot(color='b')
   ....:
```

16.1.4 Targeting different subplots

You can pass an `ax` argument to `Series.plot` to plot on a particular axis:

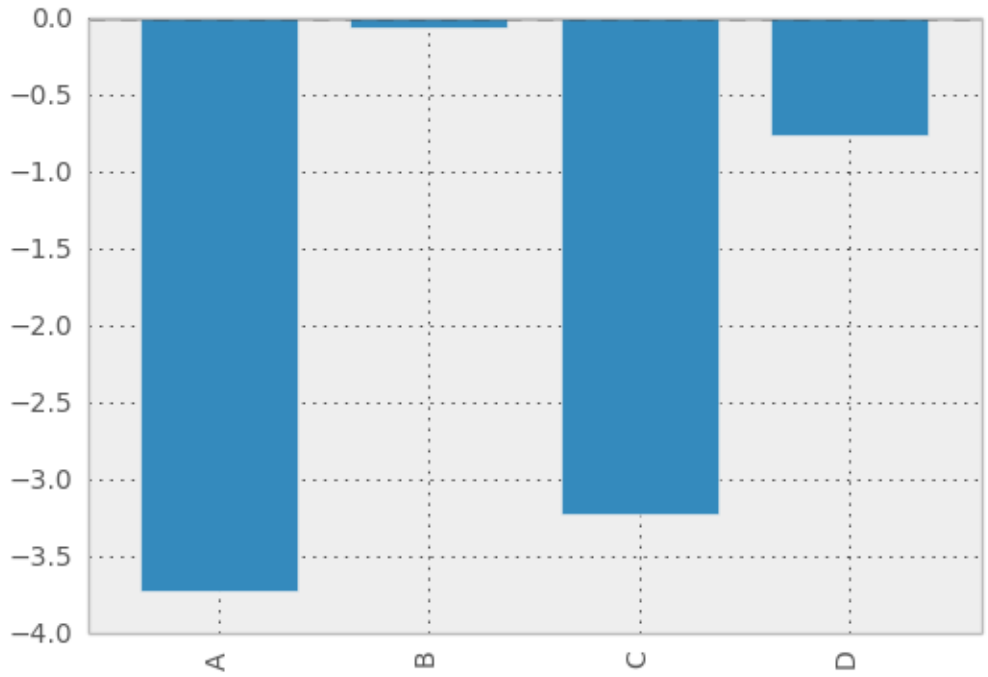
```
In [34]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [35]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A')
<matplotlib.text.Text at 0x987ac90>

In [36]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B')
<matplotlib.text.Text at 0x8a37850>

In [37]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C')
<matplotlib.text.Text at 0x81196d0>

In [38]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D')
<matplotlib.text.Text at 0x8106e90>
```

16.2 Other plotting features

16.2.1 Bar plots

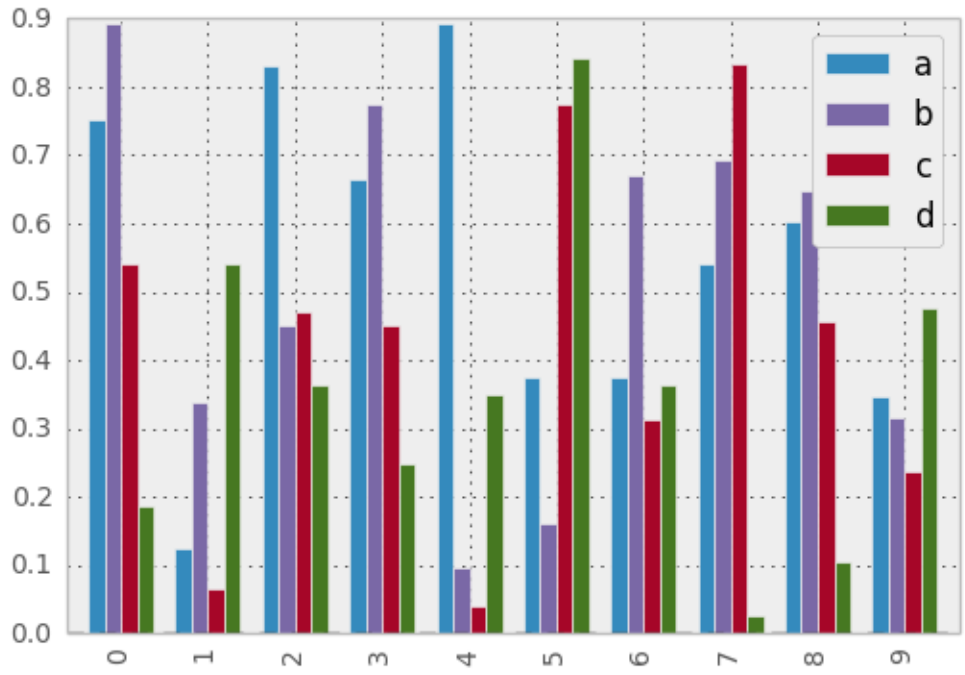
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [39]: plt.figure();
In [39]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
<matplotlib.lines.Line2D at 0xc46aed0>
```



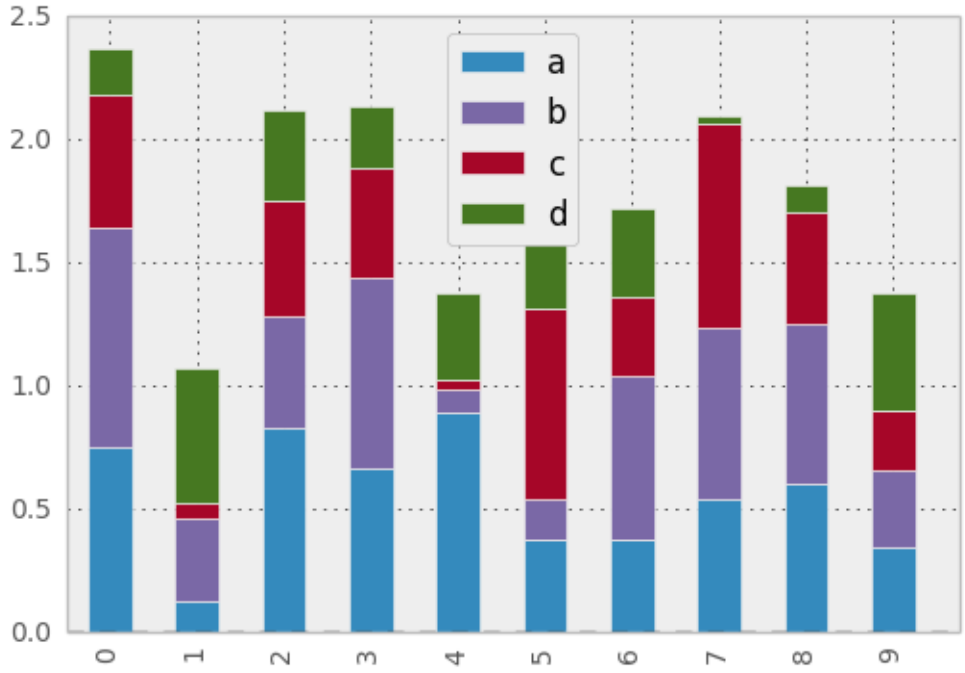
Calling a DataFrame's `plot` method with `kind='bar'` produces a multiple bar plot:

```
In [40]: df2 = DataFrame(rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [41]: df2.plot(kind='bar');
```



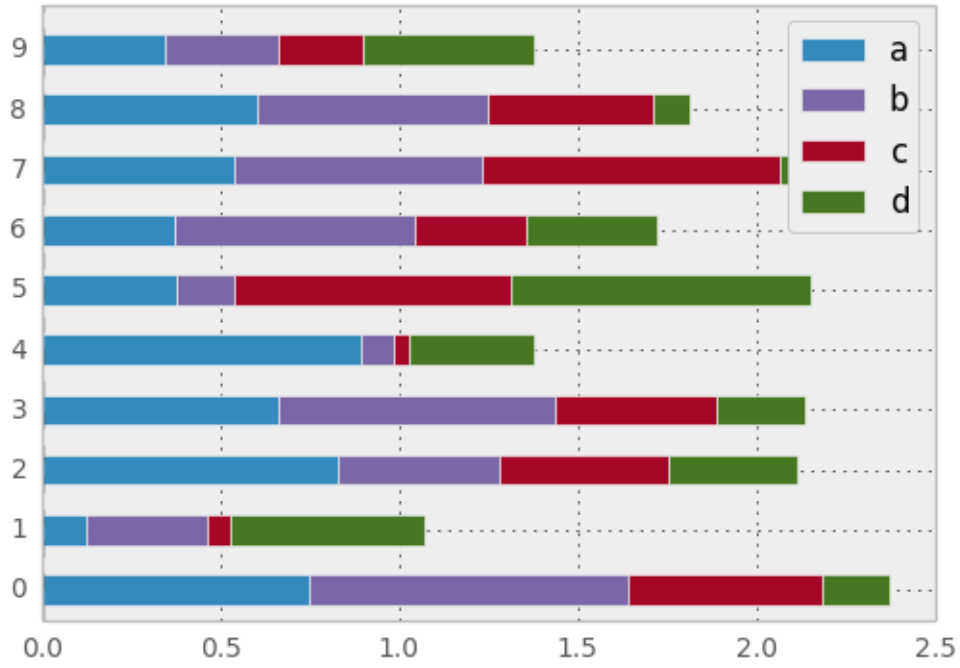
To produce a stacked bar plot, pass `stacked=True`:

```
In [41]: df2.plot(kind='bar', stacked=True);
```



To get horizontal bar plots, pass `kind='barh'`:

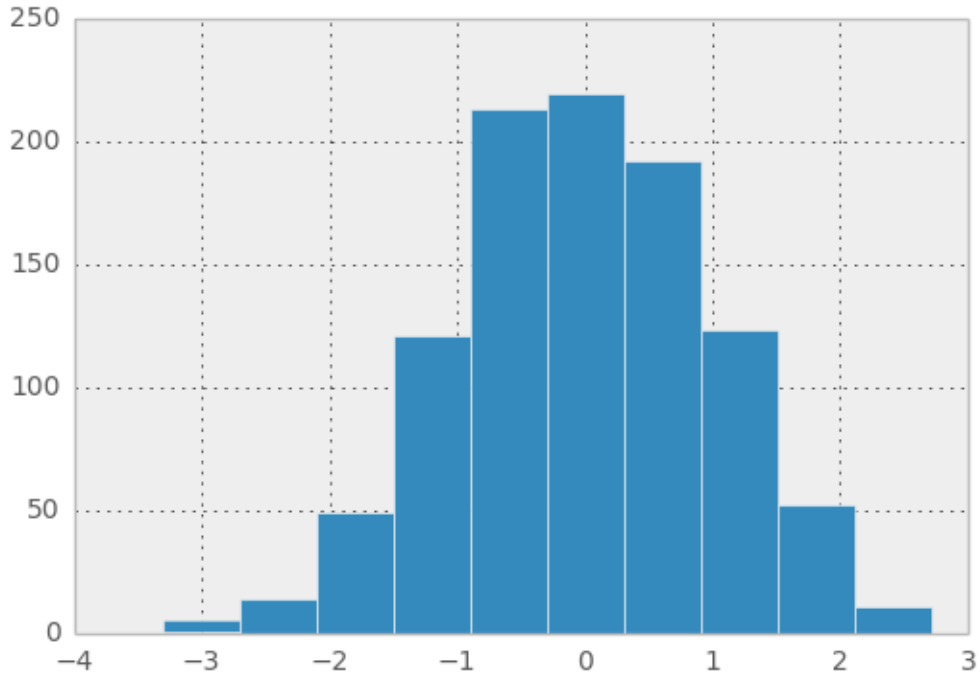
```
In [41]: df2.plot(kind='barh', stacked=True);
```



16.2.2 Histograms

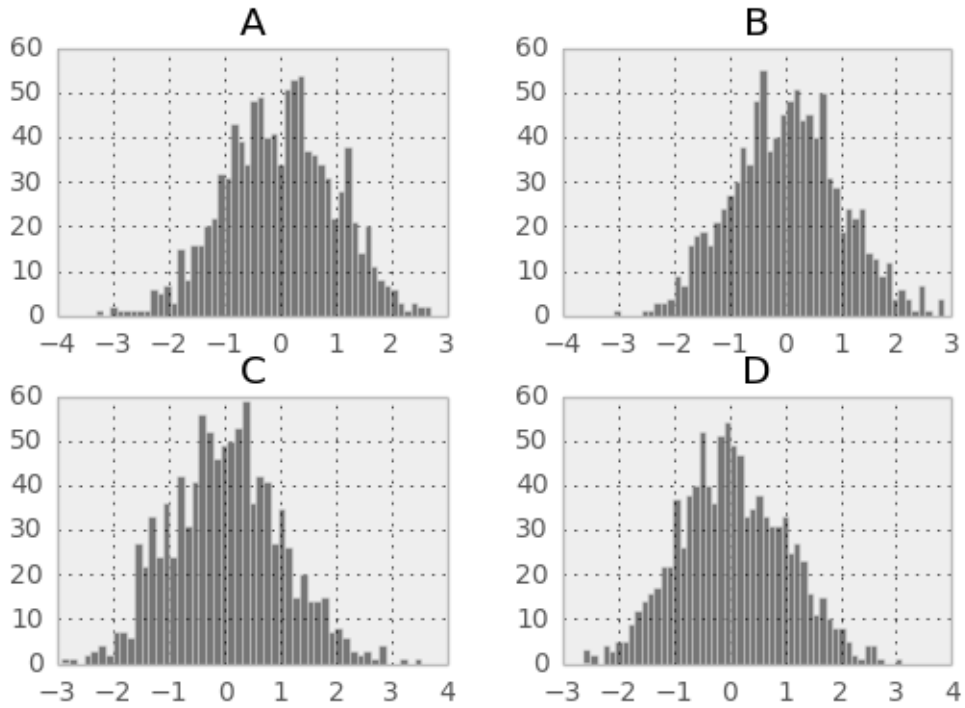
```
In [41]: plt.figure();
In [41]: df['A'].diff().hist()
```

```
<matplotlib.axes.AxesSubplot at 0xd02fd50>
```



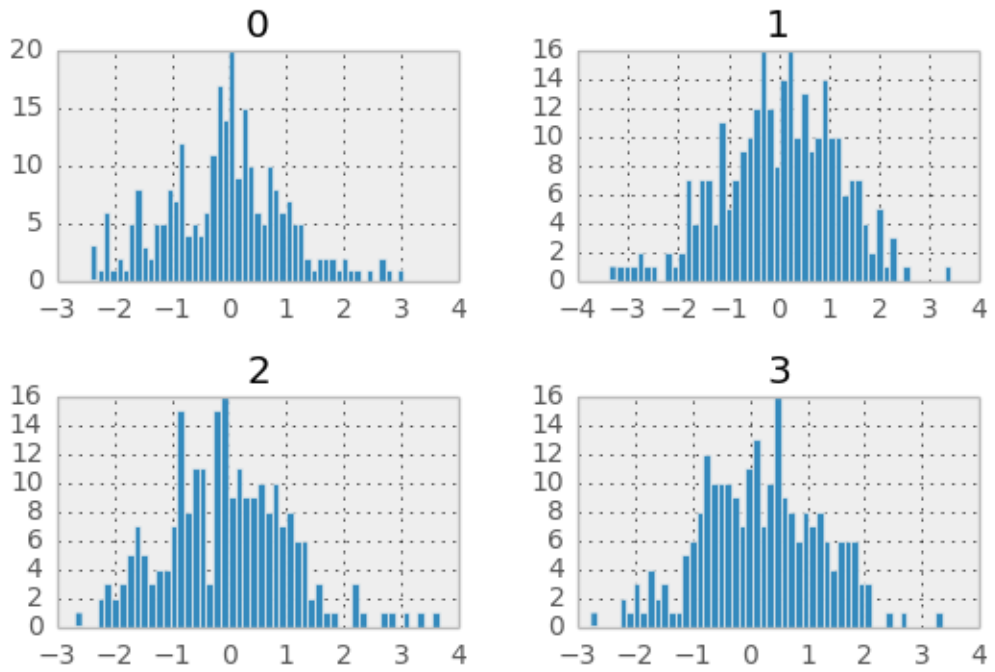
For a DataFrame, `hist` plots the histograms of the columns on multiple subplots:

```
In [42]: plt.figure()
<matplotlib.figure.Figure at 0xd438350>

In [43]: df.diff().hist(color='k', alpha=0.5, bins=50)

array([[<matplotlib.axes.AxesSubplot object at 0x8a29990>,
        <matplotlib.axes.AxesSubplot object at 0xd45a2d0>],
       [<matplotlib.axes.AxesSubplot object at 0xd805850>,
        <matplotlib.axes.AxesSubplot object at 0xd81ad10>]], dtype=object)
```



New since 0.10.0, the `by` keyword can be specified to plot grouped histograms:

```
In [44]: data = Series(randn(1000))

In [45]: data.hist(by=randint(0, 4, 1000), figsize=(6, 4))

array([[<matplotlib.axes.AxesSubplot object at 0xe1df790>,
        <matplotlib.axes.AxesSubplot object at 0xe0058d0>],
       [<matplotlib.axes.AxesSubplot object at 0xe01fbd0>,
        <matplotlib.axes.AxesSubplot object at 0xe5b6910>]], dtype=object)
```

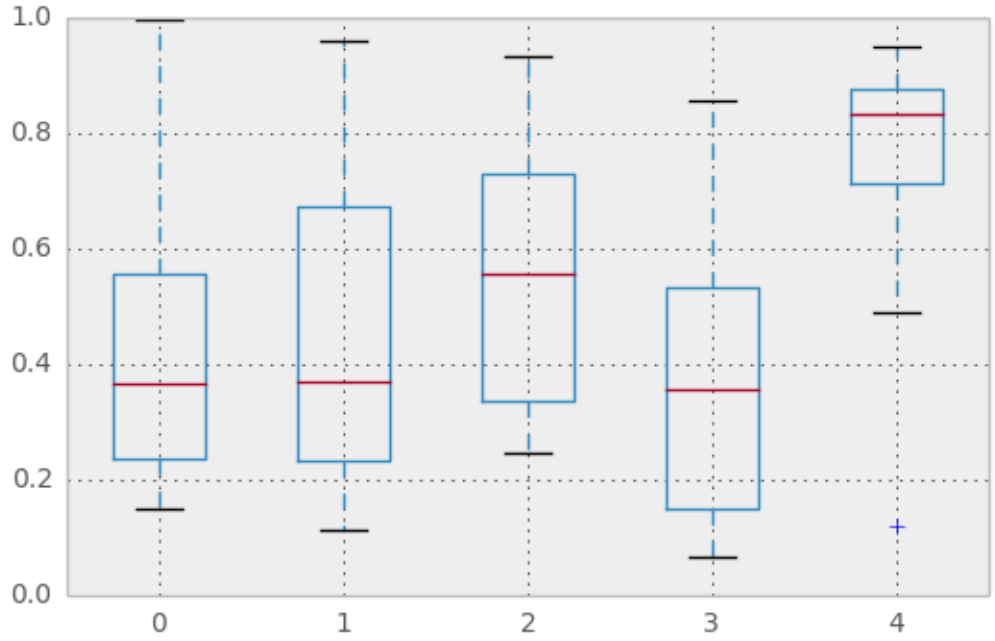


16.2.3 Box-Plotting

DataFrame has a `boxplot` method which allows you to visualize the distribution of values within each column.

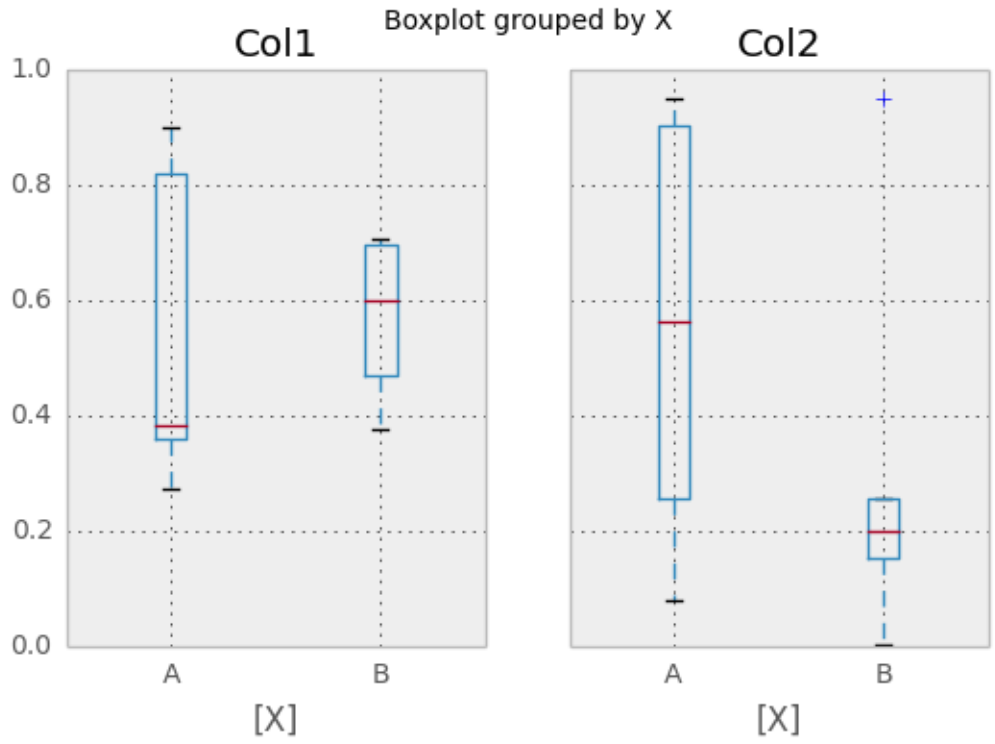
For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

```
In [46]: df = DataFrame(rand(10,5))

In [47]: plt.figure();
In [47]: bp = df.boxplot()
```



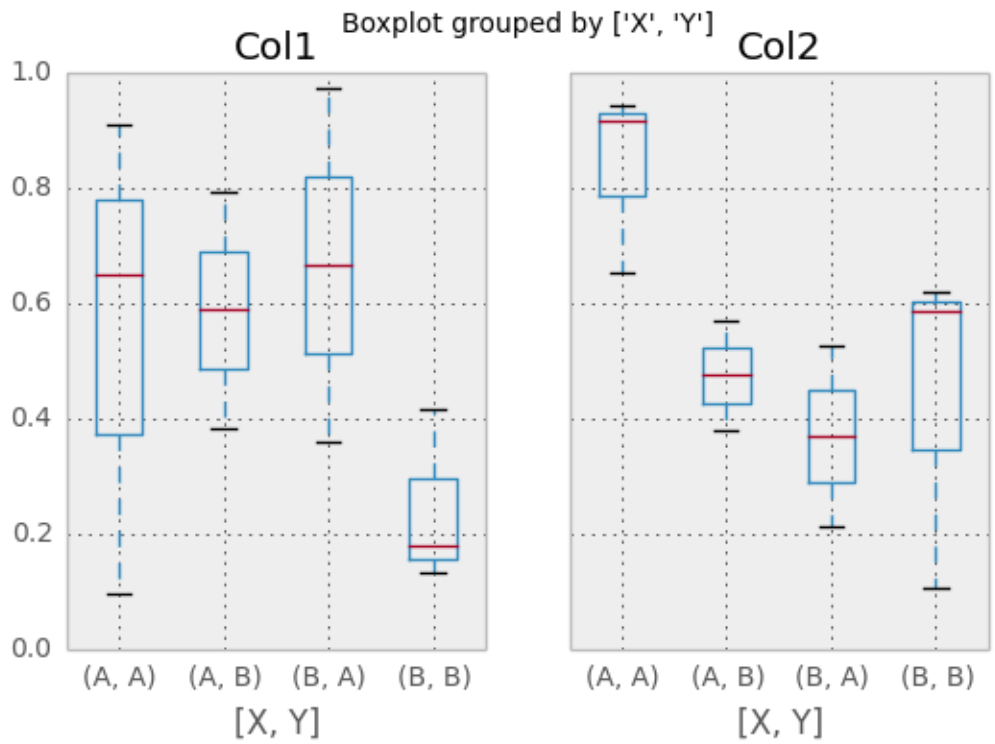
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [48]: df = DataFrame(rand(10,2), columns=['Col1', 'Col2'] )

In [49]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])

In [50]: plt.figure();
In [50]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [51]: df = DataFrame(rand(10,3), columns=['Col1', 'Col2', 'Col3'])

In [52]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])

In [53]: df['Y'] = Series(['A','B','A','B','A','B','A','B','A','B'])

In [54]: plt.figure();
In [54]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```

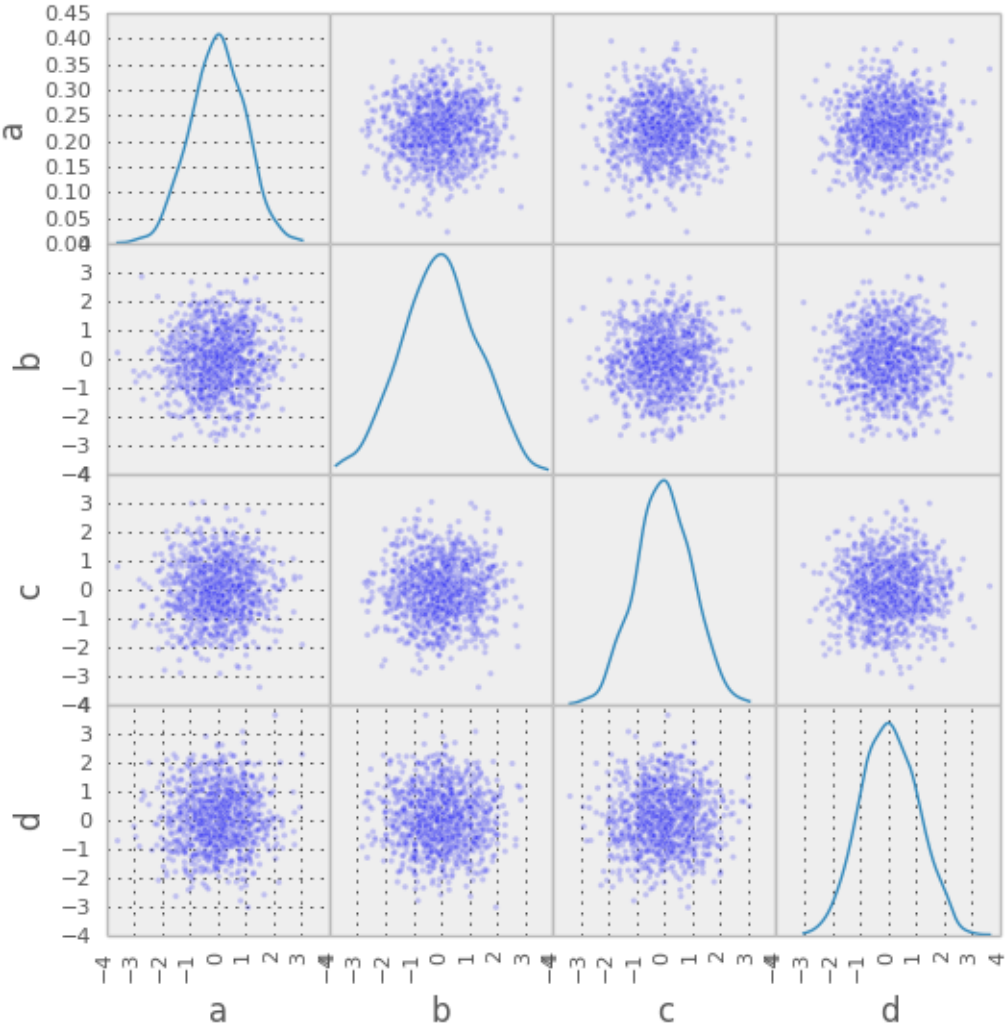


16.2.4 Scatter plot matrix

New in 0.7.3. **You can create a scatter plot matrix using the** `scatter_matrix` method in `pandas.tools.plotting`:

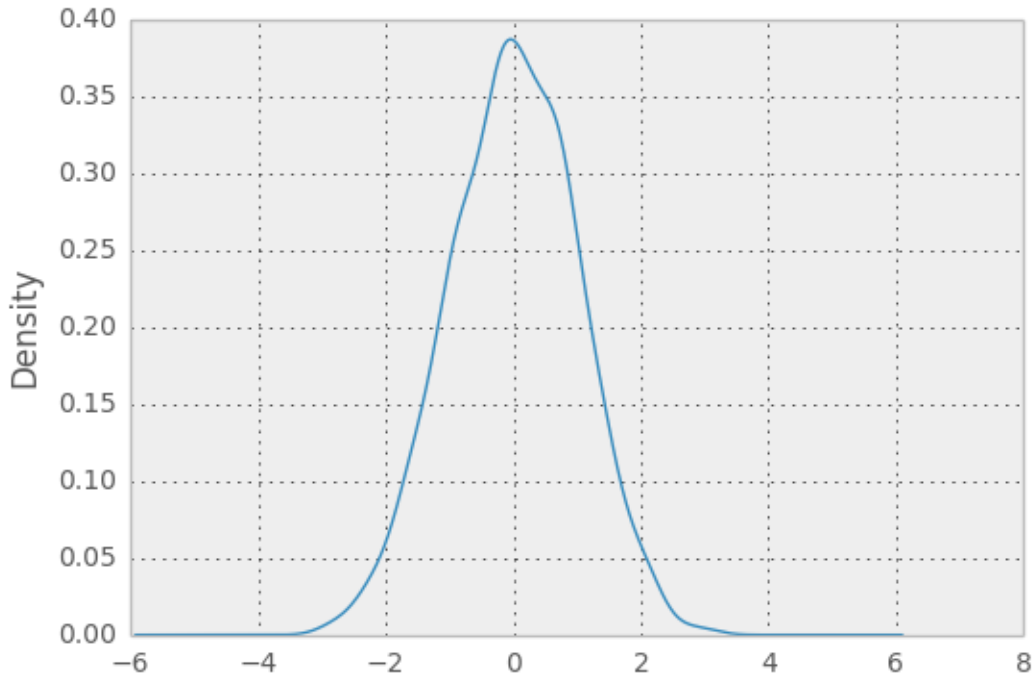
```
In [55]: from pandas.tools.plotting import scatter_matrix

In [56]: df = DataFrame(randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [57]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')

array([[<matplotlib.axes.AxesSubplot object at 0xfc5dfd0>,
        <matplotlib.axes.AxesSubplot object at 0xfbd9050>,
        <matplotlib.axes.AxesSubplot object at 0xfbfe5d0>,
        <matplotlib.axes.AxesSubplot object at 0xff2c4d0>],
       [<matplotlib.axes.AxesSubplot object at 0xff4edd0>,
        <matplotlib.axes.AxesSubplot object at 0x100a1910>,
        <matplotlib.axes.AxesSubplot object at 0x100d3610>,
        <matplotlib.axes.AxesSubplot object at 0x1022cbd0>],
       [<matplotlib.axes.AxesSubplot object at 0x10240f90>,
```

```
        <matplotlib.axes.AxesSubplot object at 0x103b78d0>,
        <matplotlib.axes.AxesSubplot object at 0x10519050>,
        <matplotlib.axes.AxesSubplot object at 0x10532d50>],
       [<matplotlib.axes.AxesSubplot object at 0x1069a910>,
        <matplotlib.axes.AxesSubplot object at 0x10543090>,
        <matplotlib.axes.AxesSubplot object at 0x10824310>,
        <matplotlib.axes.AxesSubplot object at 0x1084b7d0>]], dtype=object)
```



New in 0.8.0 You can create density plots using the Series/DataFrame.plot and setting *kind='kde'*:

```
In [58]: ser = Series(randn(1000))

In [59]: ser.plot(kind='kde')
<matplotlib.axes.AxesSubplot at 0x10b1eb50>
```

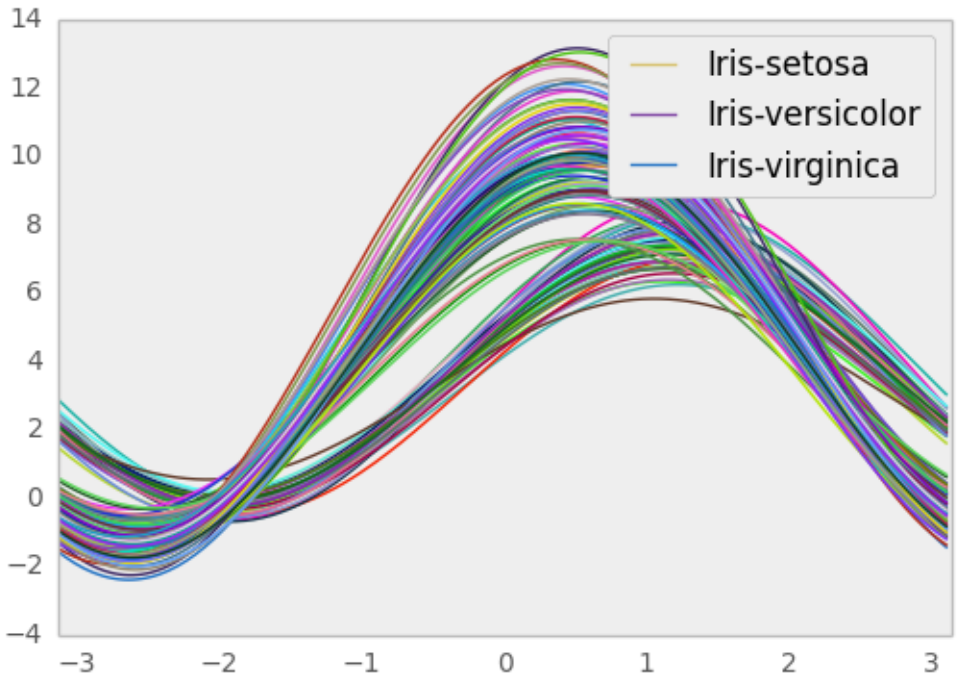


16.2.5 Andrews Curves

Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

Note: The "Iris" dataset is available here.

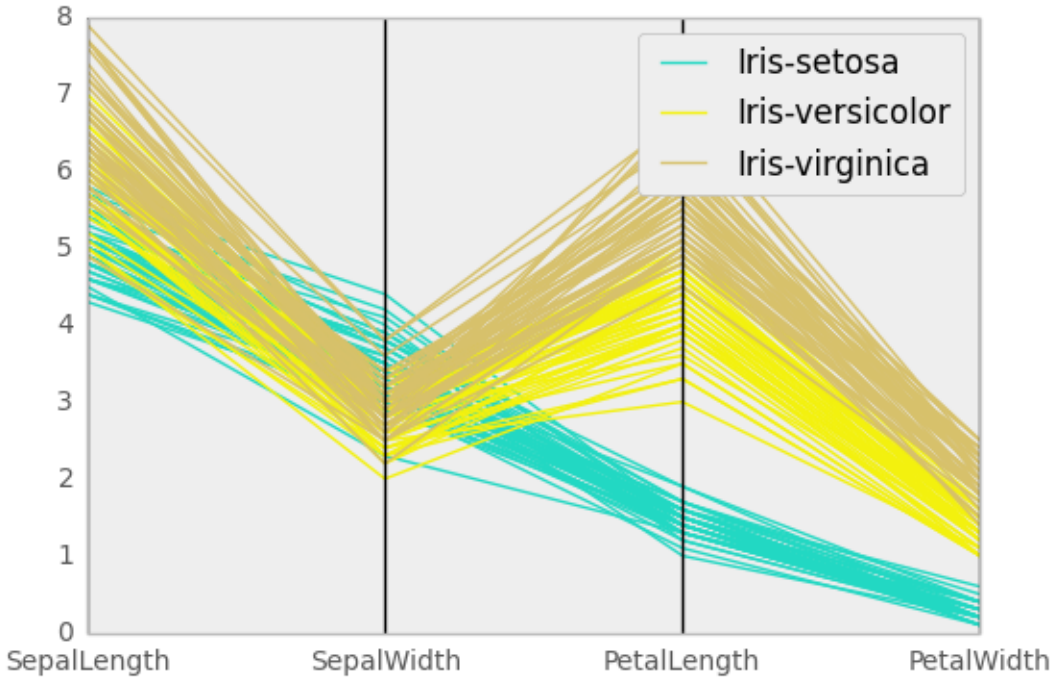
```
In [60]: from pandas import read_csv

In [61]: from pandas.tools.plotting import andrews_curves

In [62]: data = read_csv('data/iris.data')

In [63]: plt.figure()
<matplotlib.figure.Figure at 0x9f6a090>

In [64]: andrews_curves(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x9f6a690>
```



16.2.6 Parallel Coordinates

Parallel coordinates is a plotting technique for plotting multivariate data. It allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

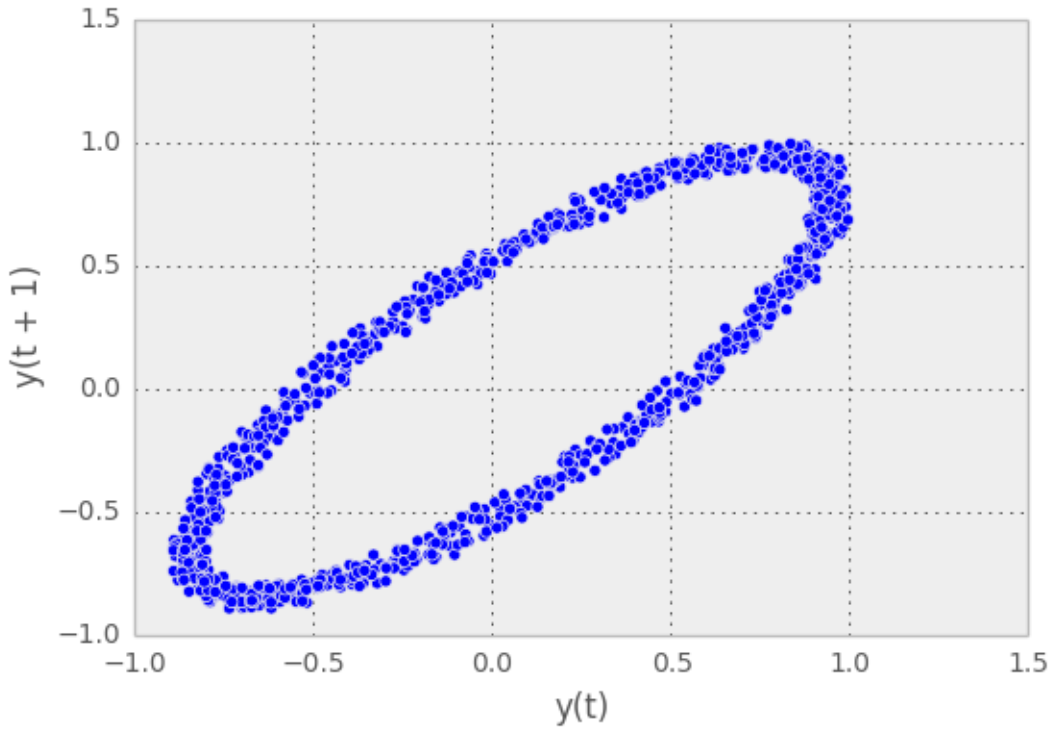
```
In [65]: from pandas import read_csv

In [66]: from pandas.tools.plotting import parallel_coordinates

In [67]: data = read_csv('data/iris.data')

In [68]: plt.figure()
<matplotlib.figure.Figure at 0x11b2e290>

In [69]: parallel_coordinates(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x11b2ecd0>
```



16.2.7 Lag Plot

Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random.

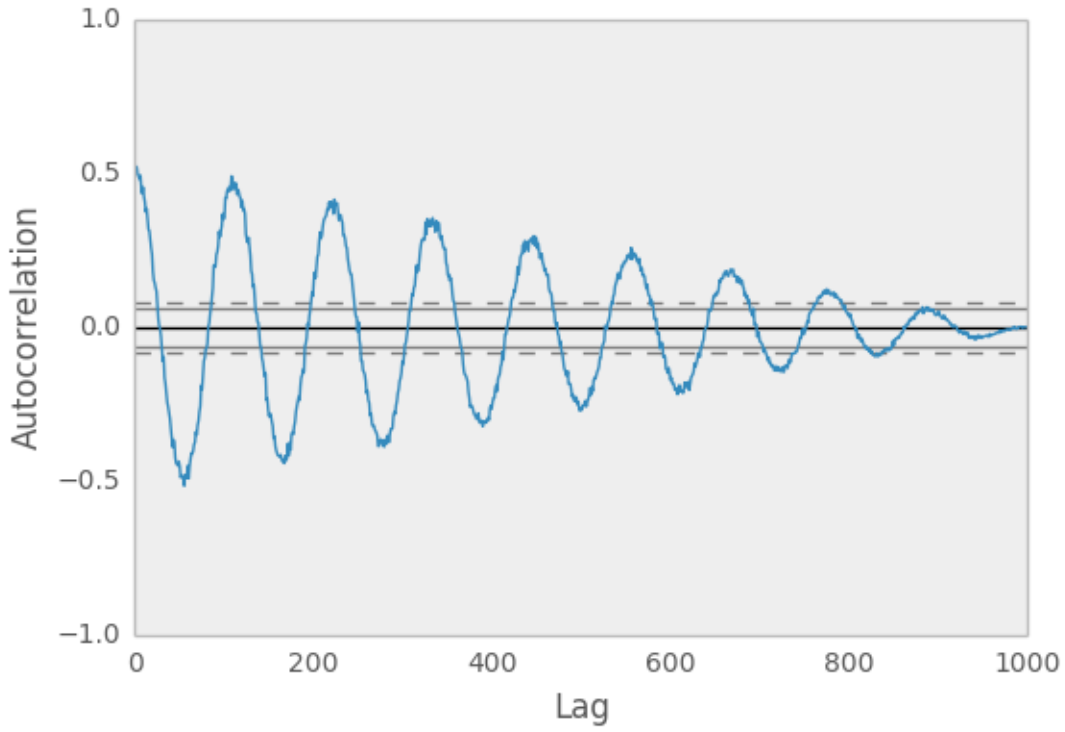
```
In [70]: from pandas.tools.plotting import lag_plot

In [71]: plt.figure()
<matplotlib.figure.Figure at 0x9f6ae10>

In [72]: data = Series(0.1 * rand(1000) +
   ....:     0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
   ....:

In [73]: lag_plot(data)
<matplotlib.axes.AxesSubplot at 0x123378d0>
```



16.2.8 Autocorrelation Plot

Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band.

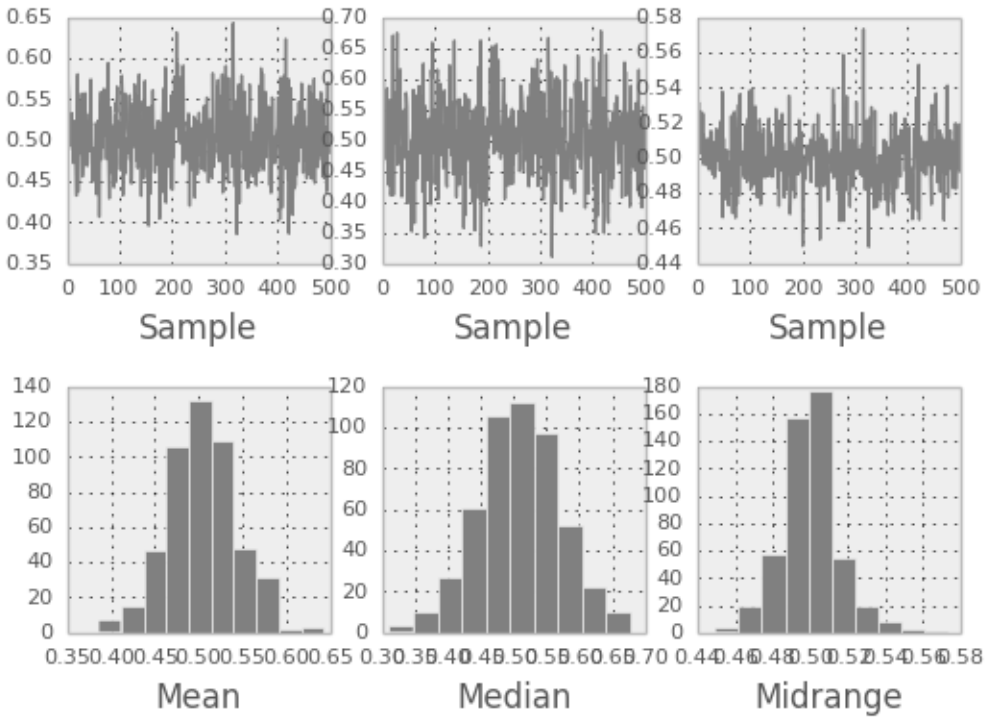
```
In [74]: from pandas.tools.plotting import autocorrelation_plot

In [75]: plt.figure()
<matplotlib.figure.Figure at 0x1235e8d0>

In [76]: data = Series(0.7 * rand(1000) +
   ....:     0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
   ....:

In [77]: autocorrelation_plot(data)
<matplotlib.axes.AxesSubplot at 0x1235e6d0>
```



16.2.9 Bootstrap Plot

Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc. A random subset of a specified size is selected from a data set, the statistic in question is computed for this subset and the process is repeated a specified number of times. Resulting plots and histograms are what constitutes the bootstrap plot.

```
In [78]: from pandas.tools.plotting import bootstrap_plot

In [79]: data = Series(rand(1000))

In [80]: bootstrap_plot(data, size=50, samples=500, color='grey')
<matplotlib.figure.Figure at 0x11a5bf90>
```

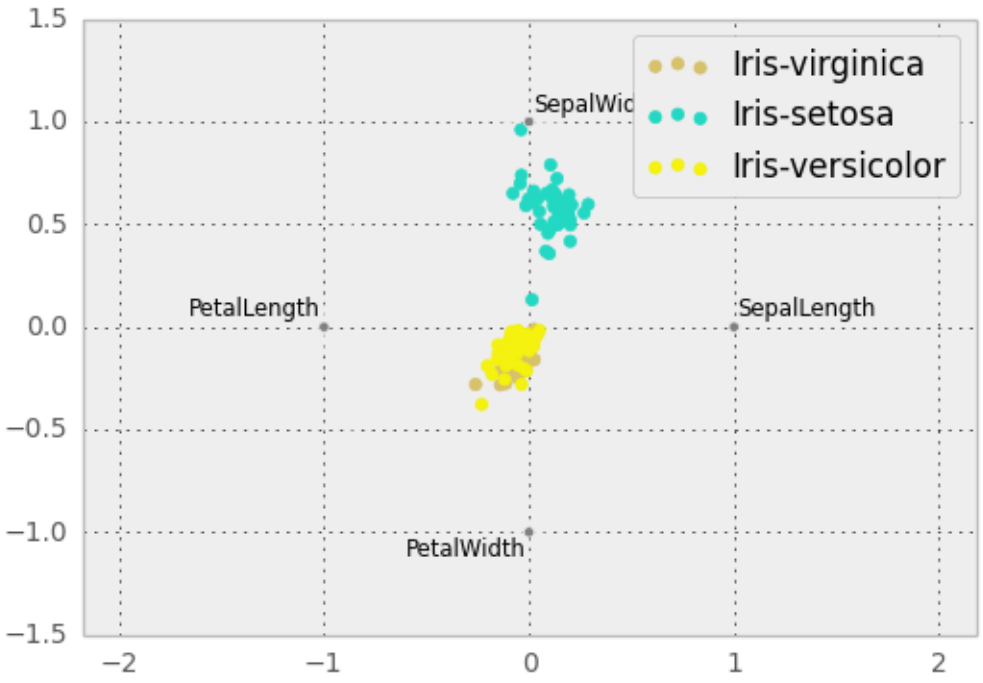


16.2.10 RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs it will be colored differently.

Note: The "Iris" dataset is available here.

```
In [81]: from pandas import read_csv

In [82]: from pandas.tools.plotting import radviz

In [83]: data = read_csv('data/iris.data')

In [84]: plt.figure()
<matplotlib.figure.Figure at 0x12bf3cd0>

In [85]: radviz(data, 'Name')
<matplotlib.axes.AxesSubplot at 0x1233d350>
```



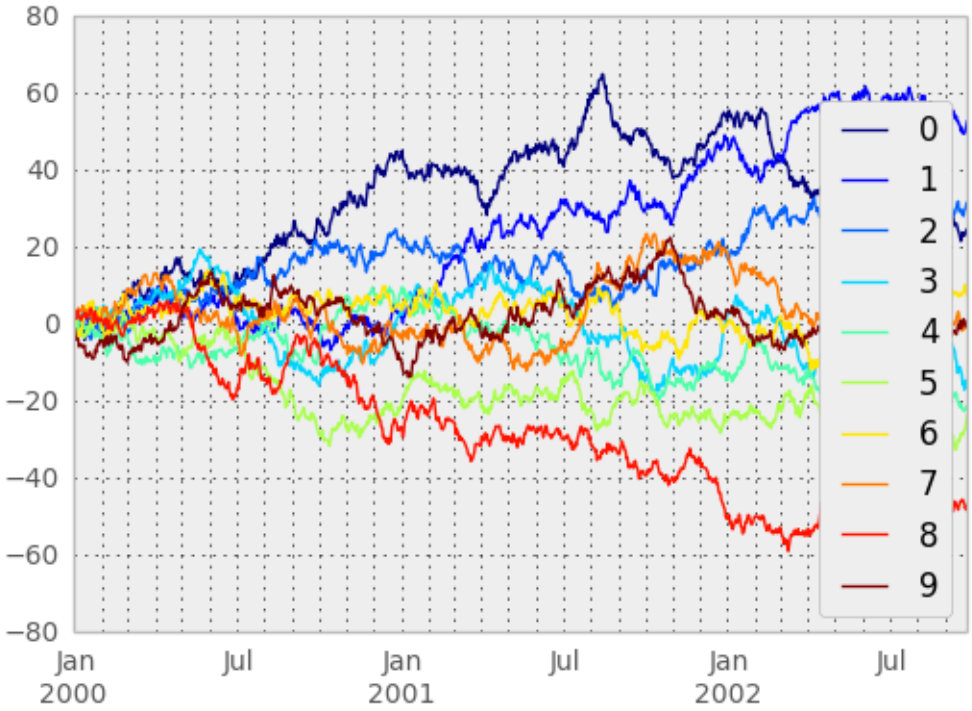
16.2.11 Colormaps

A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, DataFrame plotting supports the use of the `colormap=` argument, which accepts either a Matplotlib colormap or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available here.

As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the DataFrame. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

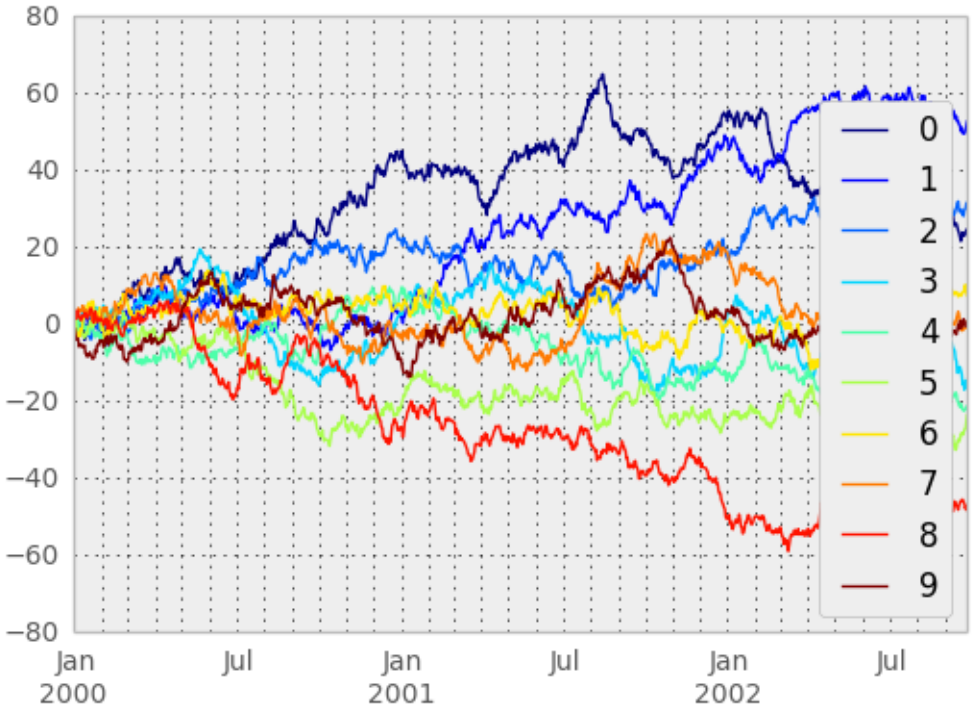
To use the jet colormap, we can simply pass `'jet'` to `colormap=`

```
In [86]: df = DataFrame(randn(1000, 10), index=ts.index)

In [87]: df = df.cumsum()

In [88]: plt.figure()
<matplotlib.figure.Figure at 0x130f32d0>

In [89]: df.plot(colormap='jet')
<matplotlib.axes.AxesSubplot at 0x13107950>
```



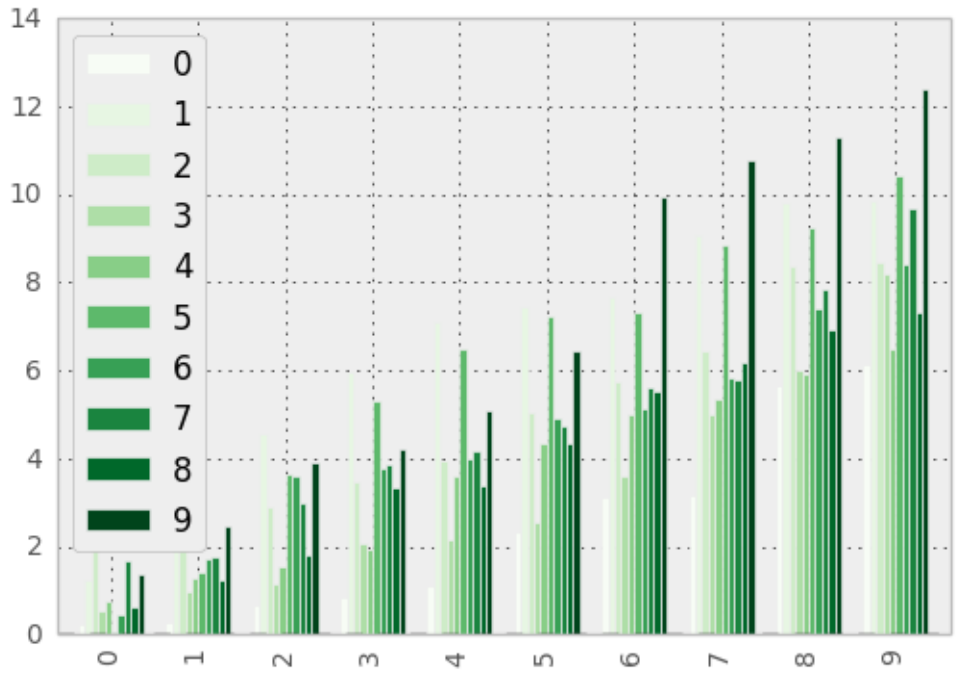
or we can pass the colormap itself

```
In [90]: from matplotlib import cm

In [91]: plt.figure()
<matplotlib.figure.Figure at 0x130ff910>

In [92]: df.plot(colormap=cm.jet)
<matplotlib.axes.AxesSubplot at 0x13ae9950>
```



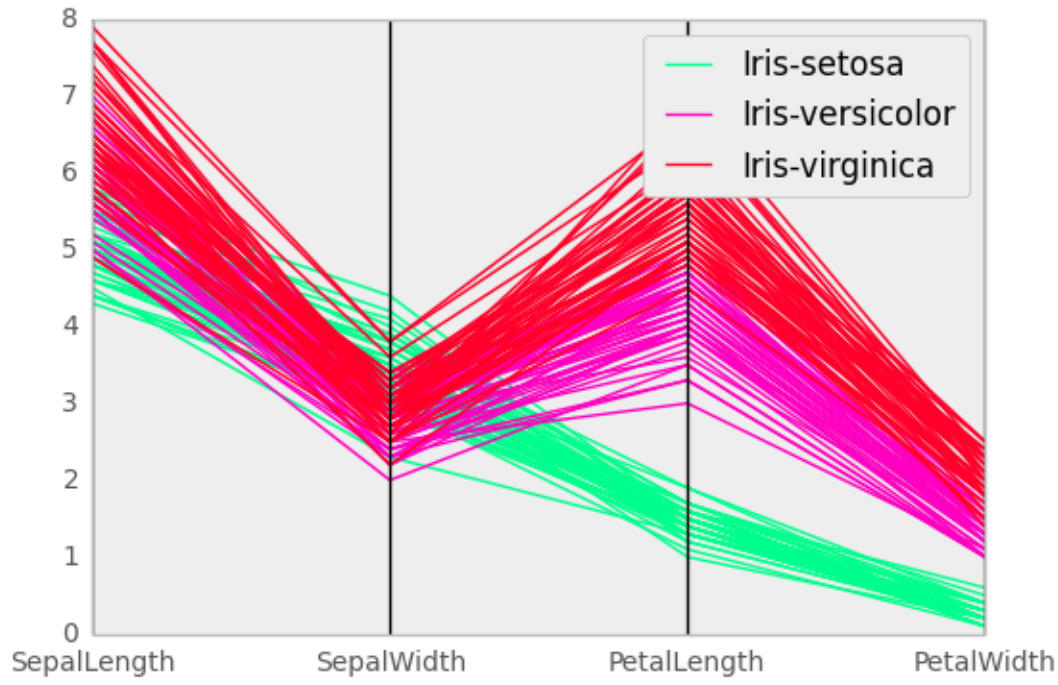
Colormaps can also be used other plot types, like bar charts:

```
In [93]: dd = DataFrame(randn(10, 10)).applymap(abs)

In [94]: dd = dd.cumsum()

In [95]: plt.figure()
<matplotlib.figure.Figure at 0x130f3510>

In [96]: dd.plot(kind='bar', colormap='Greens')
<matplotlib.axes.AxesSubplot at 0x14009fd0>
```



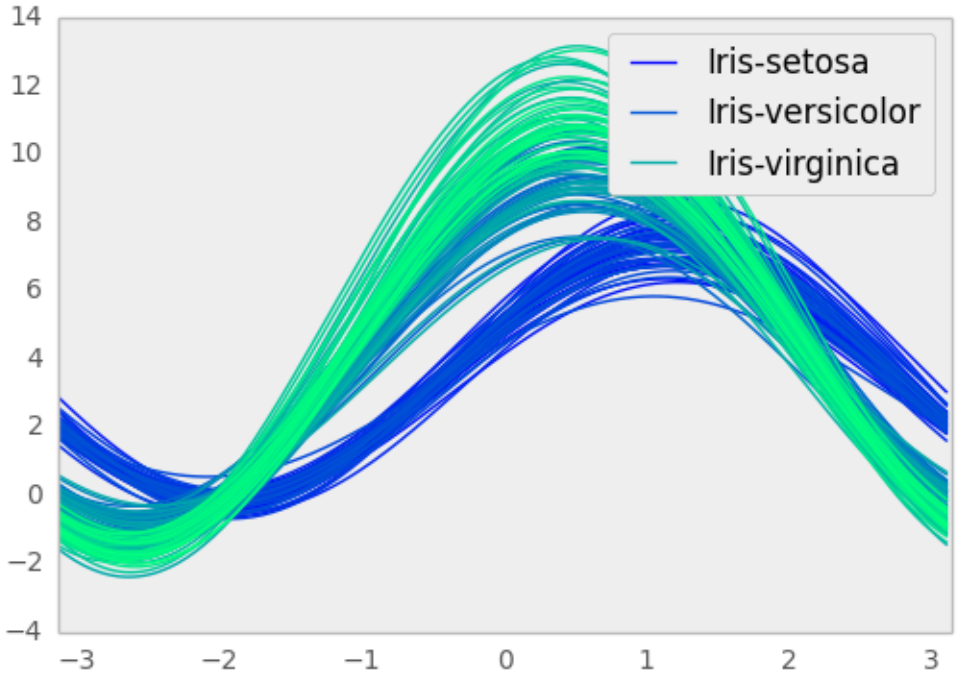
Parallel coordinates charts:

```
In [97]: plt.figure()
<matplotlib.figure.Figure at 0x1400f910>

In [98]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
<matplotlib.axes.AxesSubplot at 0x13e0b690>
```



Andrews curves charts:

```
In [99]: plt.figure()
<matplotlib.figure.Figure at 0x14aa5610>

In [100]: andrews_curves(data, 'Name', colormap='winter')
<matplotlib.axes.AxesSubplot at 0x136825d0>
```



CHAPTER

SEVENTEEN

TRELLIS PLOTTING INTERFACE

Note: The tips data set can be downloaded here. Once you download it execute

```
from pandas import read_csv
tips_data = read_csv('tips.csv')
```

from the directory where you downloaded the file.

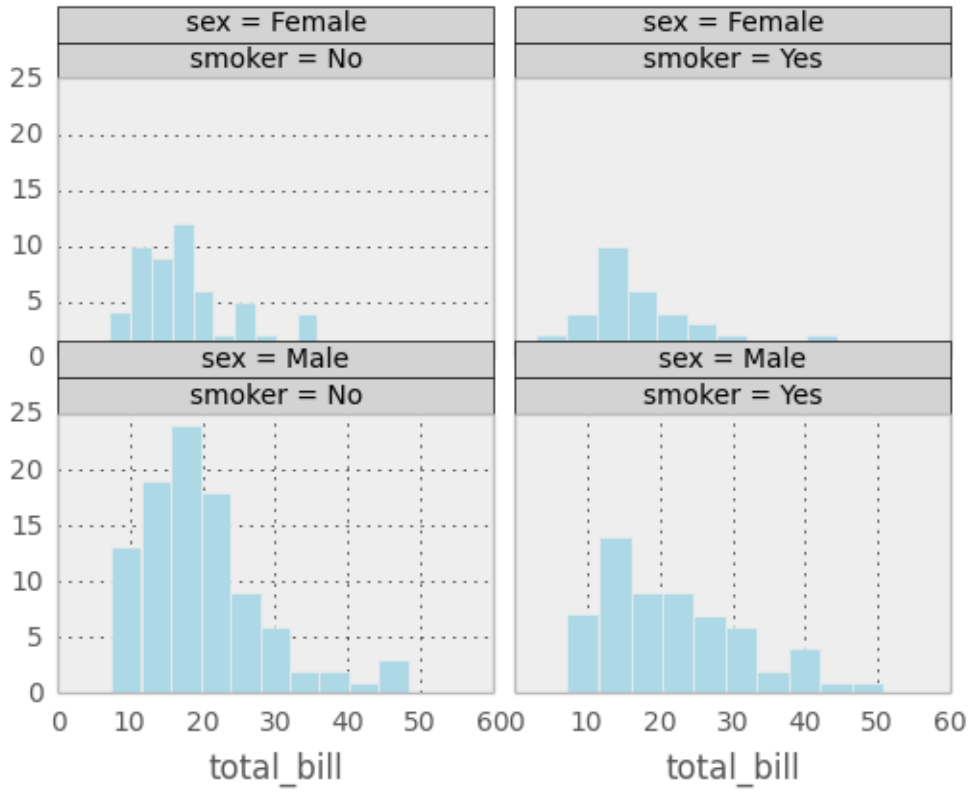
We import the rplot API:

```
In [1]: import pandas.tools.rplot as rplot
```

17.1 Examples

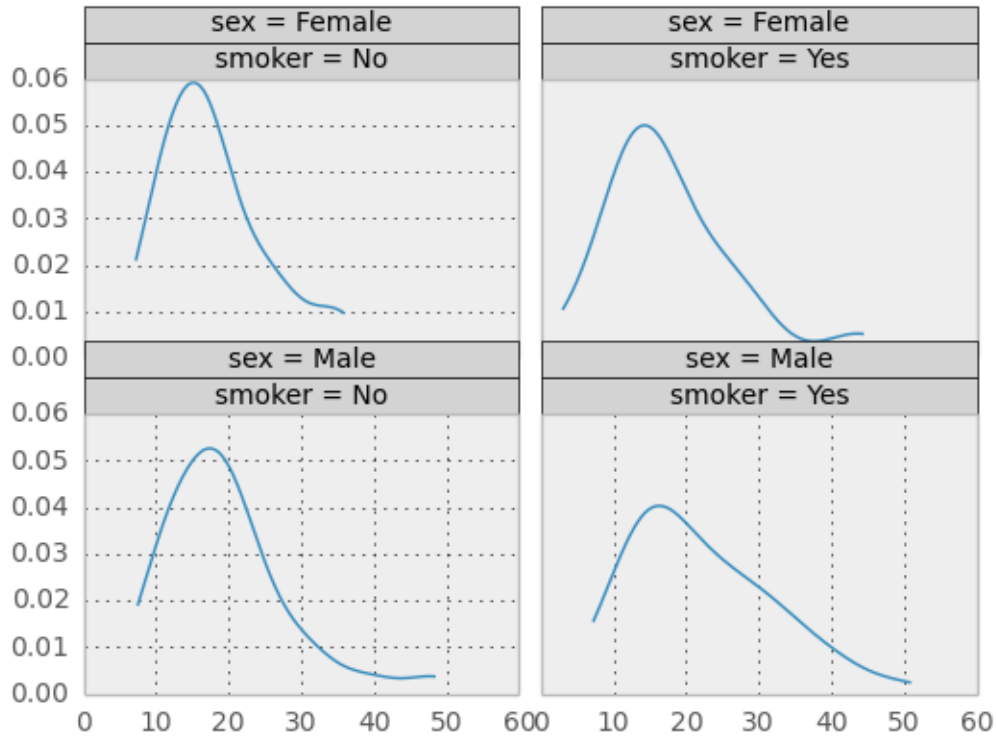
RPlot is a flexible API for producing Trellis plots. These plots allow you to arrange data in a rectangular grid by values of certain attributes.

```
In [2]: plt.figure()
<matplotlib.figure.Figure at 0x6271a90>

In [3]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [4]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [5]: plot.add(rplot.GeomHistogram())

In [6]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x6271a90>
```



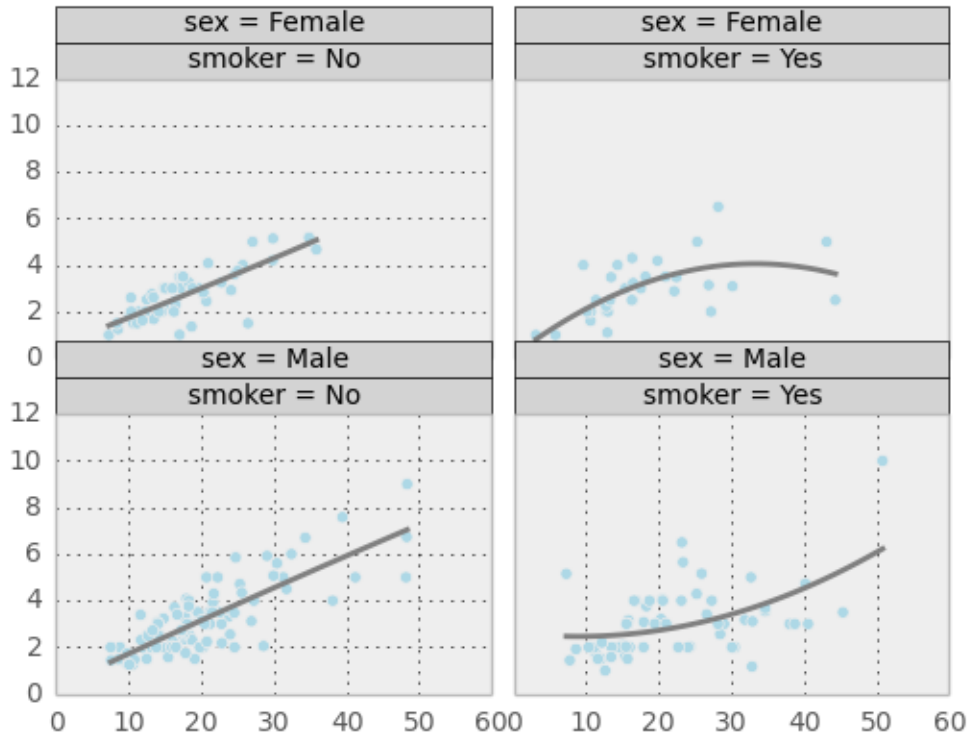
In the example above, data from the tips data set is arranged by the attributes 'sex' and 'smoker'. Since both of those attributes can take on one of two values, the resulting grid has two columns and two rows. A histogram is displayed for each cell of the grid.

```
In [7]: plt.figure()
<matplotlib.figure.Figure at 0x7bc5f10>

In [8]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [9]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [10]: plot.add(rplot.GeomDensity())

In [11]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x7bc5f10>
```



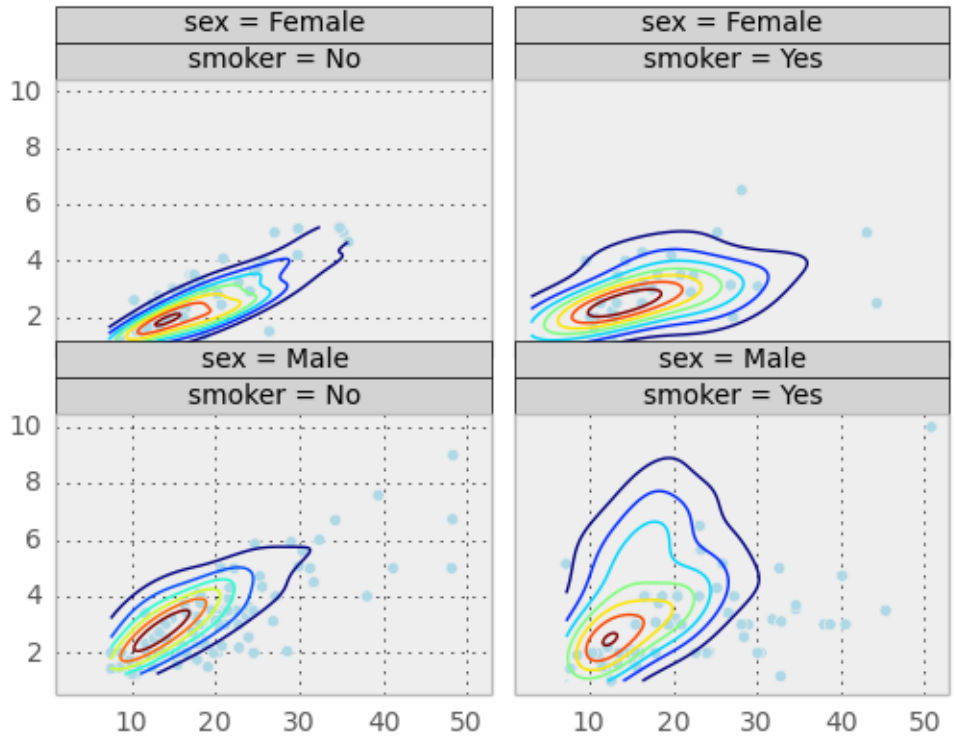
Example above is the same as previous except the plot is set to kernel density estimation. This shows how easy it is to have different plots for the same Trellis structure.

```
In [12]: plt.figure()
<matplotlib.figure.Figure at 0x7bd2890>

In [13]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [14]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [15]: plot.add(rplot.GeomScatter())

In [16]: plot.add(rplot.GeomPolyFit(degree=2))

In [17]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x7bd2890>
```



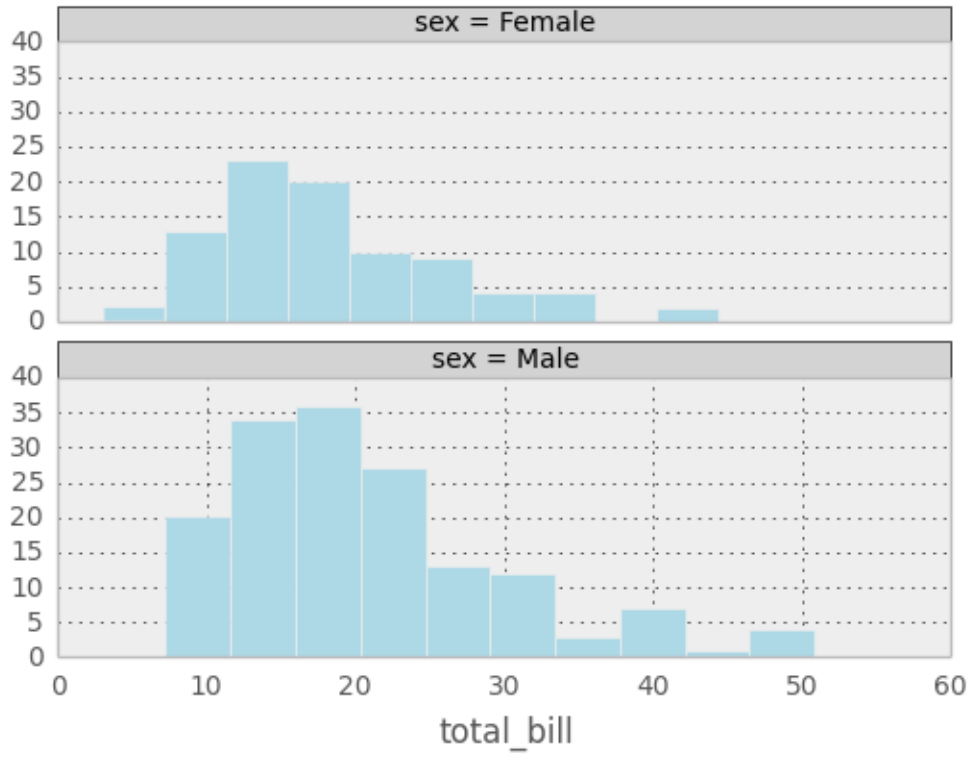
The plot above shows that it is possible to have two or more plots for the same data displayed on the same Trellis grid cell.

```
In [18]: plt.figure()
<matplotlib.figure.Figure at 0x7bc5c90>

In [19]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [20]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [21]: plot.add(rplot.GeomScatter())

In [22]: plot.add(rplot.GeomDensity2D())

In [23]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x7bc5c90>
```



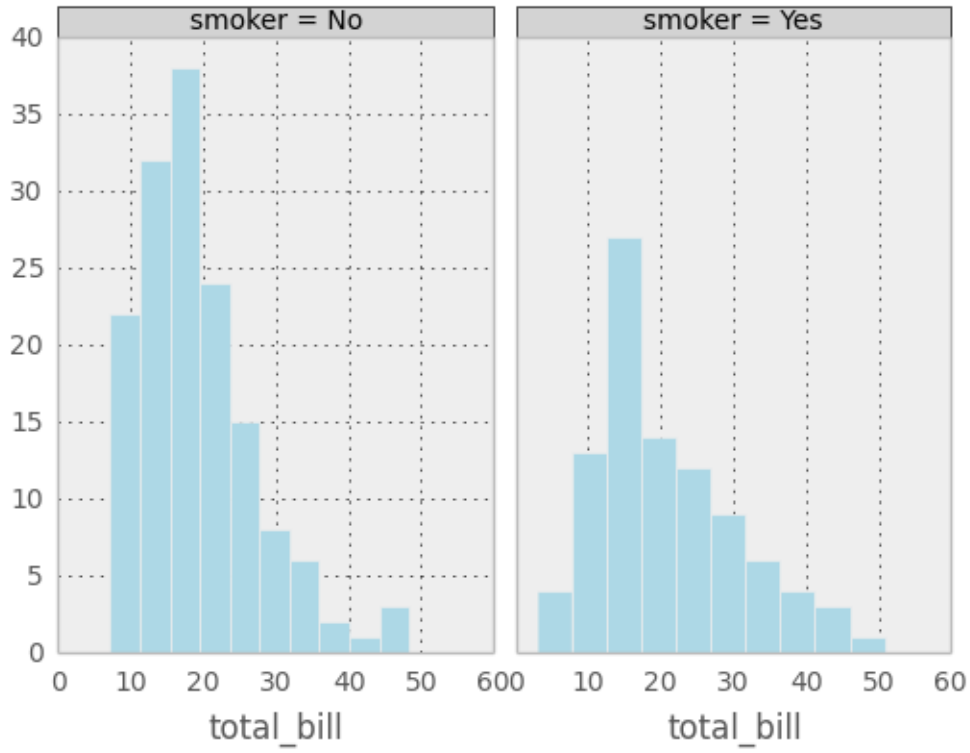
Above is a similar plot but with 2D kernel desnity estimation plot superimposed.

```
In [24]: plt.figure()
<matplotlib.figure.Figure at 0x8171e50>

In [25]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [26]: plot.add(rplot.TrellisGrid(['sex', '.']))

In [27]: plot.add(rplot.GeomHistogram())

In [28]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x8171e50>
```



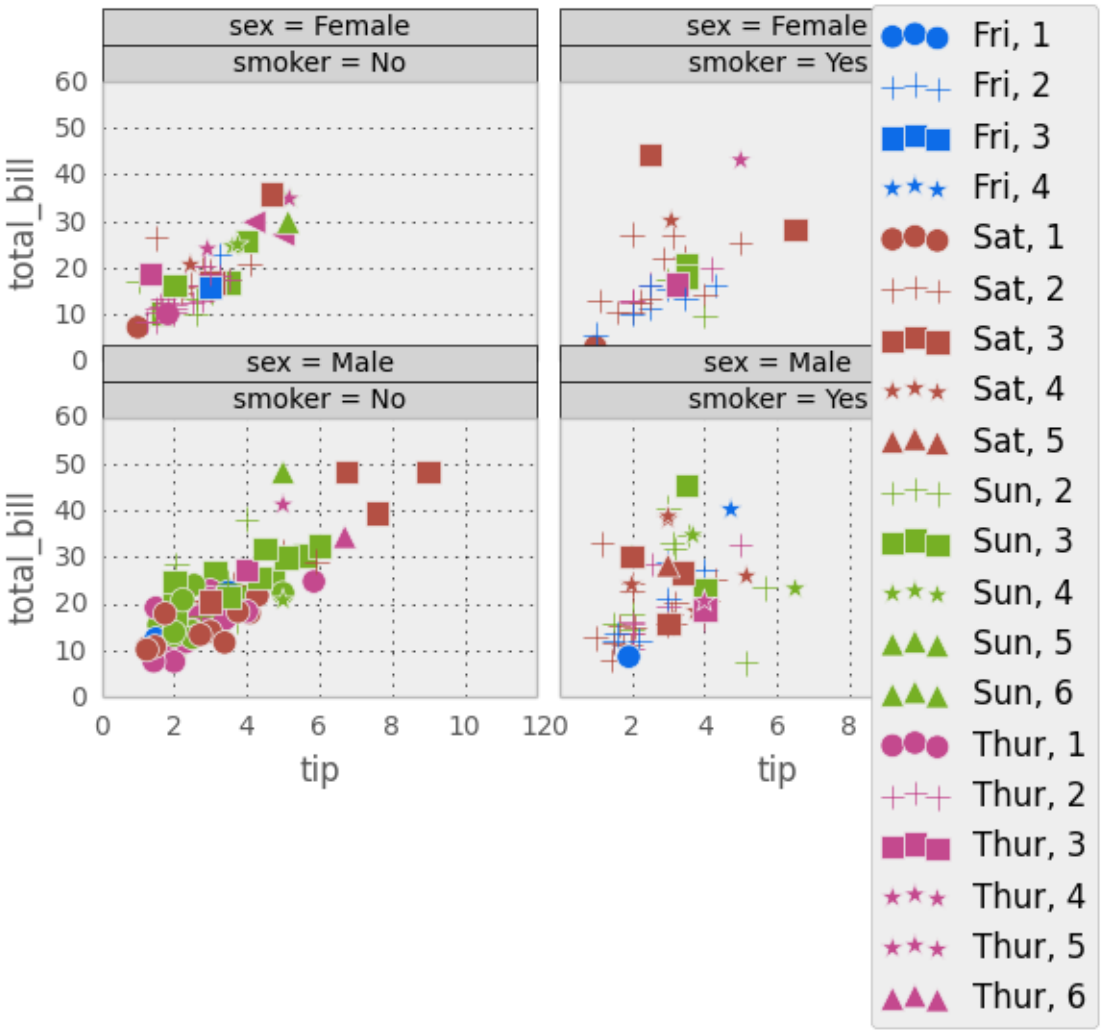
It is possible to only use one attribute for grouping data. The example above only uses 'sex' attribute. If the second grouping attribute is not specified, the plots will be arranged in a column.

```
In [29]: plt.figure()
<matplotlib.figure.Figure at 0x878e750>

In [30]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [31]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [32]: plot.add(rplot.GeomHistogram())

In [33]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x878e750>
```



If the first grouping attribute is not specified the plots will be arranged in a row.

```
In [34]: plt.figure()
<matplotlib.figure.Figure at 0x7bb9c50>

In [35]: plot = rplot.RPlot(tips_data, x='total_bill', y='tip')

In [36]: plot.add(rplot.TrellisGrid(['.', 'smoker']))

In [37]: plot.add(rplot.GeomHistogram())

In [38]: plot = rplot.RPlot(tips_data, x='tip', y='total_bill')

In [39]: plot.add(rplot.TrellisGrid(['sex', 'smoker']))

In [40]: plot.add(rplot.GeomPoint(size=80.0, colour=rplot.ScaleRandomColour('day'), shape=rplot.Scale

In [41]: plot.render(plt.gcf())
<matplotlib.figure.Figure at 0x7bb9c50>
```



As shown above, scatter plots are also possible. Scatter plots allow you to map various data attributes to graphical properties of the plot. In the example above the colour and shape of the scatter plot graphical objects is mapped to 'day' and 'size' attributes respectively. You use scale objects to specify these mappings. The list of scale classes is given below with initialization arguments for quick reference.

17.2 Scales

```
ScaleGradient(column, colour1, colour2)
```

This one allows you to map an attribute (specified by parameter column) value to the colour of a graphical object. The larger the value of the attribute the closer the colour will be to colour2, the smaller the value, the closer it will be to colour1.

```
ScaleGradient2(column, colour1, colour2, colour3)
```

The same as ScaleGradient but interpolates linearly between three colours instead of two.

```
ScaleSize(column, min_size, max_size, transform)
```

Map attribute value to size of the graphical object. Parameter min_size (default 5.0) is the minimum size of the graphical object, max_size (default 100.0) is the maximum size and transform is a one argument function that will be used to transform the attribute value (defaults to lambda x: x).

```
ScaleShape(column)
```

Map the shape of the object to attribute value. The attribute has to be categorical.

```
ScaleRandomColour(column)
```

Assign a random colour to a value of categorical attribute specified by column.

CHAPTER

EIGHTEEN

IO TOOLS (TEXT, CSV, HDF5, ...)

The Pandas I/O api is a set of top level `reader` functions accessed like `pd.read_csv()` that generally return a `pandas` object.

- `read_csv`
- `read_excel`
- `read_hdf`
- `read_sql`
- `read_json`
- `read_html`
- `read_stata`
- `read_clipboard`
- `read_pickle`

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

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

18.1 CSV & Text files

The two workhorse functions for reading text files (a.k.a. flat files) are `read_csv()` and `read_table()`. They both use the same parsing code to intelligently convert tabular data into a DataFrame object. See the *cookbook* for some advanced strategies

They can take a number of arguments:

- `filepath_or_buffer`: Either a string path to a file, url (including http, ftp, and s3 locations), or any object with a `read` method (such as an open file or `StringIO`).
- `sep` or `delimiter`: A delimiter / separator to split fields on. *read_csv* is capable of inferring the delimiter automatically in some cases by "sniffing." The separator may be specified as a regular expression; for instance you may use '|\s*' to indicate a pipe plus arbitrary whitespace.
- `delim_whitespace`: Parse whitespace-delimited (spaces or tabs) file (much faster than using a regular expression)
- `compression`: decompress `'gzip'` and `'bz2'` formats on the fly.
- `dialect`: string or `csv.Dialect` instance to expose more ways to specify the file format
- `dtype`: A data type name or a dict of column name to data type. If not specified, data types will be inferred.
- `header`: row number to use as the column names, and the start of the data. Defaults to 0 if no `names` passed, otherwise `None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns E.g. [0,1,3]. Interveaning rows that are not specified will be skipped. (E.g. 2 in this example are skipped)
- `skiprows`: A collection of numbers for rows in the file to skip. Can also be an integer to skip the first `n` rows
- `index_col`: column number, column name, or list of column numbers/names, to use as the `index` (row labels) of the resulting DataFrame. By default, it will number the rows without using any column, unless there is one more data column than there are headers, in which case the first column is taken as the index.
- `names`: List of column names to use as column names. To replace header existing in file, explicitly pass `header=0`.
- `na_values`: optional list of strings to recognize as NaN (missing values), either in addition to or in lieu of the default set.
- `true_values`: list of strings to recognize as `True`
- `false_values`: list of strings to recognize as `False`
- `keep_default_na`: whether to include the default set of missing values in addition to the ones specified in `na_values`
- `parse_dates`: if True then index will be parsed as dates (False by default). You can specify more complicated options to parse a subset of columns or a combination of columns into a single date column (list of ints or names, list of lists, or dict) [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column [[1, 3]] -> combine columns 1 and 3 and parse as a single date column {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'
- `keep_date_col`: if True, then date component columns passed into `parse_dates` will be retained in the output (False by default).
- `date_parser`: function to use to parse strings into datetime objects. If `parse_dates` is True, it defaults to the very robust `dateutil.parser`. Specifying this implicitly sets `parse_dates` as True. You can also use functions from community supported date converters from date_converters.py
- `dayfirst`: if True then uses the DD/MM international/European date format (This is False by default)
- `thousands`: sepcifies the thousands separator. If not None, then parser will try to look for it in the output and parse relevant data to integers. Because it has to essentially scan through the data again, this causes a significant performance hit so only use if necessary.
- `lineterminator` : string (length 1), default `None`, Character to break file into lines. Only valid with C parser
- `quotechar` : string, The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

- `quoting` : int, Controls whether quotes should be recognized. Values are taken from *csv.QUOTE_** values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.
- `skipinitialspace` : boolean, default `False`, Skip spaces after delimiter
- `escapechar` : string, to specify how to escape quoted data
- `comment`: denotes the start of a comment and ignores the rest of the line. Currently line commenting is not supported.
- `nrows`: Number of rows to read out of the file. Useful to only read a small portion of a large file
- `iterator`: If True, return a `TextFileReader` to enable reading a file into memory piece by piece
- `chunksize`: An number of rows to be used to "chunk" a file into pieces. Will cause an `TextFileReader` object to be returned. More on this below in the section on *iterating and chunking*
- `skip_footer`: number of lines to skip at bottom of file (default 0)
- `converters`: a dictionary of functions for converting values in certain columns, where keys are either integers or column labels
- `encoding`: a string representing the encoding to use for decoding unicode data, e.g. `'utf-8`` or `'latin-1'`.
- `verbose`: show number of NA values inserted in non-numeric columns
- `squeeze`: if True then output with only one column is turned into Series
- `error_bad_lines`: if False then any lines causing an error will be skipped *bad lines*
- `usecols`: a subset of columns to return, results in much faster parsing time and lower memory usage.
- `mangle_dupe_cols`: boolean, default True, then duplicate columns will be specified as 'X.0'...'X.N', rather than 'X'...'X'
- `tupleize_cols`: boolean, default True, if False, convert a list of tuples to a multi-index of columns, otherwise, leave the column index as a list of tuples

Consider a typical CSV file containing, in this case, some time series data:

```
In [1]: print open('foo.csv').read()
date,A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

The default for *read_csv* is to create a DataFrame with simple numbered rows:

```
In [2]: pd.read_csv('foo.csv')

       date  A  B  C
0  20090101  a  1  2
1  20090102  b  3  4
2  20090103  c  4  5
```

In the case of indexed data, you can pass the column number or column name you wish to use as the index:

```
In [3]: pd.read_csv('foo.csv', index_col=0)

          A  B  C
date
20090101  a  1  2
```

```
20090102  b  3  4
20090103  c  4  5
```

```
In [4]: pd.read_csv('foo.csv', index_col='date')
```

```
          A  B  C
date
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

You can also use a list of columns to create a hierarchical index:

```
In [5]: pd.read_csv('foo.csv', index_col=[0, 'A'])
```

```
            B  C
date     A
20090101 a  1  2
20090102 b  3  4
20090103 c  4  5
```

The `dialect` keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a `csv.Dialect` instance.

Suppose you had data with unenclosed quotes:

```
In [6]: print data
label1,label2,label3
index1,"a,c,e
index2,b,d,f
```

By default, `read_csv` uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using `dialect`

```
In [7]: dia = csv.excel()

In [8]: dia.quoting = csv.QUOTE_NONE

In [9]: pd.read_csv(StringIO(data), dialect=dia)
```

```
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [10]: data = 'a,b,c~1,2,3~4,5,6'

In [11]: pd.read_csv(StringIO(data), lineterminator='~')
```

```
   a  b  c
0  1  2  3
1  4  5  6
```

Another common dialect option is `skipinitialspace`, to skip any whitespace after a delimiter:

```
In [12]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [13]: print data
```

```
a, b, c
1, 2, 3
4, 5, 6

In [14]: pd.read_csv(StringIO(data), skipinitialspace=True)

   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to "do the right thing" and not be very fragile. Type inference is a pretty big deal. So if a column can be coerced to integer dtype without altering the contents, it will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

18.1.1 Specifying column data types

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

```
In [15]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'

In [16]: print data
a,b,c
1,2,3
4,5,6
7,8,9

In [17]: df = pd.read_csv(StringIO(data), dtype=object)

In [18]: df

   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

In [19]: df['a'][0]
'1'

In [20]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})

In [21]: df.dtypes

a      int64
b     object
c    float64
dtype: object
```

18.1.2 Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

```
In [22]: from StringIO import StringIO

In [23]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'

In [24]: print data
```

```
a,b,c
1,2,3
4,5,6
7,8,9

In [25]: pd.read_csv(StringIO(data))

   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

By specifying the `names` argument in conjunction with `header` you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [26]: print data
a,b,c
1,2,3
4,5,6
7,8,9

In [27]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)

   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [28]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)

  foo bar baz
0   a   b   c
1   1   2   3
2   4   5   6
3   7   8   9
```

If the header is in a row other than the first, pass the row number to `header`. This will skip the preceding rows:

```
In [29]: data = 'skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9'

In [30]: pd.read_csv(StringIO(data), header=1)

   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

18.1.3 Filtering columns (`usecols`)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names or position numbers:

```
In [31]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [32]: pd.read_csv(StringIO(data))

   a  b  c    d
0  1  2  3  foo
```

```
1  4  5  6  bar
2  7  8  9  baz

In [33]: pd.read_csv(StringIO(data), usecols=['b', 'd'])

   b    d
0  2  foo
1  5  bar
2  8  baz

In [34]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])

   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz
```

18.1.4 Dealing with Unicode Data

The `encoding` argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

```
In [35]: data = 'word,length\nTr\xe4umen,7\nGr\xfc\xdfe,5'

In [36]: df = pd.read_csv(StringIO(data), encoding='latin-1')

In [37]: df

      word  length
0  Träumen       7
1    Grüße       5

In [38]: df['word'][1]
u'Gr\xfc\xdfe'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won't parse correctly at all without specifying the encoding.

18.1.5 Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the DataFrame's row names:

```
In [39]: data = 'a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [40]: pd.read_csv(StringIO(data))

        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0

In [41]: data = 'index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [42]: pd.read_csv(StringIO(data), index_col=0)

            a    b     c
```

```
index
4       apple  bat    5.7
8      orange  cow   10.0
```

Ordinarily, you can achieve this behavior using the `index_col` option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass `index_col=False`:

```
In [43]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [44]: print data
a,b,c
4,apple,bat,
8,orange,cow,

In [45]: pd.read_csv(StringIO(data))

        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [46]: pd.read_csv(StringIO(data), index_col=False)

   a       b    c
0  4   apple  bat
1  8  orange  cow
```

18.1.6 Specifying Date Columns

To better facilitate working with datetime data, `read_csv()` and `read_table()` uses the keyword arguments `parse_dates` and `date_parser` to allow users to specify a variety of columns and date/time formats to turn the input text data into `datetime` objects.

The simplest case is to just pass in `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [47]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [48]: df

            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5

# These are python datetime objects
In [49]: df.index

<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

It is often the case that we may want to store date and time data separately, or store various date fields separately. the `parse_dates` keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to `parse_dates`, the resulting date columns will be prepended to the output

(so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

```
In [50]: print open('tmp.csv').read()
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

In [51]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [52]: df

                  1_2                 1_3     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [53]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   ....:                  keep_date_col=True)
   ....:

In [54]: df

                  1_2                 1_3     0         1          2  \
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00
           3     4
0   18:56:00  0.81
1   19:56:00  0.01
2   20:56:00 -0.59
3   21:18:00 -0.99
4   21:56:00 -0.59
5   22:56:00 -0.59
```

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, `parse_dates=[1, 2]` indicates that the second and third columns should each be parsed as separate date columns while `parse_dates=[[1, 2]]` means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [55]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [56]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [57]: df

              nominal              actual     0     4
```

```
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The *index_col* specification is based off of this new set of columns rather than the original data columns:

```
In [58]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [59]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   ....:                  index_col=0) #index is the nominal column
   ....:

In [60]: df

                                 actual     0     4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Note: When passing a dict as the *parse_dates* argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use *collections.OrderedDict* instead of a regular *dict* if this matters to you. Because of this, when using a dict for 'parse_dates' in conjunction with the *index_col* argument, it's best to specify *index_col* as a column label rather then as an index on the resulting frame.

18.1.7 Date Parsing Functions

Finally, the parser allows you can specify a custom `date_parser` function to take full advantage of the flexiblity of the date parsing API:

```
In [61]: import pandas.io.date_converters as conv

In [62]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   ....:                  date_parser=conv.parse_date_time)
   ....:

In [63]: df

              nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

You can explore the date parsing functionality in `date_converters.py` and add your own. We would love to turn this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contains functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second

columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single date rather than the entire array.

18.1.8 International Date Formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a `dayfirst` keyword is provided:

```
In [64]: print open('tmp.csv').read()
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [65]: pd.read_csv('tmp.csv', parse_dates=[0])

                  date  value cat
0 2000-01-06 00:00:00      5   a
1 2000-02-06 00:00:00     10   b
2 2000-03-06 00:00:00     15   c

In [66]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])

                  date  value cat
0 2000-06-01 00:00:00      5   a
1 2000-06-02 00:00:00     10   b
2 2000-06-03 00:00:00     15   c
```

18.1.9 Thousand Separators

For large integers that have been written with a thousands separator, you can set the `thousands` keyword to `True` so that integers will be parsed correctly:

By default, integers with a thousands separator will be parsed as strings

```
In [67]: print open('tmp.csv').read()
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [68]: df = pd.read_csv('tmp.csv', sep='|')

In [69]: df

         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [70]: df.level.dtype
dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly

```
In [71]: print open('tmp.csv').read()
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [72]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [73]: df

         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [74]: df.level.dtype
dtype('int64')
```

18.1.10 Comments

Sometimes comments or meta data may be included in a file:

```
In [75]: print open('tmp.csv').read()
ID,level,category
Patient1,123000,x # really unpleasant
Patient2,23000,y # wouldn't take his medicine
Patient3,1234018,z # awesome
```

By default, the parse includes the comments in the output:

```
In [76]: df = pd.read_csv('tmp.csv')

In [77]: df

         ID    level                          category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

We can suppress the comments using the `comment` keyword:

```
In [78]: df = pd.read_csv('tmp.csv', comment='#')

In [79]: df

         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z
```

18.1.11 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a `Series`:

```
In [80]: print open('tmp.csv').read()
level
```

```
Patient1,123000
Patient2,23000
Patient3,1234018

In [81]: output =  pd.read_csv('tmp.csv', squeeze=True)

In [82]: output

Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [83]: type(output)
pandas.core.series.Series
```

18.1.12 Boolean values

The common values `True`, `False`, `TRUE`, and `FALSE` are all recognized as boolean. Sometime you would want to recognize some other values as being boolean. To do this use the `true_values` and `false_values` options:

```
In [84]: data= 'a,b,c\n1,Yes,2\n3,No,4'

In [85]: print data
a,b,c
1,Yes,2
3,No,4

In [86]: pd.read_csv(StringIO(data))

   a    b  c
0  1  Yes  2
1  3   No  4

In [87]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])

   a      b  c
0  1   True  2
1  3  False  4
```

18.1.13 Handling "bad" lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many will cause an error by default:

```
In [27]: data = 'a,b,c\n1,2,3\n4,5,6,7\n8,9,10'

In [28]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
CParserError                              Traceback (most recent call last)
CParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4
```

You can elect to skip bad lines:

```
In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
Skipping line 3: expected 3 fields, saw 4

Out[29]:
   a  b   c
0  1  2   3
1  8  9  10
```

18.1.14 Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the `escapechar` option:

```
In [88]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [89]: print data
a,b
"hello, \"Bob\", nice to see you",5

In [90]: pd.read_csv(StringIO(data), escapechar='\\')

                               a  b
0  hello, "Bob", nice to see you  5
```

18.1.15 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 [91]: print open('bar.csv').read()
id8141    360.242940   149.910199   11950.7
id1594    444.953632   166.985655   11788.4
id1849    364.136849   183.628767   11806.2
id1230    413.836124   184.375703   11916.8
id1948    502.953953   173.237159   12468.3
```

In order to parse this file into a DataFrame, we simply need to supply the column specifications to the *read_fwf* function along with the file name:

```
#Column specifications are a list of half-intervals
In [92]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [93]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [94]: df

                 1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
```

```
id1849   364.136849   183.628767   11806.2
id1230   413.836124   184.375703   11916.8
id1948   502.953953   173.237159   12468.3
```

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
#Widths are a list of integers
In [95]: widths = [6, 14, 13, 10]

In [96]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [97]: df
```

```
        0           1           2        3
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

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.

18.1.16 Files with an "implicit" index column

Consider a file with one less entry in the header than the number of data column:

```
In [98]: 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 [99]: pd.read_csv('foo.csv')
```

```
          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 [100]: df = pd.read_csv('foo.csv', parse_dates=True)

In [101]: df.index

<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

18.1.17 Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [102]: print open('data/mindex_ex.csv').read()
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

The `index_col` argument to `read_csv` and `read_table` can take a list of column numbers to turn multiple columns into a `MultiIndex` for the index of the returned object:

```
In [103]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])

In [104]: df
```

```
             zit   xit
year indiv
1977 A      1.20  0.60
     B      1.50  0.50
     C      1.70  0.80
1978 A      0.20  0.06
     B      0.70  0.20
     C      0.80  0.30
     D      0.90  0.50
     E      1.40  0.90
1979 C      0.20  0.15
     D      0.14  0.05
     E      0.50  0.15
     F      1.20  0.50
     G      3.40  1.90
     H      5.40  2.70
     I      6.40  1.20
```

```
In [105]: df.ix[1978]
```

```
       zit   xit
indiv
A      0.2  0.06
B      0.7  0.20
C      0.8  0.30
D      0.9  0.50
E      1.4  0.90
```

18.1.18 Reading columns with a `MultiIndex`

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the interveaning rows.

```
In [106]: from pandas.util.testing import makeCustomDataframe as mkdf

In [107]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)

In [108]: df.to_csv('mi.csv',tupleize_cols=False)

In [109]: print open('mi.csv').read()
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2

In [110]: pd.read_csv('mi.csv',header=[0,1,2,3],index_col=[0,1],tupleize_cols=False)

C0              C_l0_g0 C_l0_g1 C_l0_g2
C1              C_l1_g0 C_l1_g1 C_l1_g2
C2              C_l2_g0 C_l2_g1 C_l2_g2
C3              C_l3_g0 C_l3_g1 C_l3_g2
R0      R1
R_l0_g0 R_l1_g0    R0C0    R0C1    R0C2
R_l0_g1 R_l1_g1    R1C0    R1C1    R1C2
R_l0_g2 R_l1_g2    R2C0    R2C1    R2C2
R_l0_g3 R_l1_g3    R3C0    R3C1    R3C2
R_l0_g4 R_l1_g4    R4C0    R4C1    R4C2
```

Note: The default behavior in 0.12 remains unchanged (`tupleize_cols=True`) from prior versions, but starting with 0.13, the default *to* write and read multi-index columns will be in the new format (`tupleize_cols=False`)

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

18.1.19 Automatically "sniffing" the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files. YMMV, as pandas uses the `csv.Sniffer` class of the csv module.

```
In [111]: print open('tmp2.sv').read()
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498

In [112]: pd.read_csv('tmp2.sv')
```

```
                                                 :0:1:2:3
0  0:0.4691122999071863:-0.282863344328663:-1.50...
1  1:1.2121120250208506:-0.17321464905330858:0.11...
2  2:-0.8618489633477999:-2.1045692188948086:-0.4...
3  3:0.7215551622443669:-0.7067711336300845:-1.03...
4  4:-0.42497232978883753:0.567020349793672:0.276...
5  5:-0.6736897080883706:0.1136484096888855:-1.47...
6  6:0.4047052186802365:0.5770459859204836:-1.715...
7  7:-0.3706468582364464:-1.1578922506419993:-1.3...
8  8:1.0757697837155533:-0.10904997528022223:1.64...
9  9:0.35702056413309086:-0.6746001037299882:-1.7...
```

18.1.20 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 [113]: print open('tmp.sv').read()
|0|1|2|3
0|0.4691122999071863|-0.282863344328663|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498

In [114]: table = pd.read_table('tmp.sv', sep='|')

In [115]: table

   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

By specifiying a `chunksize` to `read_csv` or `read_table`, the return value will be an iterable object of type `TextFileReader`:

```
In [116]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)

In [117]: reader
<pandas.io.parsers.TextFileReader at 0xbd77950>

In [118]: for chunk in reader:
   .....:     print chunk
   .....:
```

```
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
   Unnamed: 0         0         1         2         3
0           4 -0.424972  0.567020  0.276232 -1.087401
1           5 -0.673690  0.113648 -1.478427  0.524988
2           6  0.404705  0.577046 -1.715002 -1.039268
3           7 -0.370647 -1.157892 -1.344312  0.844885
   Unnamed: 0         0        1         2         3
0           8  1.075770 -0.10905  1.643563 -1.469388
1           9  0.357021 -0.67460 -1.776904 -0.968914
```

Specifying `iterator=True` will also return the `TextFileReader` object:

```
In [119]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)

In [120]: reader.get_chunk(5)
```

```
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
```

18.1.21 Writing to CSV format

The Series and DataFrame objects have an instance method `to_csv` which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- `path`: A string path to the file to write
- `na_rep`: A string representation of a missing value (default ‘’)
- `cols`: Columns to write (default None)
- `header`: Whether to write out the column names (default True)
- `index`: whether to write row (index) names (default True)
- `index_label`: Column label(s) for index column(s) if desired. If None (default), and *header* and *index* are True, then the index names are used. (A sequence should be given if the DataFrame uses MultiIndex).
- `mode` : Python write mode, default ‘w’
- `sep` : Field delimiter for the output file (default ”,”)
- `encoding`: a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3
- `tupleize_cols`: boolean, default True, if False, write as a list of tuples, otherwise write in an expanded line format suitable for `read_csv`

18.1.22 Writing a formatted string

The DataFrame object has an instance method `to_string` which allows control over the string representation of the object. All arguments are optional:

- `buf` default None, for example a StringIO object
- `columns` default None, which columns to write
- `col_space` default None, minimum width of each column.
- `na_rep` default `NaN`, representation of NA value
- `formatters` default None, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- `float_format` default None, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the DataFrame.
- `sparsify` default True, set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.
- `index_names` default True, will print the names of the indices
- `index` default True, will print the index (ie, row labels)
- `header` default True, will print the column labels
- `justify` default `left`, will print column headers left- or right-justified

The Series object also has a `to_string` method, but with only the `buf`, `na_rep`, `float_format` arguments. There is also a `length` argument which, if set to `True`, will additionally output the length of the Series.

18.2 JSON

Read and write `JSON` format files.

18.2.1 Writing JSON

A `Series` or `DataFrame` can be converted to a valid JSON string. Use `to_json` with optional parameters:

- `path_or_buf` : the pathname or buffer to write the output This can be `None` in which case a JSON string is returned
- `orient` :

  Series :

  - default is `index`
  - allowed values are {`split`, `records`, `index`}

  DataFrame

  - default is `columns`
  - allowed values are {`split`, `records`, `index`, `columns`, `values`}

  The format of the JSON string

`split`	dict like {index -> [index], columns -> [columns], data -> [values]}
`records`	list like [{column -> value}, ... , {column -> value}]
`index`	dict like {index -> {column -> value}}
`columns`	dict like {column -> {index -> value}}
`values`	just the values array

- `date_format` : type of date conversion (epoch = epoch milliseconds, iso = ISO8601), default is epoch

- `double_precision` : The number of decimal places to use when encoding floating point values, default 10.
- `force_ascii` : force encoded string to be ASCII, default True.

Note NaN's and None will be converted to null and datetime objects will be converted based on the date_format parameter

```
In [121]: dfj = DataFrame(randn(5, 2), columns=list('AB'))

In [122]: json = dfj.to_json()

In [123]: json
'{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.0061535699,"4":0.8957173022},"B":
```

Writing in iso date format

```
In [124]: dfd = DataFrame(randn(5, 2), columns=list('AB'))

In [125]: dfd['date'] = Timestamp('20130101')

In [126]: json = dfd.to_json(date_format='iso')

In [127]: json
'{"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.4108345112,"4":0.1320031703},"B":{"
```

Writing to a file, with a date index and a date column

```
In [128]: dfj2 = dfj.copy()

In [129]: dfj2['date'] = Timestamp('20130101')

In [130]: dfj2['ints'] = range(5)

In [131]: dfj2['bools'] = True

In [132]: dfj2.index = date_range('20130101',periods=5)

In [133]: dfj2.to_json('test.json')

In [134]: open('test.json').read()
'{"A":{"1356998400000000000":-1.2945235903,"1357084800000000000":0.2766617129,"1357171200000000000":-
```

18.2.2 Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a `DataFrame` if `typ` is not supplied or is `None`. To explicity force `Series` parsing, pass `typ=series`

- `filepath_or_buffer` : a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json
- `typ` : type of object to recover (series or frame), default 'frame'
- `orient` :

  Series :

  - default is `index`
  - allowed values are {`split`, `records`, `index`}

DataFrame

- default is `columns`
- allowed values are {`split`, `records`, `index`, `columns`, `values`}

The format of the JSON string

`split`	dict like {index -> [index], columns -> [columns], data -> [values]}
`records`	list like [{column -> value}, ... , {column -> value}]
`index`	dict like {index -> {column -> value}}
`columns`	dict like {column -> {index -> value}}
`values`	just the values array

- `dtype` : if True, infer dtypes, if a dict of column to dtype, then use those, if False, then don't infer dtypes at all, default is True, apply only to the data
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is True
- `convert_dates` : a list of columns to parse for dates; If True, then try to parse datelike columns, default is True
- `keep_default_dates` : boolean, default True. If parsing dates, then parse the default datelike columns
- `numpy` : direct decoding to numpy arrays. default is False; Note that the JSON ordering **MUST** be the same for each term if `numpy=True`
- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parsable.

The default of `convert_axes=True`, `dtype=True`, and `convert_dates=True` will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to `dtype`. `convert_axes` should only be set to `False` if you need to preserve string-like numbers (e.g. '1', '2') in an axes.

Warning: When reading JSON data, automatic coercing into dtypes has some quirks:
- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was `float` data will be converted to `integer` if it can be done safely, e.g. a column of `1.`
- bool columns will be converted to `integer` on reconstruction

Thus there are times where you may want to specify specific dtypes via the `dtype` keyword argument.

Reading from a JSON string

```
In [135]: pd.read_json(json)
```

```
          A         B                date
0 -1.206412  2.565646 2013-01-01 00:00:00
1  1.431256  1.340309 2013-01-01 00:00:00
2 -1.170299 -0.226169 2013-01-01 00:00:00
3  0.410835  0.813850 2013-01-01 00:00:00
4  0.132003 -0.827317 2013-01-01 00:00:00
```

Reading from a file

```
In [136]: pd.read_json('test.json')
```

```
                   A         B bools                 date  ints
2013-01-01 -1.294524  0.413738  True 2013-01-01 00:00:00     0
2013-01-02  0.276662 -0.472035  True 2013-01-01 00:00:00     1
```

```
2013-01-03 -0.013960 -0.362543  True 2013-01-01 00:00:00     2
2013-01-04 -0.006154 -0.923061  True 2013-01-01 00:00:00     3
2013-01-05  0.895717  0.805244  True 2013-01-01 00:00:00     4
```

Don't convert any data (but still convert axes and dates)

```
In [137]: pd.read_json('test.json',dtype=object).dtypes

A                object
B                object
bools            object
date     datetime64[ns]
ints             object
dtype: object
```

Specify how I want to convert data

```
In [138]: pd.read_json('test.json',dtype={'A' : 'float32', 'bools' : 'int8'}).dtypes

A               float32
B               float64
bools              int8
date     datetime64[ns]
ints              int64
dtype: object
```

I like my string indicies

```
In [139]: si = DataFrame(np.zeros((4, 4)),
   .....:                columns=range(4),
   .....:                index=[str(i) for i in range(4)])
   .....:

In [140]: si

   0  1  2  3
0  0  0  0  0
1  0  0  0  0
2  0  0  0  0
3  0  0  0  0

In [141]: si.index
Index([u'0', u'1', u'2', u'3'], dtype=object)

In [142]: si.columns
Int64Index([0, 1, 2, 3], dtype=int64)

In [143]: json = si.to_json()

In [144]: sij = pd.read_json(json,convert_axes=False)

In [145]: sij

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

```
In [146]: sij.index
Index([u'0', u'1', u'2', u'3'], dtype=object)

In [147]: sij.columns
Index([u'0', u'1', u'2', u'3'], dtype=object)
```

18.3 HTML

18.3.1 Reading HTML Content

Warning: We **highly encourage** you to read the *HTML parsing gotchas* regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

New in version 0.12. The top-level `read_html()` function can accept an HTML string/file/url and will parse HTML tables into list of pandas DataFrames. Let's look at a few examples.

Note: `read_html` returns a `list` of `DataFrame` objects, even if there is only a single table contained in the HTML content

---

Read a URL with no options

```
In [148]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'

In [149]: dfs = read_html(url)

In [150]: dfs

[<class 'pandas.core.frame.DataFrame'>
Int64Index: 518 entries, 0 to 517
Data columns (total 8 columns):
Bank Name                518  non-null values
City                     518  non-null values
ST                       518  non-null values
CERT                     518  non-null values
Acquiring Institution    518  non-null values
Closing Date             518  non-null values
Updated Date             518  non-null values
Loss Share Type          518  non-null values
dtypes: datetime64[ns](2), int64(1), object(5)]
```

Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string

```
In [151]: with open(file_path, 'r') as f:
   .....:     dfs = read_html(f.read())
   .....:

In [152]: dfs

[<class 'pandas.core.frame.DataFrame'>
Int64Index: 506 entries, 0 to 505
```

```
Data columns (total 7 columns):
Bank Name               506  non-null values
City                    506  non-null values
ST                      506  non-null values
CERT                    506  non-null values
Acquiring Institution   506  non-null values
Closing Date            506  non-null values
Updated Date            506  non-null values
dtypes: datetime64[ns](2), int64(1), object(4)]
```

You can even pass in an instance of `StringIO` if you so desire

```
In [153]: from cStringIO import StringIO

In [154]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:

In [155]: dfs = read_html(sio)

In [156]: dfs
```

```
[<class 'pandas.core.frame.DataFrame'>
Int64Index: 506 entries, 0 to 505
Data columns (total 7 columns):
Bank Name               506  non-null values
City                    506  non-null values
ST                      506  non-null values
CERT                    506  non-null values
Acquiring Institution   506  non-null values
Closing Date            506  non-null values
Updated Date            506  non-null values
dtypes: datetime64[ns](2), int64(1), object(4)]
```

Note: The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn't run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text

```
match = 'Metcalf Bank'
df_list = read_html(url, match=match)
```

Specify a header row (by default <th> elements are used to form the column index); if specified, the header row is taken from the data minus the parsed header elements (<th> elements).

```
dfs = read_html(url, header=0)
```

Specify an index column

```
dfs = read_html(url, index_col=0)
```

Specify a number of rows to skip

```
dfs = read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (`xrange` (Python 2 only) works as well)

```
dfs = read_html(url, skiprows=range(2))
```

Don't infer numeric and date types

```
dfs = read_html(url, infer_types=False)
```

Specify an HTML attribute

```
dfs1 = read_html(url, attrs={'id': 'table'})
dfs2 = read_html(url, attrs={'class': 'sortable'})
print np.array_equal(dfs1[0], dfs2[0])  # Should be True
```

Use some combination of the above

```
dfs = read_html(url, match='Metcalf Bank', index_col=0)
```

Read in pandas `to_html` output (with some loss of floating point precision)

```
df = DataFrame(randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = read_html(s, index_col=0)
```

The `lxml` backend will raise an error on a failed parse if that is the only parser you provide (if you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings)

```
dfs = read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])
```

or

```
dfs = read_html(url, 'Metcalf Bank', index_col=0, flavor='lxml')
```

However, if you have bs4 and html5lib installed and pass `None` or `['lxml', 'bs4']` then the parse will most likely succeed. Note that *as soon as a parse succeeds, the function will return*.

```
dfs = read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])
```

18.3.2 Writing to HTML files

`DataFrame` objects have an instance method `to_html` which renders the contents of the `DataFrame` as an HTML table. The function arguments are as in the method `to_string` described above.

Note: Not all of the possible options for `DataFrame.to_html` are shown here for brevity's sake. See `to_html()` for the full set of options.

```
In [157]: df = DataFrame(randn(2, 2))

In [158]: df

          0         1
0 -0.076467 -1.187678
1  1.130127 -1.436737

In [159]: print df.to_html()  # raw html
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
```

```
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.076467</td>
      <td>-1.187678</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 1.130127</td>
      <td>-1.436737</td>
    </tr>
  </tbody>
</table>
```

HTML:

The `columns` argument will limit the columns shown

```
In [160]: print df.to_html(columns=[0])
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.076467</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 1.130127</td>
    </tr>
  </tbody>
</table>
```

HTML:

`float_format` takes a Python callable to control the precision of floating point values

```
In [161]: print df.to_html(float_format='{0:.10f}'.format)
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.0764670176</td>
```

```
      <td>-1.1876775774</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 1.1301272978</td>
      <td>-1.4367373184</td>
    </tr>
  </tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off

```
In [162]: print df.to_html(bold_rows=False)
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>
      <td>-0.076467</td>
      <td>-1.187678</td>
    </tr>
    <tr>
      <td>1</td>
      <td> 1.130127</td>
      <td>-1.436737</td>
    </tr>
  </tbody>
</table>
```

The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing `'dataframe'` class.

```
In [163]: print df.to_html(classes=['awesome_table_class', 'even_more_awesome_class'])
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.076467</td>
      <td>-1.187678</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 1.130127</td>
      <td>-1.436737</td>
```

```
    </tr>
  </tbody>
</table>
```

Finally, the `escape` argument allows you to control whether the "<", ">" and "&" characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

```
In [164]: df = DataFrame({'a': list('&<>'), 'b': randn(3)})
```

Escaped:

```
In [165]: print df.to_html()
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td> &amp;</td>
      <td>-1.413681</td>
    </tr>
    <tr>
      <th>1</th>
      <td> &lt;</td>
      <td> 1.607920</td>
    </tr>
    <tr>
      <th>2</th>
      <td> &gt;</td>
      <td> 1.024180</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```
In [166]: print df.to_html(escape=False)
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td> &</td>
      <td>-1.413681</td>
    </tr>
    <tr>
      <th>1</th>
      <td> <</td>
```

```
      <td> 1.607920</td>
    </tr>
    <tr>
      <th>2</th>
      <td> ></td>
      <td> 1.024180</td>
    </tr>
  </tbody>
</table>
```

Note: Some browsers may not show a difference in the rendering of the previous two HTML tables.

18.4 Clipboard

A handy way to grab data is to use the `read_clipboard` method, which takes the contents of the clipboard buffer and passes them to the `read_table` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a DataFrame by calling:

```
clipdf = pd.read_clipboard()
```

```
In [167]: clipdf
```

```
   A  B  C
x  1  4  p
y  2  5  q
z  3  6  r
```

The `to_clipboard` method can be used to write the contents of a DataFrame to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a DataFrame into clipboard and reading it back.

```
In [168]: df=pd.DataFrame(randn(5,3))
```

```
In [169]: df
```

```
          0         1         2
0  0.569605  0.875906 -2.211372
1  0.974466 -2.006747 -0.410001
2 -0.078638  0.545952 -1.219217
3 -1.226825  0.769804 -1.281247
4 -0.727707 -0.121306 -0.097883
```

```
In [170]: df.to_clipboard()
```

```
In [171]: pd.read_clipboard()
```

```
          0         1         2
0  0.569605  0.875906 -2.211372
```

```
1  0.974466 -2.006747 -0.410001
2 -0.078638  0.545952 -1.219217
3 -1.226825  0.769804 -1.281247
4 -0.727707 -0.121306 -0.097883
```

We can see that we got the same content back, which we had earlier written to the clipboard.

Note: You may need to install xclip or xsel (with gtk or PyQt4 modules) on Linux to use these methods.

18.5 Pickling and serialization

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

```
In [172]: df
```

```
          0         1         2
0  0.569605  0.875906 -2.211372
1  0.974466 -2.006747 -0.410001
2 -0.078638  0.545952 -1.219217
3 -1.226825  0.769804 -1.281247
4 -0.727707 -0.121306 -0.097883
```

```
In [173]: df.to_pickle('foo.pkl')
```

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

```
In [174]: read_pickle('foo.pkl')
```

```
          0         1         2
0  0.569605  0.875906 -2.211372
1  0.974466 -2.006747 -0.410001
2 -0.078638  0.545952 -1.219217
3 -1.226825  0.769804 -1.281247
4 -0.727707 -0.121306 -0.097883
```

Warning: Loading pickled data received from untrusted sources can be unsafe.
See: http://docs.python.org/2.7/library/pickle.html

Note: These methods were previously `save` and `load`, now deprecated.

18.6 Excel files

The `read_excel` method can read Excel 2003 (`.xls`) and Excel 2007 (`.xlsx`) files using the `xlrd` Python module and use the same parsing code as the above to convert tabular data into a DataFrame. See the *cookbook* for some advanced strategies

---

Note: The prior method of accessing Excel is now deprecated as of 0.12, this will work but will be removed in a future version.

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

Replaced by

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

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. *read_excel* takes a *parse_cols* keyword to allow you to specify a subset of columns to parse.

If *parse_cols* is an integer, then it is assumed to indicate the last column to be parsed.

```
read_excel('path_to_file.xls', 'Sheet1', parse_cols=2, index_col=None, na_values=['NA'])
```

If *parse_cols* is a list of integers, then it is assumed to be the file column indices to be parsed.

```
read_excel('path_to_file.xls', Sheet1', parse_cols=[0, 2, 3], index_col=None, na_values=['NA'])
```

To write a DataFrame object to a sheet of an Excel file, you can use the `to_excel` instance method. The arguments are largely the same as `to_csv` described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the DataFrame should be written. For example:

```
df.to_excel('path_to_file.xlsx', sheet_name='sheet1')
```

Files with a `.xls` extension will be written using `xlwt` and those with a `.xlsx` extension will be written using `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()
```

18.7 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the *cookbook* for some advanced strategies

---

Note: `PyTables` 3.0.0 was recently released to enables support for Python 3. Pandas should be fully compatible (and previously written stores should be backwards compatible) with all `PyTables` >= 2.3

```
In [175]: store = HDFStore('store.h5')

In [176]: 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 [177]: index = date_range('1/1/2000', periods=8)

In [178]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [179]: df = DataFrame(randn(8, 3), index=index,
   .....:                columns=['A', 'B', 'C'])
   .....:

In [180]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:            major_axis=date_range('1/1/2000', periods=5),
   .....:            minor_axis=['A', 'B', 'C', 'D'])
   .....:

# store.put('s', s) is an equivalent method
In [181]: store['s'] = s

In [182]: store['df'] = df

In [183]: store['wp'] = wp

# the type of stored data
In [184]: store.root.wp._v_attrs.pandas_type
'wide'

In [185]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df             frame        (shape->[8,3])
/s              series       (shape->[5])
/wp             wide         (shape->[2,5,4])
```

In a current or later Python session, you can retrieve stored objects:

```
# store.get('df') is an equivalent method
In [186]: store['df']

                   A         B         C
2000-01-01  0.149748 -0.732339  0.687738
2000-01-02  0.176444  0.403310 -0.154951
2000-01-03  0.301624 -2.179861 -1.369849
2000-01-04 -0.954208  1.462696 -1.743161
2000-01-05 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780
2000-01-07  0.014871  3.357427 -0.317441
2000-01-08 -1.236269  0.896171 -0.487602

# dotted (attribute) access provides get as well
In [187]: store.df

                   A         B         C
2000-01-01  0.149748 -0.732339  0.687738
2000-01-02  0.176444  0.403310 -0.154951
2000-01-03  0.301624 -2.179861 -1.369849
2000-01-04 -0.954208  1.462696 -1.743161
2000-01-05 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780
2000-01-07  0.014871  3.357427 -0.317441
```

```
2000-01-08 -1.236269  0.896171 -0.487602
```

Deletion of the object specified by the key

```
# store.remove('wp') is an equivalent method
In [188]: del store['wp']

In [189]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df             frame        (shape->[8,3])
/s              series       (shape->[5])
```

Closing a Store, Context Manager

```
# closing a store
In [190]: store.close()

# Working with, and automatically closing the store with the context
# manager
In [191]: with get_store('store.h5') as store:
   .....:       store.keys()
   .....:
```

18.7.1 Read/Write API

`HDFStore` supports an top-level API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work. (new in 0.11.0)

```
In [192]: df_tl = DataFrame(dict(A=range(5), B=range(5)))

In [193]: df_tl.to_hdf('store_tl.h5','table',append=True)

In [194]: read_hdf('store_tl.h5', 'table', where = ['index>2'])

   A  B
3  3  3
4  4  4
```

18.7.2 Storer Format

The examples above show storing using `put`, which write the HDF5 to `PyTables` in a fixed array format, called the `storer` format. These types of stores are are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. These offer very fast writing and slightly faster reading than `table` stores.

> **Warning:** A `storer` format will raise a `TypeError` if you try to retrieve using a `where` .

```
DataFrame(randn(10,2)).to_hdf('test_storer.h5','df')

pd.read_hdf('test_storer.h5','df',where='index>5')
TypeError: cannot pass a where specification when reading a non-table
           this store must be selected in its entirety
```

18.7.3 Table Format

`HDFStore` supports another `PyTables` format on disk, the `table` format. Conceptually a `table` is shaped very much like a DataFrame, with rows and columns. A `table` may be appended to in the same or other sessions. In addition, delete & query type operations are supported.

```
In [195]: store = HDFStore('store.h5')

In [196]: df1 = df[0:4]

In [197]: df2 = df[4:]

# append data (creates a table automatically)
In [198]: store.append('df', df1)

In [199]: store.append('df', df2)

In [200]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])

# select the entire object
In [201]: store.select('df')

                   A         B         C
2000-01-01  0.149748 -0.732339  0.687738
2000-01-02  0.176444  0.403310 -0.154951
2000-01-03  0.301624 -2.179861 -1.369849
2000-01-04 -0.954208  1.462696 -1.743161
2000-01-05 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780
2000-01-07  0.014871  3.357427 -0.317441
2000-01-08 -1.236269  0.896171 -0.487602

# the type of stored data
In [202]: store.root.df._v_attrs.pandas_type
'frame_table'
```

Note: You can also create a `table` by passing `table=True` to a `put` operation.

18.7.4 Hierarchical Keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. `foo/bar/bah`), which will generate a hierarchy of sub-stores (or `Groups` in PyTables parlance). Keys can be specified with out the leading '/' and are ALWAYS absolute (e.g. 'foo' refers to '/foo'). Removal operations can remove everying in the sub-store and BELOW, so be *careful*.

```
In [203]: store.put('foo/bar/bah', df)

In [204]: store.append('food/orange', df)

In [205]: store.append('food/apple',  df)
```

```
In [206]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/apple             frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah            frame        (shape->[8,3])

# a list of keys are returned
In [207]: store.keys()
['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [208]: store.remove('food')

In [209]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah            frame        (shape->[8,3])
```

18.7.5 Storing Mixed Types in a Table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent appends will truncate strings at this length.

Passing `min_itemsize={'values': size}` as a parameter to append will set a larger minimum for the string columns. Storing `floats, strings, ints, bools, datetime64` are currently supported. For string columns, passing `nan_rep = 'nan'` to append will change the default nan representation on disk (which converts to/from *np.nan*), this defaults to *nan*.

```
In [210]: df_mixed = DataFrame({ 'A' : randn(8),
   .....:                        'B' : randn(8),
   .....:                        'C' : np.array(randn(8),dtype='float32'),
   .....:                        'string' :'string',
   .....:                        'int' : 1,
   .....:                        'bool' : True,
   .....:                        'datetime64' : Timestamp('20010102')},
   .....:                       index=range(8))
   .....:

In [211]: df_mixed.ix[3:5,['A', 'B', 'string', 'datetime64']] = np.nan

In [212]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})

In [213]: df_mixed1 = store.select('df_mixed')

In [214]: df_mixed1

          A         B         C  bool          datetime64  int  string
0 -0.064034 -0.744471  1.682706  True 2001-01-02 00:00:00    1  string
1 -1.282782  0.758527 -1.717693  True 2001-01-02 00:00:00    1  string
2  0.781836  1.729689  0.888782  True 2001-01-02 00:00:00    1  string
3       NaN       NaN  0.228440  True                 NaT    1     NaN
4       NaN       NaN  0.901805  True                 NaT    1     NaN
```

```
5       NaN       NaN  1.171216  True                  NaT    1     NaN
6  0.583787  1.846883  0.520260  True 2001-01-02 00:00:00    1  string
7  0.221471 -1.328865 -1.197071  True 2001-01-02 00:00:00    1  string

In [215]: df_mixed1.get_dtype_counts()

bool              1
datetime64[ns]    1
float32           1
float64           2
int64             1
object            1
dtype: int64

# we have provided a minimum string column size
In [216]: store.root.df_mixed.table

/df_mixed/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
  "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
  "values_block_2": Int64Col(shape=(1,), dflt=0, pos=3),
  "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
  "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
  "values_block_5": StringCol(itemsize=50, shape=(1,), dflt='', pos=6)}
  byteorder := 'little'
  chunkshape := (689,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

18.7.6 Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.

```
In [217]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   .....:                            ['one', 'two', 'three']],
   .....:                    labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   .....:                            [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:                    names=['foo', 'bar'])
   .....:

In [218]: df_mi = DataFrame(np.random.randn(10, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....:

In [219]: df_mi

                  A         B         C
foo bar
foo one   -1.066969 -0.303421 -0.858447
    two    0.306996 -0.028665  0.384316
    three  1.574159  1.588931  0.476720
bar one    0.473424 -0.242861 -0.014805
    two   -0.284319  0.650776 -1.461665
baz two   -1.137707 -0.891060 -0.693921
```

```
    three  1.613616  0.464000  0.227371
qux one   -0.496922  0.306389 -2.290613
    two   -1.134623 -1.561819 -0.260838
    three  0.281957  1.523962 -0.902937

In [220]: store.append('df_mi',df_mi)

In [221]: store.select('df_mi')

                  A         B         C
foo bar
foo one   -1.066969 -0.303421 -0.858447
    two    0.306996 -0.028665  0.384316
    three  1.574159  1.588931  0.476720
bar one    0.473424 -0.242861 -0.014805
    two   -0.284319  0.650776 -1.461665
baz two   -1.137707 -0.891060 -0.693921
    three  1.613616  0.464000  0.227371
qux one   -0.496922  0.306389 -2.290613
    two   -1.134623 -1.561819 -0.260838
    three  0.281957  1.523962 -0.902937

# the levels are automatically included as data columns
In [222]: store.select('df_mi', Term('foo=bar'))

                A         B         C
foo bar
bar one  0.473424 -0.242861 -0.014805
    two -0.284319  0.650776 -1.461665
```

18.7.7 Querying a Table

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

A query is specified using the `Term` class under the hood.

- 'index' and 'columns' are supported indexers of a DataFrame
- 'major_axis', 'minor_axis', and 'items' are supported indexers of the Panel

Valid terms can be created from `dict, list, tuple, or string`. Objects can be embeded as values. Allowed operations are: `<, <=, >, >=, =, !=`. `=` will be inferred as an implicit set operation (e.g. if 2 or more values are provided). The following are all valid terms.

- `dict(field = 'index', op = '>', value = '20121114')`
- `('index', '>', '20121114')`
- `'index > 20121114'`
- `('index', '>', datetime(2012, 11, 14))`
- `('index', ['20121114', '20121115'])`
- `('major_axis', '=', Timestamp('2012/11/14'))`
- `('minor_axis', ['A', 'B'])`

Queries are built up using a list of `Terms` (currently only **anding** of terms is supported). An example query for a panel might be specified as follows. `['major_axis>20000102', ('minor_axis', '=', ['A', 'B']) ]`. This is roughly translated to: *major_axis must be greater than the date 20000102 and the minor_axis must be A or B*

```
In [223]: store.append('wp',wp)

In [224]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table   (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df_mi                  frame_table   (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->
/df_mixed               frame_table   (typ->appendable,nrows->8,ncols->7,indexers->[index])
/wp                     wide_table    (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_
/foo/bar/bah            frame         (shape->[8,3])

In [225]: store.select('wp', [ Term('major_axis>20000102'), Term('minor_axis', '=', ['A', 'B']) ])

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B
```

The `columns` keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a `Term('columns', list_of_columns_to_filter)`:

```
In [226]: store.select('df', columns=['A', 'B'])

                   A         B
2000-01-01  0.149748 -0.732339
2000-01-02  0.176444  0.403310
2000-01-03  0.301624 -2.179861
2000-01-04 -0.954208  1.462696
2000-01-05 -0.826591 -0.345352
2000-01-06  0.690579  0.995761
2000-01-07  0.014871  3.357427
2000-01-08 -1.236269  0.896171
```

`start` and `stop` parameters can be specified to limit the total search space. These are in terms of the total number of rows in a table.

```
# this is effectively what the storage of a Panel looks like
In [227]: wp.to_frame()

                     Item1     Item2
major      minor
2000-01-01 A     -0.082240  0.408204
           B     -2.182937 -1.048089
           C      0.380396 -0.025747
           D      0.084844 -0.988387
2000-01-02 A      0.432390  0.094055
           B      1.519970  1.262731
           C     -0.493662  1.289997
           D      0.600178  0.082423
2000-01-03 A      0.274230 -0.055758
           B      0.132885  0.536580
           C     -0.023688 -0.489682
           D      2.410179  0.369374
```

```
2000-01-04 A      1.450520 -0.034571
           B      0.206053 -2.484478
           C     -0.251905 -0.281461
           D     -2.213588  0.030711
2000-01-05 A      1.063327  0.109121
           B      1.266143  1.126203
           C      0.299368 -0.977349
           D     -0.863838  1.474071

# limiting the search
In [228]: store.select('wp',[ Term('major_axis>20000102'),
   .....:                     Term('minor_axis', '=', ['A','B']) ],
   .....:              start=0, stop=10)
   .....:

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 1 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to B
```

18.7.8 Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`. **Indexes are automagically created (starting 0.10.1)** on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to `append`.

```
# we have automagically already created an index (in the first section)
In [229]: i = store.root.df.table.cols.index.index

In [230]: i.optlevel, i.kind
(6, 'medium')

# change an index by passing new parameters
In [231]: store.create_table_index('df', optlevel=9, kind='full')

In [232]: i = store.root.df.table.cols.index.index

In [233]: i.optlevel, i.kind
(9, 'full')
```

18.7.9 Query via Data Columns

You can designate (and index) certain columns that you want to be able to perform queries (other than the *indexable* columns, which you can always query). For instance say you want to perform this common operation, on-disk, and return just the frame that matches this query. You can specify `data_columns = True` to force all columns to be data_columns

```
In [234]: df_dc = df.copy()

In [235]: df_dc['string'] = 'foo'

In [236]: df_dc.ix[4:6,'string'] = np.nan
```

```
In [237]: df_dc.ix[7:9,'string'] = 'bar'

In [238]: df_dc['string2'] = 'cool'

In [239]: df_dc

                   A         B         C string string2
2000-01-01  0.149748 -0.732339  0.687738    foo    cool
2000-01-02  0.176444  0.403310 -0.154951    foo    cool
2000-01-03  0.301624 -2.179861 -1.369849    foo    cool
2000-01-04 -0.954208  1.462696 -1.743161    foo    cool
2000-01-05 -0.826591 -0.345352  1.314232    NaN    cool
2000-01-06  0.690579  0.995761  2.396780    NaN    cool
2000-01-07  0.014871  3.357427 -0.317441    foo    cool
2000-01-08 -1.236269  0.896171 -0.487602    bar    cool

# on-disk operations
In [240]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])

In [241]: store.select('df_dc', [ Term('B>0') ])

                   A         B         C string string2
2000-01-02  0.176444  0.403310 -0.154951    foo    cool
2000-01-04 -0.954208  1.462696 -1.743161    foo    cool
2000-01-06  0.690579  0.995761  2.396780    NaN    cool
2000-01-07  0.014871  3.357427 -0.317441    foo    cool
2000-01-08 -1.236269  0.896171 -0.487602    bar    cool

# getting creative
In [242]: store.select('df_dc', ['B > 0', 'C > 0', 'string == foo'])

Empty DataFrame
Columns: [A, B, C, string, string2]
Index: []

# this is in-memory version of this type of selection
In [243]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]

Empty DataFrame
Columns: [A, B, C, string, string2]
Index: []

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [244]: store.root.df_dc.table

/df_dc/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2),
  "C": Float64Col(shape=(), dflt=0.0, pos=3),
  "string": StringCol(itemsize=3, shape=(), dflt='', pos=4),
  "string2": StringCol(itemsize=4, shape=(), dflt='', pos=5)}
  byteorder := 'little'
  chunkshape := (1680,)
  autoindex := True
  colindexes := {
```

```
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

There is some performance degredation by making lots of columns into *data columns*, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!)

18.7.10 Iterator

Starting in 0.11, you can pass, `iterator=True` or `chunksize=number_in_a_chunk` to `select` and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```
In [245]: for df in store.select('df', chunksize=3):
   .....:     print df
   .....:
                   A         B         C
2000-01-01  0.149748 -0.732339  0.687738
2000-01-02  0.176444  0.403310 -0.154951
2000-01-03  0.301624 -2.179861 -1.369849
                   A         B         C
2000-01-04 -0.954208  1.462696 -1.743161
2000-01-05 -0.826591 -0.345352  1.314232
2000-01-06  0.690579  0.995761  2.396780
                   A         B         C
2000-01-07  0.014871  3.357427 -0.317441
2000-01-08 -1.236269  0.896171 -0.487602
```

Note: New in version 0.12. You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in read_hdf('store.h5','df', chunsize=3):
    print df
```

Note, that the chunksize keyword applies to the **returned** rows. So if you are doing a query, then that set will be subdivided and returned in the iterator. Keep in mind that if you do not pass a `where` selection criteria then the `nrows` of the table are considered.

18.7.11 Advanced Queries

Select a Single Column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector (coming soon)

```
In [246]: store.select_column('df_dc', 'index')

0   2000-01-01 00:00:00
1   2000-01-02 00:00:00
2   2000-01-03 00:00:00
3   2000-01-04 00:00:00
```

```
4   2000-01-05 00:00:00
5   2000-01-06 00:00:00
6   2000-01-07 00:00:00
7   2000-01-08 00:00:00
dtype: datetime64[ns]
```

```
In [247]: store.select_column('df_dc', 'string')
```

```
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
dtype: object
```

Replicating or

`not` and `or` conditions are unsupported at this time; however, `or` operations are easy to replicate, by repeatedly applying the criteria to the table, and then `concat` the results.

```
In [248]: crit1 = [ Term('B>0'), Term('C>0'), Term('string=foo') ]
```

```
In [249]: crit2 = [ Term('B<0'), Term('C>0'), Term('string=foo') ]
```

```
In [250]: concat([store.select('df_dc',c) for c in [crit1, crit2]])
```

```
                   A         B         C string string2
2000-01-01  0.149748 -0.732339  0.687738    foo    cool
```

Storer Object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [251]: store.get_storer('df_dc').nrows
8
```

18.7.12 Multiple Table Queries

New in 0.10.1 are the methods `append_to_multiple` and `select_as_multiple`, that can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table's index. You can then perform a very fast query on the selector table, yet get lots of data back. This method works similar to having a very wide table, but is more efficient in terms of queries.

Note, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. This means, append to the tables in the same order; `append_to_multiple` splits a single object to multiple tables, given a specification (as a dictionary). This dictionary is a mapping of the table names to the 'columns' you want included in that table. Pass a *None* for a single table (optional) to let it have the remaining columns. The argument `selector` defines which table is the selector table.

```
In [252]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
   .....:                                columns=['A', 'B', 'C', 'D', 'E', 'F'])
   .....:
```

```
In [253]: df_mt['foo'] = 'bar'

# you can also create the tables individually
In [254]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None },
   .....:                          df_mt, selector='df1_mt')
   .....:

In [255]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df1_mt                 frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt                 frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_dc                  frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,st
/df_mi                  frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->
/df_mixed               frame_table  (typ->appendable,nrows->8,ncols->7,indexers->[index])
/wp                     wide_table   (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_
/foo/bar/bah            frame        (shape->[8,3])

# indiviual tables were created
In [256]: store.select('df1_mt')

                   A         B
2000-01-01  0.068159 -0.057873
2000-01-02  0.782098 -1.069094
2000-01-03  0.379319 -0.008434
2000-01-04  0.040403 -0.507516
2000-01-05  1.488753 -0.896484
2000-01-06  2.121453  0.597701
2000-01-07 -0.928797 -0.308853
2000-01-08 -1.553902  2.015523

In [257]: store.select('df2_mt')

                   C         D         E         F  foo
2000-01-01 -0.368204 -1.144073  0.861209  0.800193  bar
2000-01-02 -1.099248  0.255269  0.009750  0.661084  bar
2000-01-03  1.952541 -1.056652  0.533946 -1.226970  bar
2000-01-04 -0.230096  0.394500 -1.934370 -1.652499  bar
2000-01-05  0.576897  1.146000  1.487349  0.604603  bar
2000-01-06  0.563700  0.967661 -1.057909  1.375020  bar
2000-01-07 -0.681087  0.377953  0.493672 -2.461467  bar
2000-01-08 -1.833722  1.771740 -0.670027  0.049307  bar

# as a multiple
In [258]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                          selector = 'df1_mt')
   .....:

                   A         B       C         D         E        F  foo
2000-01-06  2.121453  0.597701  0.5637  0.967661 -1.057909  1.37502  bar
```

18.7.13 Delete from a Table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the `PyTables` deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very

expensive operation depending on the orientation of your data. This is especially true in higher dimensional objects (`Panel` and `Panel4D`). To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the `indexables`.

Data is ordered (on the disk) in terms of the `indexables`. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- **date_1**
  - id_1
  - id_2
  - .
  - id_n
- **date_2**
  - id_1
  - .
  - id_n

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is removed, then the following data moved. On the other hand a delete operation on the `minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

```
# returns the number of rows deleted
In [259]: store.remove('wp', 'major_axis>20000102' )
12

In [260]: store.select('wp')

<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

Please note that HDF5 **DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again **WILL TEND TO INCREASE THE FILE SIZE**. To *clean* the file, use `ptrepack` (see below).

18.7.14 Compression

`PyTables` allows the stored data to be compressed. Tthis applies to all kinds of stores, not just tables.

- Pass `complevel=int` for a compression level (1-9, with 0 being no compression, and the default)
- Pass `complib=lib` where lib is any of `zlib, bzip2, lzo, blosc` for whichever compression library you prefer.

`HDFStore` will use the file based compression scheme if no overriding `complib` or `complevel` options are provided. `blosc` offers very fast compression, and is my most used. Note that `lzo` and `bzip2` may not be installed (by Python) by default.

Compression for all objects within the file

- `store_compressed = HDFStore('store_compressed.h5', complevel=9, complib='blosc')`

Or on-the-fly compression (this only applies to tables). You can turn off file compression for a specific table by passing `complevel=0`

- `store.append('df', df, complib='zlib', complevel=5)`

ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

- `ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5`

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Aalternatively, one can simply remove the file and write again, or use the `copy` method.

18.7.15 Notes & Caveats

- Once a `table` is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
- If a row has `np.nan` for **EVERY COLUMN** (having a `nan` in a string, or a `NaT` in a datetime-like column counts as having a value), then those rows **WILL BE DROPPED IMPLICITLY**. This limitation *may* be addressed in the future.
- `HDFStore` is **not-threadsafe for writing**. The underlying `PyTables` only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the issue <https://github.com/pydata/pandas/issues/2397> for more information.
- `PyTables` only supports fixed-width string columns in `tables`. The sizes of a string based indexing column (e.g. *columns* or *minor_axis*) are determined as the maximum size of the elements in that axis or by passing the parameter

18.7.16 DataTypes

`HDFStore` will map an object dtype to the `PyTables` underlying dtype. This means the following types are known to work:

- floating : `float64, float32, float16` *(using* `np.nan` *to represent invalid values)*
- integer : `int64, int32, int8, uint64, uint32, uint8`
- bool
- datetime64[ns] *(using* `NaT` *to represent invalid values)*
- object : `strings` *(using* `np.nan` *to represent invalid values)*

Currently, `unicode` and `datetime` columns (represented with a dtype of `object`), **WILL FAIL**. In addition, even though a column may look like a `datetime64[ns]`, if it contains `np.nan`, this **WILL FAIL**. You can try to convert datetimelike columns to proper `datetime64[ns]` columns, that possibly contain `NaT` to represent invalid values. (Some of these issues have been addressed and these conversion may not be necessary in future versions of pandas)

```
In [261]: import datetime

In [262]: df = DataFrame(dict(datelike=Series([datetime.datetime(2001, 1, 1),
   .....:                                      datetime.datetime(2001, 1, 2), np.nan])))
```

```
   .....:

In [263]: df

              datelike
0  2001-01-01 00:00:00
1  2001-01-02 00:00:00
2                  NaN

In [264]: df.dtypes

datelike    object
dtype: object

# to convert
In [265]: df['datelike'] = Series(df['datelike'].values, dtype='M8[ns]')

In [266]: df

             datelike
0 2001-01-01 00:00:00
1 2001-01-02 00:00:00
2                 NaT

In [267]: df.dtypes

datelike    datetime64[ns]
dtype: object
```

18.7.17 String Columns

The underlying implementation of `HDFStore` uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specifiy the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data_columns* to have this min_itemsize.

Starting in 0.11, passing a `min_itemsize` dict will cause all passed columns to be created as *data_columns* automatically.

Note: If you are not passing any *data_columns*, then the min_itemsize will be the maximum of the length of any string passed

---

```
In [268]: dfs = DataFrame(dict(A = 'foo', B = 'bar'),index=range(5))

In [269]: dfs

     A    B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
```

```
4  foo  bar

# A and B have a size of 30
In [270]: store.append('dfs', dfs, min_itemsize = 30)

In [271]: store.get_storer('dfs').table

/dfs/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=30, shape=(2,), dflt='', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [272]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })

In [273]: store.get_storer('dfs2').table

/dfs2/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=3, shape=(1,), dflt='', pos=1),
  "A": StringCol(itemsize=30, shape=(), dflt='', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

18.7.18 External Compatibility

`HDFStore` write storer objects in specific formats suitable for producing loss-less roundtrips to pandas objects. For external compatibility, `HDFStore` can read native `PyTables` format tables. It is possible to write an `HDFStore` object that can easily be imported into `R` using the `rhdf5` library. Create a table format store like this:

```
In [274]: store_export = HDFStore('export.h5')

In [275]: store_export.append('df_dc', df_dc, data_columns=df_dc.columns)

In [276]: store_export

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

18.7.19 Backwards Compatibility

0.10.1 of `HDFStore` can read tables created in a prior version of pandas, however query terms using the prior (undocumented) methodology are unsupported. `HDFStore` will issue a warning if you try to use a legacy-format file.

You must read in the entire file and write it out using the new format, using the method `copy` to take advantage of the updates. The group attribute `pandas_version` contains the version information. `copy` takes a number of options, please see the docstring.

```
# a legacy store
In [277]: legacy_store = HDFStore(legacy_file_path,'r')

In [278]: legacy_store

<class 'pandas.io.pytables.HDFStore'>
File path: /home/docbuild/CI/pandas/doc/source/_static/legacy_0.10.h5
/a                      series       (shape->[30])
/b                      frame        (shape->[30,4])
/df1_mixed              frame_table [0.10.0] (typ->appendable,nrows->30,ncols->11,indexers->[index
/p1_mixed               wide_table  [0.10.0] (typ->appendable,nrows->120,ncols->9,indexers->[major
/p4d_mixed              ndim_table  [0.10.0] (typ->appendable,nrows->360,ncols->9,indexers->[items
/foo/bar                wide         (shape->[3,30,4])

# copy (and return the new handle)
In [279]: new_store = legacy_store.copy('store_new.h5')

In [280]: new_store

<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a                      series       (shape->[30])
/b                      frame        (shape->[30,4])
/df1_mixed              frame_table  (typ->appendable,nrows->30,ncols->11,indexers->[index])
/p1_mixed               wide_table   (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,mi
/p4d_mixed              wide_table   (typ->appendable,nrows->360,ncols->9,indexers->[items,major_a
/foo/bar                wide         (shape->[3,30,4])

In [281]: new_store.close()
```

18.7.20 Performance

- `Tables` come with a writing performance penalty as compared to regular stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will signficantly lower your memory usage on writing.
- You can pass `expectedrows=<int>` to the first `append`, to set the TOTAL number of expected rows that `PyTables` will expected. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See <http://stackoverflow.com/questions/14355151/how-to-make-pandas-hdfstore-put-operation-faster/14370190#14370190> for more information and some solutions.

18.7.21 Experimental

HDFStore supports `Panel4D` storage.

```
In [282]: p4d = Panel4D({ 'l1' : wp })

In [283]: p4d

<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 1 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: l1 to l1
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

In [284]: store.append('p4d', p4d)

In [285]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df1_mt                 frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt                 frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_dc                  frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,st
/df_mi                  frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->
/df_mixed               frame_table  (typ->appendable,nrows->8,ncols->7,indexers->[index])
/dfs                    frame_table  (typ->appendable,nrows->5,ncols->2,indexers->[index])
/dfs2                   frame_table  (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])
/p4d                    wide_table   (typ->appendable,nrows->40,ncols->1,indexers->[items,major_axis,
/wp                     wide_table   (typ->appendable,nrows->8,ncols->2,indexers->[major_axis,minor_a
/foo/bar/bah            frame        (shape->[8,3])
```

These, by default, index the three axes `items, major_axis, minor_axis`. On an `AppendableTable` it is possible to setup with the first append a different indexing scheme, depending on how you want to store your data. Pass the `axes` keyword with a list of dimensions (currently must by exactly 1 less than the total dimensions of the object). This cannot be changed after table creation.

```
In [286]: store.append('p4d2', p4d, axes=['labels', 'major_axis', 'minor_axis'])

In [287]: store

<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df1_mt                 frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt                 frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_dc                  frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,st
/df_mi                  frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->
/df_mixed               frame_table  (typ->appendable,nrows->8,ncols->7,indexers->[index])
/dfs                    frame_table  (typ->appendable,nrows->5,ncols->2,indexers->[index])
/dfs2                   frame_table  (typ->appendable,nrows->5,ncols->2,indexers->[index],dc->[A])
/p4d                    wide_table   (typ->appendable,nrows->40,ncols->1,indexers->[items,major_axis,
/p4d2                   wide_table   (typ->appendable,nrows->20,ncols->2,indexers->[labels,major_axis
/wp                     wide_table   (typ->appendable,nrows->8,ncols->2,indexers->[major_axis,minor_a
/foo/bar/bah            frame        (shape->[8,3])

In [288]: store.select('p4d2', [ Term('labels=l1'), Term('items=Item1'), Term('minor_axis=A_big_strin

<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 0 (labels) x 1 (items) x 0 (major_axis) x 0 (minor_axis)
Labels axis: None
```

```
Items axis: Item1 to Item1
Major_axis axis: None
Minor_axis axis: None
```

18.8 SQL Queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. These wrappers only support the Python database adapters which respect the Python DB-API. See some *cookbook examples* for some advanced strategies

For example, suppose you want to query some data with different types from a table such as:

id	Date	Col_1	Col_2	Col_3
26	2012-10-18	X	25.7	True
42	2012-10-19	Y	-12.4	False
63	2012-10-20	Z	5.73	True

Functions from `pandas.io.sql` can extract some data into a DataFrame. In the following example, we use the SQlite SQL database engine. You can use a temporary SQLite database where data are stored in "memory". Just do:

```
import sqlite3
from pandas.io import sql
# Create your connection.
cnx = sqlite3.connect(':memory:')
```

Let `data` be the name of your SQL table. With a query and your database connection, just use the `read_frame()` function to get the query results into a DataFrame:

```
In [289]: sql.read_frame("SELECT * FROM data;", cnx)

   id                 date Col_1  Col_2  Col_3
0  26  2010-10-18 00:00:00     X  27.50      1
1  42  2010-10-19 00:00:00     Y -12.50      0
2  63  2010-10-20 00:00:00     Z   5.73      1
```

You can also specify the name of the column as the DataFrame index:

```
In [290]: sql.read_frame("SELECT * FROM data;", cnx, index_col='id')

                   date Col_1  Col_2  Col_3
id
26  2010-10-18 00:00:00     X  27.50      1
42  2010-10-19 00:00:00     Y -12.50      0
63  2010-10-20 00:00:00     Z   5.73      1

In [291]: sql.read_frame("SELECT * FROM data;", cnx, index_col='date')

                     id Col_1  Col_2  Col_3
date
2010-10-18 00:00:00  26     X  27.50      1
2010-10-19 00:00:00  42     Y -12.50      0
2010-10-20 00:00:00  63     Z   5.73      1
```

Of course, you can specify a more "complex" query.

```
In [292]: sql.read_frame("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", cnx)
```

```
   id Col_1  Col_2
0  42     Y  -12.5
```

There are a few other available functions:

- `tquery` returns a list of tuples corresponding to each row.
- `uquery` does the same thing as tquery, but instead of returning results it returns the number of related rows.
- `write_frame` writes records stored in a DataFrame into the SQL table.
- `has_table` checks if a given SQLite table exists.

Note: For now, writing your DataFrame into a database works only with **SQLite**. Moreover, the **index** will currently be **dropped**.

18.9 STATA Format

18.9.1 Writing to STATA format

The method `to_stata()` will write a DataFrame into a .dta file. The format version of this file is always the latest one, 115.

```
In [293]: df = DataFrame(randn(10, 2), columns=list('AB'))

In [294]: df.to_stata('stata.dta')
```

18.9.2 Reading from STATA format

New in version 0.12. The top-level function `read_stata` will read a dta format file and return a DataFrame: The class `StataReader` will read the header of the given dta file at initialization. Its method `data()` will read the observations, converting them to a DataFrame which is returned:

```
In [295]: pd.read_stata('stata.dta')

   index         A         B
0      0 -0.521493 -3.201750
1      1  0.792716  0.146111
2      2  1.903247 -0.747169
3      3 -0.309038  0.393876
4      4  1.861468  0.936527
5      5  1.255746 -2.655452
6      6  1.219492  0.062297
7      7 -0.110388 -1.184357
8      8 -0.558081  0.077849
9      9  0.629498 -1.035260
```

Currently the `index` is retrieved as a column on read back.

The parameter `convert_categoricals` indicates wheter value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `variable_labels`, which requires data to be called before (see `pandas.io.stata.StataReader`).

The StataReader supports .dta Formats 104, 105, 108, 113-115. Alternatively, the function `read_stata()` can be used

18.10 Data Reader

Functions from `pandas.io.data` extract data from various Internet sources into a DataFrame. Currently the following sources are supported:

- Yahoo! Finance
- Google Finance
- St. Louis FED (FRED)
- Kenneth French's data library

It should be noted, that various sources support different kinds of data, so not all sources implement the same methods and the data elements returned might also differ.

18.10.1 Yahoo! Finance

```
In [296]: import pandas.io.data as web

In [297]: start = datetime.datetime(2010, 1, 1)

In [298]: end = datetime.datetime(2013, 01, 27)

In [299]: f=web.DataReader("F", 'yahoo', start, end)

In [300]: f.ix['2010-01-04']

Open               10.17
High               10.28
Low                10.05
Close              10.28
Volume       60855800.00
Adj Close            9.75
Name: 2010-01-04 00:00:00, dtype: float64
```

18.10.2 Google Finance

```
In [301]: import pandas.io.data as web

In [302]: start = datetime.datetime(2010, 1, 1)

In [303]: end = datetime.datetime(2013, 01, 27)

In [304]: f=web.DataReader("F", 'google', start, end)

In [305]: f.ix['2010-01-04']

Open         10.17
High         10.28
Low          10.05
Close        10.28
Volume    60855796
Name: 2010-01-04 00:00:00, dtype: object
```

18.10.3 FRED

```
In [306]: import pandas.io.data as web

In [307]: start = datetime.datetime(2010, 1, 1)

In [308]: end = datetime.datetime(2013, 01, 27)

In [309]: gdp=web.DataReader("GDP", "fred", start, end)

In [310]: gdp.ix['2013-01-01']

GDP    16535.3
Name: 2013-01-01 00:00:00, dtype: float64
```

18.10.4 Fama/French

Tthe dataset names are listed at Fama/French Data Library)

```
In [311]: import pandas.io.data as web

In [312]: ip=web.DataReader("5_Industry_Portfolios", "famafrench")

In [313]: ip[4].ix[192607]

1 Cnsmr    5.43
2 Manuf    2.73
3 HiTec    1.83
4 Hlth     1.64
5 Other    2.15
Name: 192607, dtype: float64
```

CHAPTER

NINETEEN

ENHANCING PERFORMANCE

19.1 Cython (Writing C extensions for pandas)

For many use cases writing pandas in pure python and numpy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizeable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in python, for example trying to remove for loops and making use of numpy vectorization, it's always worth optimising in python first.

This tutorial walks through a "typical" process of cythonizing a slow computation. We use an example from the cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure python.

19.1.1 Pure python

We have a DataFrame to which we want to apply a function row-wise.

```
In [1]: df = DataFrame({'a': randn(1000), 'b': randn(1000),'N': randint(100, 1000, (1000)), 'x': 'x'
```

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

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 0 to 999
Data columns (total 4 columns):
N    1000  non-null values
a    1000  non-null values
b    1000  non-null values
x    1000  non-null values
dtypes: float64(2), int64(1), object(1)
```

Here's the function in pure python:

```
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:

In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...:
```

We achieve our result by by using `apply` (row-wise):

```
In [5]: %timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
1 loops, best of 3: 219 ms per loop
```

But clearly this isn't fast enough for us. Let's take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the prun ipython magic function:

```
In [6]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
         573595 function calls in 0.334 seconds
   Ordered by: internal time
   List reduced from 79 to 4 due to restriction <4>
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.195    0.000    0.307    0.000 <ipython-input-4-a877a66f40a5>:1(integrate_f)
   552423    0.107    0.000    0.107    0.000 <ipython-input-3-0b27f90c4c8a>:1(f)
     1000    0.005    0.000    0.005    0.000 {range}
     1000    0.003    0.000    0.009    0.000 series.py:501(from_array)
```

By far the majority of time is spend inside either `integrate_f` or `f`, hence we'll concentrate our efforts cythonizing these two functions.

Note: In python 2 replacing the `range` with its generator counterpart (`xrange`) would mean the `range` line would vanish. In python 3 range is already a generator.

19.1.2 Plain cython

First we're going to need to import the cython magic function to ipython:

```
In [7]: %load_ext cythonmagic
```

Now, let's simply copy our functions over to cython as is (the suffix is here to distinguish between function versions):

```
In [8]: %%cython
   ...: def f_plain(x):
   ...:     return x * (x - 1)
   ...: def integrate_f_plain(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_plain(a + i * dx)
   ...:     return s * dx
   ...:
```

Note: If you're having trouble pasting the above into your ipython, you may need to be using bleeding edge ipython for paste to play well with cell magics.

```
In [9]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 117 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

19.1.3 Adding type

We get another huge improvement simply by providing type information:

```
In [10]: %%cython
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....:

In [11]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 19.2 ms per loop
```

Now, we're talking! It's now over ten times faster than the original python implementation, and we haven't *really* modified the code. Let's have another look at what's eating up time:

```
In [12]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
         20172 function calls in 0.027 seconds
   Ordered by: internal time
   List reduced from 77 to 4 due to restriction <4>
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.003    0.000    0.009    0.000 series.py:501(from_array)
     1001    0.002    0.000    0.011    0.000 frame.py:4457(<genexpr>)
        1    0.002    0.002    0.027    0.027 frame.py:4433(_apply_standard)
     3000    0.002    0.000    0.004    0.000 index.py:718(get_value)
```

19.1.4 Using ndarray

It's calling series... a lot! It's creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in python, so maybe we could minimise these by cythonizing the apply part.

Note: We are now passing ndarrays into the cython function, fortunately cython plays very nicely with numpy.

```
In [13]: %%cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_b, np.ndarray co
   ....:     assert (col_a.dtype == np.float and col_b.dtype == np.float and col_N.dtype == np.int)
   ....:     cdef Py_ssize_t i, n = len(col_N)
   ....:     assert (len(col_a) == len(col_b) == n)
   ....:     cdef np.ndarray[double] res = np.empty(n)
   ....:     for i in range(len(col_a)):
```

```
....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
....:     return res
....:
```

The implementation is simple, it creates an array of zeros and loops over the rows, applying our `integrate_f_typed`, and putting this in the zeros array.

Note: Loop like this would be *extremely* slow in python, but in cython looping over numpy arrays is *fast*.

```
In [14]: %timeit apply_integrate_f(df['a'], df['b'], df['N'])
100 loops, best of 3: 6.69 ms per loop
```

We've gone another three times faster! Let's check again where the time is spent:

```
In [15]: %prun -l 4 apply_integrate_f(df['a'], df['b'], df['N'])
         9036 function calls in 0.010 seconds
   Ordered by: internal time
   List reduced from 13 to 4 due to restriction <4>
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.004    0.004    0.010    0.010 {_cython_magic_e90d139b668b225d575329a3043b5155.apply_
     3000    0.002    0.000    0.004    0.000 index.py:718(get_value)
     3000    0.002    0.000    0.006    0.000 series.py:616(__getitem__)
     3000    0.001    0.000    0.001    0.000 {method 'get_value' of 'pandas.index.IndexEngine' objec
```

As one might expect, the majority of the time is now spent in `apply_integrate_f`, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

19.1.5 More advanced techniques

There is still scope for improvement, here's an example of using some more advanced cython techniques:

```
In [16]: %%cython
....: cimport cython
....: cimport numpy as np
....: import numpy as np
....: cdef double f_typed(double x) except? -2:
....:     return x * (x - 1)
....: cpdef double integrate_f_typed(double a, double b, int N):
....:     cdef int i
....:     cdef double s, dx
....:     s = 0
....:     dx = (b - a) / N
....:     for i in range(N):
....:         s += f_typed(a + i * dx)
....:     return s * dx
....: @cython.boundscheck(False)
....: @cython.wraparound(False)
....: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a, np.ndarray[double]
....:     cdef Py_ssize_t i, n = len(col_N)
....:     assert len(col_a) == len(col_b) == n
....:     cdef np.ndarray[double] res = np.empty(n)
....:     for i in range(n):
....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
....:     return res
....:
```

```
In [17]: %timeit apply_integrate_f_wrap(df['a'], df['b'], df['N'])
100 loops, best of 3: 2.14 ms per loop
```

This shaves another third off!

19.1.6 Further topics

- Loading C modules into cython.

Read more in the cython docs.

CHAPTER

TWENTY

SPARSE DATA STRUCTURES

We have implemented "sparse" versions of Series, DataFrame, and Panel. These are not sparse in the typical "mostly 0". You can view these objects as being "compressed" where any data matching a specific value (NaN/missing by default, though any value can be chosen) is omitted. A special `SparseIndex` object tracks where data has been "sparsified". This will make much more sense in an example. All of the standard pandas data structures have a `to_sparse` method:

```
In [1]: ts = Series(randn(10))

In [2]: ts[2:-2] = np.nan

In [3]: sts = ts.to_sparse()

In [4]: sts

0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The `to_sparse` method takes a `kind` argument (for the sparse index, see below) and a `fill_value`. So if we had a mostly zero Series, we could convert it to sparse with `fill_value=0`:

```
In [5]: ts.fillna(0).to_sparse(fill_value=0)

0    0.469112
1   -0.282863
2    0.000000
3    0.000000
4    0.000000
5    0.000000
6    0.000000
7    0.000000
8   -0.861849
9   -2.104569
```

```
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA DataFrame:

```
In [6]: df = DataFrame(randn(10000, 4))

In [7]: df.ix[:9998] = np.nan

In [8]: sdf = df.to_sparse()

In [9]: sdf

<class 'pandas.sparse.frame.SparseDataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns (total 4 columns):
0    1  non-null values
1    1  non-null values
2    1  non-null values
3    1  non-null values
dtypes: float64(4)

In [10]: sdf.density
0.0001
```

As you can see, the density (% of values that have not been "compressed") is extremely low. This sparse object takes up much less memory on disk (pickled) and in the Python interpreter. Functionally, their behavior should be nearly identical to their dense counterparts.

Any sparse object can be converted back to the standard dense form by calling `to_dense`:

```
In [11]: sts.to_dense()

0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
dtype: float64
```

20.1 SparseArray

`SparseArray` is the base layer for all of the sparse indexed data structures. It is a 1-dimensional ndarray-like object storing only values distinct from the `fill_value`:

```
In [12]: arr = np.random.randn(10)

In [13]: arr[2:5] = np.nan; arr[7:8] = np.nan

In [14]: sparr = SparseArray(arr)
```

```
In [15]: sparr

[-1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.606027190513, 1.
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

Like the indexed objects (SparseSeries, SparseDataFrame, SparsePanel), a `SparseArray` can be converted back to a regular ndarray by calling `to_dense`:

```
In [16]: sparr.to_dense()

array([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
           nan,  0.606 ,  1.3342])
```

20.2 SparseList

`SparseList` is a list-like data structure for managing a dynamic collection of SparseArrays. To create one, simply call the `SparseList` constructor with a `fill_value` (defaulting to `NaN`):

```
In [17]: spl = SparseList()

In [18]: spl

<pandas.sparse.list.SparseList object at 0x124edf10>
```

The two important methods are `append` and `to_array`. `append` can accept scalar values or any 1-dimensional sequence:

```
In [19]: spl.append(np.array([1., nan, nan, 2., 3.]))

In [20]: spl.append(5)

In [21]: spl.append(sparr)

In [22]: spl

<pandas.sparse.list.SparseList object at 0x124edf10>
[1.0, nan, nan, 2.0, 3.0]
IntIndex
Indices: array([0, 3, 4], dtype=int32)
[5.0]
IntIndex
Indices: array([0], dtype=int32)
[-1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.606027190513, 1.
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

As you can see, all of the contents are stored internally as a list of memory-efficient `SparseArray` objects. Once you've accumulated all of the data, you can call `to_array` to get a single `SparseArray` with all the data:

```
In [23]: spl.to_array()

[1.0, nan, nan, 2.0, 3.0, 5.0, -1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.1452971
IntIndex
Indices: array([ 0,  3,  4,  5,  6,  7, 11, 12, 14, 15], dtype=int32)
```

20.3 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

TWENTYONE

CAVEATS AND GOTCHAS

21.1 NaN, Integer NA values and NA type promotions

21.1.1 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.

21.1.2 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 [1]: s = Series([1, 2, 3, 4, 5], index=list('abcde'))

In [2]: s

a    1
b    2
c    3
d    4
e    5
dtype: int64

In [3]: s.dtype
dtype('int64')

In [4]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [5]: s2

a     1
```

```
b     2
c     3
f   NaN
u   NaN
dtype: float64

In [6]: s2.dtype
dtype('float64')
```

This trade-off is made largely for memory and performance reasons, and also so that the resulting Series continues to be "numeric". One possibility is to use `dtype=object` arrays instead.

21.1.3 `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.

21.1.4 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.

21.2 Integer indexing

Label-based indexing with integer axis labels is a thorny topic. It has been discussed heavily on mailing lists and among various members of the scientific Python community. In pandas, our general viewpoint is that labels matter more than integer locations. Therefore, with an integer axis index *only* label-based indexing is possible with the standard tools like `.ix`. The following code will generate exceptions:

```
s = Series(range(5))
s[-1]
df = DataFrame(np.random.randn(5, 4))
df
df.ix[-2:]
```

This deliberate decision was made to prevent ambiguities and subtle bugs (many users reported finding bugs when the API change was made to stop "falling back" on position-based indexing).

21.3 Label-based slicing conventions

21.3.1 Non-monotonic indexes require exact matches

21.3.2 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 determine the "successor" or next element after a particular label in an index. For example, consider the following Series:

```
In [7]: s = Series(randn(6), index=list('abcdef'))

In [8]: s

a    1.337122
b   -1.531095
c    1.331458
d   -0.571329
e   -0.026671
f   -1.085663
dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be

```
In [9]: s[2:5]

c    1.331458
d   -0.571329
e   -0.026671
dtype: float64
```

However, if you only had `c` and `e`, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.ix['c':'e'+1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design design to make label-based slicing include both endpoints:

```
In [10]: s.ix['c':'e']

c    1.331458
d   -0.571329
e   -0.026671
dtype: float64
```

This is most definitely a "practicality beats purity" sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

21.4 Miscellaneous indexing gotchas

21.4.1 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 [11]: df = DataFrame(randn(6, 4), columns=['one', 'two', 'three', 'four'],
   ....:                index=list('abcdef'))
   ....:

In [12]: df

        one       two     three      four
a -1.114738 -0.058216 -0.486768  1.685148
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
d  0.184735 -0.255069 -0.271020  1.288393
e  0.294633 -1.165787  0.846974 -0.685597
f  0.609099 -0.303961  0.625555 -0.059268

In [13]: df.ix[['b', 'c', 'e']]

        one       two     three      four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
```

This is, of course, completely equivalent *in this case* to using th `reindex` method:

```
In [14]: df.reindex(['b', 'c', 'e'])

        one       two     three      four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
```

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 [15]: df.ix[[1, 2, 4]]

        one       two     three      four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
```

If you pass `[1, 2, 4]` to `reindex` you will get another thing entirely:

```
In [16]: df.reindex([1, 2, 4])

   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 [17]: s = Series([1, 2, 3], index=['a', 0, 1])

In [18]: s

a    1
0    2
1    3
dtype: int64

In [19]: s.ix[[0, 1]]

0    2
1    3
dtype: int64

In [20]: s.reindex([0, 1])

0    2
1    3
dtype: int64
```

Because the index in this case does not contain solely integers, `ix` falls back on integer indexing. By contrast, `reindex` only looks for the values passed in the index, thus finding the integers `0` and `1`. While it would be possible to insert some logic to check whether a passed sequence is all contained in the index, that logic would exact a very high cost in large data sets.

21.4.2 Reindex potentially changes underlying Series dtype

The use of `reindex_like` can potentially change the dtype of a `Series`.

```
series = pandas.Series([1, 2, 3])
x = pandas.Series([True])
x.dtype
x = pandas.Series([True]).reindex_like(series)
x.dtype
```

This is because `reindex_like` silently inserts `NaNs` and the `dtype` changes accordingly. This can cause some issues when using `numpy ufuncs` such as `numpy.logical_and`.

See the this old issue for a more detailed discussion.

21.5 Timestamp limitations

21.5.1 Minimum and maximum timestamps

Since pandas represents timestamps in nanosecond resolution, the timespan that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [21]: begin = Timestamp.min

In [22]: begin
Timestamp('1677-09-22 00:12:43.145225', tz=None)

In [23]: end = Timestamp.max

In [24]: end
Timestamp('2262-04-11 23:47:16.854775807', tz=None)
```

If you need to represent time series data outside the nanosecond timespan, use PeriodIndex:

```
In [25]: span = period_range('1215-01-01', '1381-01-01', freq='D')

In [26]: span

<class 'pandas.tseries.period.PeriodIndex'>
freq: D
[1215-01-01, ..., 1381-01-01]
length: 60632
```

21.6 Parsing Dates from Text Files

When parsing multiple text file columns into a single date column, the new date column is prepended to the data and then *index_col* specification is indexed off of the new set of columns rather than the original ones:

```
In [27]: print open('tmp.csv').read()
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

In [28]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [29]: df = read_csv('tmp.csv', header=None,
   ....:               parse_dates=date_spec,
   ....:               keep_date_col=True,
   ....:               index_col=0)
   ....:

# index_col=0 refers to the combined column "nominal" and not the original
# first column of 'KORD' strings
In [30]: df

                                actual     0         1          2          3  \
nominal
```

```
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00   18:56:00
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00   19:56:00
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00   20:56:00
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00   21:18:00
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00   21:56:00
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00   22:56:00
                       4
nominal
1999-01-27 19:00:00  0.81
1999-01-27 20:00:00  0.01
1999-01-27 21:00:00 -0.59
1999-01-27 21:00:00 -0.99
1999-01-27 22:00:00 -0.59
1999-01-27 23:00:00 -0.59
```

21.7 Differences with NumPy

For Series and DataFrame objects, `var` normalizes by `N-1` to produce unbiased estimates of the sample variance, while NumPy's `var` normalizes by N, which measures the variance of the sample. Note that `cov` normalizes by `N-1` in both pandas and NumPy.

21.8 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the `DataFrame.copy` method. If you are doing a lot of copying of DataFrame objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See this link for more information.

21.9 HTML Table Parsing

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function `read_html`.

Issues with lxml

- Benefits
  - **lxml** is very fast
  - **lxml** requires Cython to install correctly.
- Drawbacks
  - **lxml** does *not* make any guarantees about the results of it's parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the **lxml** backend, but **this backend will use html5lib** if **lxml** fails to parse
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if **lxml** fails.

Issues with BeautifulSoup4 using lxml as a backend

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

Issues with BeautifulSoup4 using html5lib as a backend

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is "correct", since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.
- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the url over the web, i.e., IO (input-output). For very large tables, this might not be true.

Issues with using Anaconda

- Anaconda ships with lxml version 3.2.0; the following workaround for Anaconda was successfully used to deal with the versioning issues surrounding lxml and BeautifulSoup4.

Note: Unless you have *both*:

- A strong restriction on the upper bound of the runtime of some code that incorporates `read_html()`
- Complete knowledge that the HTML you will be parsing will be 100% valid at all times

then you should install html5lib and things will work swimmingly without you having to muck around with *conda*. If you want the best of both worlds then install both html5lib and lxml. If you do install lxml then you need to perform the following commands to ensure that lxml will work correctly:

```
# remove the included version
conda remove lxml

# install the latest version of lxml
pip install 'git+git://github.com/lxml/lxml.git'

# install the latest version of beautifulsoup4
pip install 'bzr+lp:beautifulsoup'
```

Note that you need bzr and git installed to perform the last two operations.

21.10 Byte-Ordering Issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. To deal with this issue you should convert the underlying NumPy array to the native system byte order *before* passing it to Series/DataFrame/Panel constructors using something similar to the following:

```
In [31]: x = np.array(range(10), '>i4') # big endian

In [32]: newx = x.byteswap().newbyteorder() # force native byteorder

In [33]: s = Series(newx)
```

See the NumPy documentation on byte order for more details.

CHAPTER

TWENTYTWO

RPY2 / R INTERFACE

Note: This is all highly experimental. I would like to get more people involved with building a nice RPy2 interface for pandas

If your computer has R and rpy2 (> 2.2) installed (which will be left to the reader), you will be able to leverage the below functionality. On Windows, doing this is quite an ordeal at the moment, but users on Unix-like systems should find it quite easy. rpy2 evolves in time, and is currently reaching its release 2.3, while the current interface is designed for the 2.2.x series. We recommend to use 2.2.x over other series unless you are prepared to fix parts of the code, yet the rpy2-2.3.0 introduces improvements such as a better R-Python bridge memory management layer so I might be a good idea to bite the bullet and submit patches for the few minor differences that need to be fixed.

```
# if installing for the first time
hg clone http://bitbucket.org/lgautier/rpy2

cd rpy2
hg pull
hg update version_2.2.x
sudo python setup.py install
```

Note: To use R packages with this interface, you will need to install them inside R yourself. At the moment it cannot install them for you.

Once you have done installed R and rpy2, you should be able to import `pandas.rpy.common` without a hitch.

22.1 Transferring R data sets into Python

The **load_data** function retrieves an R data set and converts it to the appropriate pandas object (most likely a DataFrame):

```
In [1]: import pandas.rpy.common as com

In [2]: infert = com.load_data('infert')

In [3]: infert.head()
```

```
  education  age  parity  induced  case  spontaneous  stratum  pooled.stratum
1    0-5yrs   26       6        1     1            2        1               3
2    0-5yrs   42       1        1     1            0        2               1
3    0-5yrs   39       6        2     1            0        3               4
```

```
4    0-5yrs   34       4        2     1            0        4               2
5   6-11yrs   35       3        1     1            1        5              32
```

22.2 Converting DataFrames into R objects

New in version 0.8. Starting from pandas 0.8, there is **experimental** support to convert DataFrames into the equivalent R object (that is, **data.frame**):

```
In [4]: from pandas import DataFrame

In [5]: df = DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C':[7,8,9]},
   ...:                index=["one", "two", "three"])
   ...:

In [6]: r_dataframe = com.convert_to_r_dataframe(df)

In [7]: print type(r_dataframe)
<class 'rpy2.robjects.vectors.DataFrame'>

In [8]: print r_dataframe
      A B C
one   1 4 7
two   2 5 8
three 3 6 9
```

The DataFrame's index is stored as the `rownames` attribute of the data.frame instance.

You can also use **convert_to_r_matrix** to obtain a `Matrix` instance, but bear in mind that it will only work with homogeneously-typed DataFrames (as R matrices bear no information on the data type):

```
In [9]: r_matrix = com.convert_to_r_matrix(df)

In [10]: print type(r_matrix)
<class 'rpy2.robjects.vectors.Matrix'>

In [11]: print r_matrix
      A B C
one   1 4 7
two   2 5 8
three 3 6 9
```

22.3 Calling R functions with pandas objects

22.4 High-level interface to R estimators

CHAPTER

TWENTYTHREE

RELATED PYTHON LIBRARIES

23.1 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

23.2 statsmodels

The main statistics and econometrics library for Python. pandas has become a dependency of this library.

23.3 scikits.timeseries

scikits.timeseries provides a data structure for fixed frequency time series data based on the numpy.MaskedArray class. For time series data, it provides some of the same functionality to the pandas Series class. It has many more functions for time series-specific manipulation. Also, it has support for many more frequencies, though less customizable by the user (so 5-minutely data is easier to do with pandas for example).

We are aiming to merge these libraries together in the near future.

Progress:

- It has a collection of moving window statistics implemented in Bottleneck
- Outstanding issues

Summarising, Pandas offers superior functionality due to its combination with the `pandas.DataFrame`.

An introduction for former users of `scikits.timeseries` is provided in the *migration guide*.

CHAPTER

TWENTYFOUR

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.

24.1 data.frame

24.2 zoo

24.3 xts

24.4 plyr

24.5 reshape / reshape2

CHAPTER

TWENTYFIVE

API REFERENCE

25.1 Input/Output

25.1.1 Pickling

read_pickle(path)	Load pickled pandas object (or any other pickled object) from the specified

pandas.io.pickle.read_pickle

pandas.io.pickle.**read_pickle**(*path*)

Load pickled pandas object (or any other pickled object) from the specified file path

Warning: Loading pickled data received from untrusted sources can be unsafe. See: http://docs.python.org/2.7/library/pickle.html

Parameters **path** : string

File path

Returns **unpickled** : type of object stored in file

25.1.2 Flat File

read_table(filepath_or_buffer[, sep, ...])	Read general delimited file into DataFrame
read_csv(filepath_or_buffer[, sep, dialect, ...])	Read CSV (comma-separated) file into DataFrame
read_fwf(filepath_or_buffer[, colspecs, widths])	Read a table of fixed-width formatted lines into DataFrame
read_clipboard(**kwargs)	

pandas.io.parsers.read_table

pandas.io.parsers.**read_table**(*filepath_or_buffer*, *sep='\t'*, *dialect=None*, *compression=None*, *doublequote=True*, *escapechar=None*, *quotechar='"'*, *quoting=0*, *skipinitialspace=False*, *lineterminator=None*, *header='infer'*, *index_col=None*, *names=None*, *prefix=None*, *skiprows=None*, *skipfooter=None*, *skip_footer=0*, *na_values=None*, *na_fvalues=None*, *true_values=None*, *false_values=None*, *delimiter=None*, *converters=None*, *dtype=None*, *usecols=None*, *engine='c'*, *delim_whitespace=False*, *as_recarray=False*, *na_filter=True*, *compact_ints=False*, *use_unsigned=False*, *low_memory=True*, *buffer_lines=None*, *warn_bad_lines=True*, *error_bad_lines=True*, *keep_default_na=True*, *thousands=None*, *comment=None*, *decimal='.'*, *parse_dates=False*, *keep_date_col=False*, *dayfirst=False*, *date_parser=None*, *memory_map=False*, *nrows=None*, *iterator=False*, *chunksize=None*, *verbose=False*, *encoding=None*, *squeeze=False*, *mangle_dupe_cols=True*, *tupleize_cols=True*)

Read general delimited file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Parameters **filepath_or_buffer** : string or file handle / StringIO. The string could be

a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.csv

sep : string, default t (tab-stop)

Delimiter to use. Regular expressions are accepted.

lineterminator : string (length 1), default None

Character to break file into lines. Only valid with C parser

quotechar : string

The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting : int

Controls whether quotes should be recognized. Values are taken from *csv.QUOTE_** values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.

skipinitialspace : boolean, default False

Skip spaces after delimiter

escapechar : string

dtype : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None

For on-the-fly decompression of on-disk data

dialect : string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

header : int, default 0 if names parameter not specified,

Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row. Can be a list of integers that specify row locations for a multi-index on the columns E.g. [0,1,3]. Interveaning rows that are not specified (E.g. 2 in this example are skipped)

skiprows : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

index_col : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string or None (default)

Prefix to add to column numbers when no header, e.g 'X' for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

If True -> try parsing the index. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : function

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion.

dayfirst : boolean, default False

DD/MM format dates, international and European format

thousands : str, default None

Thousands separator

comment : str, default None

Indicates remainder of line should not be parsed Does not support line commenting (will return empty line)

decimal : str, default '.'

Character to recognize as decimal point. E.g. use ',' for European data

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

iterator : boolean, default False

Return TextFileReader object

chunksize : int, default None

Return TextFileReader object for iteration

skipfooter : int, default 0

Number of line at bottom of file to skip

converters : dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

delimiter : string, default None

Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

na_filter: boolean, default True :

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols: boolean, default True :

Duplicate columns will be specified as 'X.0'...'X.N', rather than 'X'...'X'

tupleize_cols: boolean, default False :

Leave a list of tuples on columns as is (default is to convert to a Multi Index on the columns)

Returns **result** : DataFrame or TextParser

pandas.io.parsers.read_csv

pandas.io.parsers.**read_csv**(*filepath_or_buffer*, *sep=','*, *dialect=None*, *compression=None*, *doublequote=True*, *escapechar=None*, *quotechar='"'*, *quoting=0*, *skipinitialspace=False*, *lineterminator=None*, *header='infer'*, *index_col=None*, *names=None*, *prefix=None*, *skiprows=None*, *skipfooter=None*, *skip_footer=0*, *na_values=None*, *na_fvalues=None*, *true_values=None*, *false_values=None*, *delimiter=None*, *converters=None*, *dtype=None*, *usecols=None*, *engine='c'*, *delim_whitespace=False*, *as_recarray=False*, *na_filter=True*, *compact_ints=False*, *use_unsigned=False*, *low_memory=True*, *buffer_lines=None*, *warn_bad_lines=True*, *error_bad_lines=True*, *keep_default_na=True*, *thousands=None*, *comment=None*, *decimal='.'*, *parse_dates=False*, *keep_date_col=False*, *dayfirst=False*, *date_parser=None*, *memory_map=False*, *nrows=None*, *iterator=False*, *chunksize=None*, *verbose=False*, *encoding=None*, *squeeze=False*, *mangle_dupe_cols=True*, *tupleize_cols=True*)

Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Parameters **filepath_or_buffer** : string or file handle / StringIO. The string could be

a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.csv

sep : string, default ','

Delimiter to use. If sep is None, will try to automatically determine this. Regular expressions are accepted.

lineterminator : string (length 1), default None

Character to break file into lines. Only valid with C parser

quotechar : string

The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting : int

Controls whether quotes should be recognized. Values are taken from *csv.QUOTE_** values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.

skipinitialspace : boolean, default False

Skip spaces after delimiter

escapechar : string

dtype : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None

For on-the-fly decompression of on-disk data

dialect : string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

header : int, default 0 if names parameter not specified,

Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row. Can be a list of integers that specify row locations for a multi-index on the columns E.g. [0,1,3]. Interveaning rows that are not specified (E.g. 2 in this example are skipped)

skiprows : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

index_col : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string or None (default)

Prefix to add to column numbers when no header, e.g 'X' for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

If True -> try parsing the index. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : function

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion.

dayfirst : boolean, default False

DD/MM format dates, international and European format

thousands : str, default None

Thousands separator

comment : str, default None

Indicates remainder of line should not be parsed Does not support line commenting (will return empty line)

decimal : str, default '.'

Character to recognize as decimal point. E.g. use ',' for European data

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

iterator : boolean, default False

Return TextFileReader object

chunksize : int, default None

Return TextFileReader object for iteration

skipfooter : int, default 0

Number of line at bottom of file to skip

converters : dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

delimiter : string, default None

Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

na_filter: boolean, default True :

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols: boolean, default True :

Duplicate columns will be specified as 'X.0'...'X.N', rather than 'X'...'X'

tupleize_cols: boolean, default False :

Leave a list of tuples on columns as is (default is to convert to a Multi Index on the columns)

Returns **result** : DataFrame or TextParser

pandas.io.parsers.read_fwf

pandas.io.parsers.**read_fwf**(*filepath_or_buffer*, *colspecs=None*, *widths=None*, ***kwds*)

Read a table of fixed-width formatted lines into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Parameters **filepath_or_buffer** : string or file handle / StringIO. The string could be

a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.csv

colspecs : a list of pairs (tuples), giving the extents

of the fixed-width fields of each line as half-open internals (i.e., [from, to[ ).

widths : a list of field widths, which can be used instead of

'colspecs' if the intervals are contiguous.

lineterminator : string (length 1), default None

Character to break file into lines. Only valid with C parser

quotechar : string

The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting : int

Controls whether quotes should be recognized. Values are taken from *csv.QUOTE_** values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.

skipinitialspace : boolean, default False

Skip spaces after delimiter

escapechar : string

dtype : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None

For on-the-fly decompression of on-disk data

dialect : string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

header : int, default 0 if names parameter not specified,

Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row. Can be a list of integers that specify row locations for a multi-index on the columns E.g. [0,1,3]. Interveaning rows that are not specified (E.g. 2 in this example are skipped)

skiprows : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

index_col : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string or None (default)

Prefix to add to column numbers when no header, e.g 'X' for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

If True -> try parsing the index. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : function

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion.

dayfirst : boolean, default False

DD/MM format dates, international and European format

thousands : str, default None

Thousands separator

comment : str, default None

Indicates remainder of line should not be parsed Does not support line commenting (will return empty line)

decimal : str, default '.'

Character to recognize as decimal point. E.g. use ',' for European data

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

iterator : boolean, default False

Return TextFileReader object

chunksize : int, default None

Return TextFileReader object for iteration

skipfooter : int, default 0

Number of line at bottom of file to skip

converters : dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

delimiter : string, default None

Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

na_filter: boolean, default True :

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols: boolean, default True :

Duplicate columns will be specified as 'X.0'...'X.N', rather than 'X'...'X'

tupleize_cols: boolean, default False :

Leave a list of tuples on columns as is (default is to convert to a Multi Index on the columns)

Returns **result** : DataFrame or TextParser

Also, 'delimiter' is used to specify the filler character of the :

fields if it is not spaces (e.g., '~'). :

pandas.io.parsers.read_clipboard

pandas.io.parsers.**read_clipboard**(***kwargs*)

25.1.3 Excel

read_excel(path_or_buf, sheetname[, kind])	Read an Excel table into a pandas DataFrame
ExcelFile.parse(sheetname[, header, ...])	Read an Excel table into DataFrame

pandas.io.excel.read_excel

pandas.io.excel.**read_excel**(*path_or_buf*, *sheetname*, *kind=None*, ***kwds*)

Read an Excel table into a pandas 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

Rows to skip at the beginning (0-indexed)

skip_footer : int, default 0

Rows at the end to skip (0-indexed)

index_col : int, default None

Column to use as the row labels of the DataFrame. Pass None if there is no such column

parse_cols : int or list, default None

- If None then parse all columns,
- If int then indicates last column to be parsed
- If list of ints then indicates list of column numbers to be parsed
- If string then indicates comma separated list of column names and column ranges (e.g. "A:E" or "A,C,E:F")

na_values : list-like, default None

List of additional strings to recognize as NA/NaN

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

Returns **parsed** : DataFrame

DataFrame from the passed in Excel file

pandas.io.excel.ExcelFile.parse

ExcelFile.**parse**(*sheetname*, *header=0*, *skiprows=None*, *skip_footer=0*, *index_col=None*, *parse_cols=None*, *parse_dates=False*, *date_parser=None*, *na_values=None*, *thousands=None*, *chunksize=None*, ***kwds*)

Read an 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

Rows to skip at the beginning (0-indexed)

skip_footer : int, default 0

Rows at the end to skip (0-indexed)

index_col : int, default None

Column to use as the row labels of the DataFrame. Pass None if there is no such column

parse_cols : int or list, default None

- If None then parse all columns
- If int then indicates last column to be parsed
- If list of ints then indicates list of column numbers to be parsed
- If string then indicates comma separated list of column names and column ranges (e.g. "A:E" or "A,C,E:F")

na_values : list-like, default None

List of additional strings to recognize as NA/NaN

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

Returns **parsed** : DataFrame

DataFrame parsed from the Excel file

25.1.4 JSON

read_json([path_or_buf, orient, typ, dtype, ...])	Convert JSON string to pandas object

pandas.io.json.read_json

pandas.io.json.**read_json**(*path_or_buf=None*, *orient=None*, *typ='frame'*, *dtype=True*, *convert_axes=True*, *convert_dates=True*, *keep_default_dates=True*, *numpy=False*, *precise_float=False*)

Convert JSON string to pandas object

Parameters **filepath_or_buffer** : a VALID JSON string or file handle / StringIO. The string could be

a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json

orient : :

Series : default is 'index' allowed values are: {'split','records','index'}

DataFrame : default is 'columns' allowed values are: {'split','records','index','columns','values'}

The format of the JSON string split : dict like {index -> [index], columns -> [columns], data -> [values]} records : list like [{column -> value}, ... , {column -> value}] index : dict like {index -> {column -> value}} columns : dict like {column -> {index -> value}} values : just the values array

typ : type of object to recover (series or frame), default 'frame'

dtype : if True, infer dtypes, if a dict of column to dtype, then use those,

if False, then don't infer dtypes at all, default is True, apply only to the data

convert_axes : boolean, try to convert the axes to the proper dtypes, default is True

convert_dates : a list of columns to parse for dates; If True, then try to parse datelike columns

default is True

keep_default_dates : boolean, default True. If parsing dates,

then parse the default datelike columns

numpy : direct decoding to numpy arrays. default is False.Note that the JSON ordering MUST be the same

for each term if numpy=True.

precise_float : boolean, default False. Set to enable usage of higher precision (strtod) function

when decoding string to double values. Default (False) is to use fast but less precise builtin functionality

Returns **result** : Series or DataFrame

25.1.5 HTML

read_html(io[, match, flavor, header, ...])	Read an HTML table into a DataFrame.

pandas.io.html.read_html

pandas.io.html.**read_html**(*io*, *match='.+'*, *flavor=None*, *header=None*, *index_col=None*, *skiprows=None*, *infer_types=True*, *attrs=None*)

Read an HTML table into a DataFrame.

Parameters **io** : str or file-like

A string or file like object that can be either a url, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URI that starts with `'https'` you might removing the `'s'`.

match : str or regex, optional, default '.+'

The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to '.+' (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

flavor : str, container of strings, default `None`

The parsing engine to use under the hood. 'bs4' and 'html5lib' are synonymous with each other, they are both there for backwards compatibility. The default of `None` tries to use `lxml` to parse and if that fails it falls back on `bs4` + `html5lib`.

header : int or array-like or None, optional, default `None`

The row (or rows for a MultiIndex) to use to make the columns headers. Note that this row will be removed from the data.

index_col : int or array-like or None, optional, default `None`

The column to use to make the index. Note that this column will be removed from the data.

skiprows : int or collections.Container or slice or None, optional, default `None`

If an integer is given then skip this many rows after parsing the column header. If a sequence of integers is given skip those specific rows (0-based). Note that

```
skiprows == 0
```

yields the same result as

```
skiprows is None
```

If *skiprows* is a positive integer, say n, then it is treated as "skip n rows", *not* as "skip the n^{th} row".

infer_types : bool, optional, default `True`

Whether to convert numeric types and date-appearing strings to numbers and dates, respectively.

attrs : dict or None, optional, default `None`

This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to lxml or Beautiful Soup. However, these attributes must be valid HTML table attributes to work correctly. For example,

```
attrs = {'id': 'table'}
```

is a valid attribute dictionary because the 'id' HTML tag attribute is a valid HTML attribute for *any* HTML tag as per this document.

```
attrs = {'asdf': 'table'}
```

is *not* a valid attribute dictionary because 'asdf' is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found here. A working draft of the HTML 5 spec can be found here. It contains the latest information on table attributes for the modern web.

Returns **dfs** : list of DataFrames

A list of DataFrames, each of which is the parsed data from each of the tables on the page.

Notes

Before using this function you should probably read the *gotchas about the parser libraries that this function uses*.

There's as little cleaning of the data as possible due to the heterogeneity and general disorder of HTML on the web.

Expect some cleanup after you call this function. For example, you might need to pass *infer_types=False* and perform manual conversion if the column names are converted to NaN when you pass the *header=0* argument. We try to assume as little as possible about the structure of the table and push the idiosyncrasies of the HTML contained in the table to you, the user.

This function only searches for <table> elements and only for <tr> and <th> rows and <td> elements within those rows. This could be extended by subclassing one of the parser classes contained in `pandas.io.html`.

Similar to `read_csv()` the *header* argument is applied **after** *skiprows* is applied.

This function will *always* return a list of `DataFrame` *or* it will fail, e.g., it will *not* return an empty list.

Examples

See the *read_html documentation in the IO section of the docs* for many examples of reading HTML.

25.1.6 HDFStore: PyTables (HDF5)

`read_hdf`(path_or_buf, key, **kwargs)	read from the store, closeit if we opened it
`HDFStore.put`(key, value[, table, append])	Store object in HDFStore
`HDFStore.append`(key, value[, columns])	Append to Table in file. Node must already exist and be Table
`HDFStore.get`(key)	Retrieve pandas object stored in file
`HDFStore.select`(key[, where, start, stop, ...])	Retrieve pandas object stored in file, optionally based on where

pandas.io.pytables.read_hdf

`pandas.io.pytables.`**`read_hdf`**(*path_or_buf*, *key*, ***kwargs*)

read from the store, closeit if we opened it

pandas.io.pytables.HDFStore.put

`HDFStore.`**`put`**(*key*, *value*, *table=None*, *append=False*, ***kwargs*)

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

encoding : default None, provide an encoding for strings

pandas.io.pytables.HDFStore.append

HDFStore.**append**(*key*, *value*, *columns=None*, ***kwargs*)

Append to Table in file. Node must already exist and be Table format.

Parameters **key** : object

value : {Series, DataFrame, Panel, Panel4D}

data_columns : list of columns to create as data columns, or True to use all columns

min_itemsize : dict of columns that specify minimum string sizes

nan_rep : string to use as string nan represenation

chunksize : size to chunk the writing

expectedrows : expected TOTAL row size of this table

encoding : default None, provide an encoding for strings

Notes

Does *not* check if data being appended overlaps with existing data in the table, so be careful

pandas.io.pytables.HDFStore.get

HDFStore.**get**(*key*)

Retrieve pandas object stored in file

Parameters **key** : object

Returns **obj** : type of object stored in file

pandas.io.pytables.HDFStore.select

HDFStore.**select**(*key*, *where=None*, *start=None*, *stop=None*, *columns=None*, *iterator=False*, *chunksize=None*, *auto_close=False*, ***kwargs*)

Retrieve pandas object stored in file, optionally based on where criteria

Parameters **key** : object

where : list of Term (or convertable) objects, optional

start : integer (defaults to None), row number to start selection

stop : integer (defaults to None), row number to stop selection

columns : a list of columns that if not None, will limit the return columns

iterator : boolean, return an iterator, default False

chunksize : nrows to include in iteration, return an iterator

auto_close : boolean, should automatically close the store when finished, default is False

Continued on next page

Table 25.7 – continued from previous page

25.1.7 SQL

read_sql(sql, con[, index_col, ...])	Returns a DataFrame corresponding to the result set of the query
read_frame(sql, con[, index_col, ...])	Returns a DataFrame corresponding to the result set of the query
write_frame(frame, name, con[, flavor, ...])	Write records stored in a DataFrame to a SQL database.

pandas.io.sql.read_sql

pandas.io.sql.**read_sql**(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*)

Returns a DataFrame corresponding to the result set of the query string.

Optionally provide an index_col parameter to use one of the columns as the index. Otherwise will be 0 to len(results) - 1.

Parameters **sql: string** :

SQL query to be executed

con: DB connection object, optional :

index_col: string, optional :

column name to use for the returned DataFrame object.

coerce_float : boolean, default True

Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

params: list or tuple, optional :

List of parameters to pass to execute method.

pandas.io.sql.read_frame

pandas.io.sql.**read_frame**(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*)

Returns a DataFrame corresponding to the result set of the query string.

Optionally provide an index_col parameter to use one of the columns as the index. Otherwise will be 0 to len(results) - 1.

Parameters **sql: string** :

SQL query to be executed

con: DB connection object, optional :

index_col: string, optional :

column name to use for the returned DataFrame object.

coerce_float : boolean, default True

Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

params: list or tuple, optional :

List of parameters to pass to execute method.

pandas.io.sql.write_frame

pandas.io.sql.**write_frame**(*frame*, *name*, *con*, *flavor='sqlite'*, *if_exists='fail'*, ***kwargs*)

Write records stored in a DataFrame to a SQL database.

Parameters **frame: DataFrame** :

name: name of SQL table :

con: an open SQL database connection object :

flavor: {'sqlite', 'mysql', 'oracle'}, default 'sqlite' :

if_exists: {'fail', 'replace', 'append'}, default 'fail' :

fail: If table exists, do nothing. replace: If table exists, drop it, recreate it, and insert data. append: If table exists, insert data. Create if does not exist.

25.1.8 STATA

read_stata(filepath_or_buffer[, ...])	Read Stata file into DataFrame
StataReader.data([convert_dates, ...])	Reads observations from Stata file, converting them into a dataframe
StataReader.data_label()	Returns data label of Stata file
StataReader.value_labels()	Returns a dict, associating each variable name a dict, associating each value its corresp
StataReader.variable_labels()	Returns variable labels as a dict, associating each variable name with corresponding lab
StataWriter.write_file()	

pandas.io.stata.read_stata

pandas.io.stata.**read_stata**(*filepath_or_buffer*, *convert_dates=True*, *convert_categoricals=True*, *encoding=None*, *index=None*)

Read Stata file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Parameters **filepath_or_buffer** : string or file handle / StringIO. The string could be

a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.csv

%s :

lineterminator : string (length 1), default None

Character to break file into lines. Only valid with C parser

quotechar : string

The character to used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting : int

Controls whether quotes should be recognized. Values are taken from *csv.QUOTE_** values. Acceptable values are 0, 1, 2, and 3 for QUOTE_MINIMAL, QUOTE_ALL, QUOTE_NONE, and QUOTE_NONNUMERIC, respectively.

skipinitialspace : boolean, default False

Skip spaces after delimiter

escapechar : string

dtype : Type name or dict of column -> type

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None

For on-the-fly decompression of on-disk data

dialect : string or csv.Dialect instance, default None

If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect documentation for more details

header : int, default 0 if names parameter not specified,

Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row. Can be a list of integers that specify row locations for a multi-index on the columns E.g. [0,1,3]. Interveaning rows that are not specified (E.g. 2 in this example are skipped)

skiprows : list-like or integer

Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

index_col : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly pass header=None

prefix : string or None (default)

Prefix to add to column numbers when no header, e.g 'X' for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

If True -> try parsing the index. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : function

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion.

dayfirst : boolean, default False

DD/MM format dates, international and European format

thousands : str, default None

Thousands separator

comment : str, default None

Indicates remainder of line should not be parsed Does not support line commenting (will return empty line)

decimal : str, default '.'

Character to recognize as decimal point. E.g. use ',' for European data

nrows : int, default None

Number of rows of file to read. Useful for reading pieces of large files

iterator : boolean, default False

Return TextFileReader object

chunksize : int, default None

Return TextFileReader object for iteration

skipfooter : int, default 0

Number of line at bottom of file to skip

converters : dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

verbose : boolean, default False

Indicate number of NA values placed in non-numeric columns

delimiter : string, default None

Alternative argument name for sep. Regular expressions are accepted.

encoding : string, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8')

squeeze : boolean, default False

If the parsed data only contains one column then return a Series

na_filter: boolean, default True :

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file

usecols : array-like

Return a subset of the columns. Results in much faster parsing time and lower memory usage.

mangle_dupe_cols: boolean, default True :

Duplicate columns will be specified as 'X.0'...'X.N', rather than 'X'...'X'

tupleize_cols: boolean, default False :

Leave a list of tuples on columns as is (default is to convert to a Multi Index on the columns)

Returns **result** : DataFrame or TextParser

pandas.io.stata.StataReader.data

StataReader.**data**(*convert_dates=True*, *convert_categoricals=True*, *index=None*)

Reads observations from Stata file, converting them into a dataframe

Parameters **convert_dates** : boolean, defaults to True

Convert date variables to DataFrame time values

convert_categoricals : boolean, defaults to True

Read value labels and convert columns to Categorical/Factor variables

index : identifier of index column

identifier of column that should be used as index of the DataFrame

Returns **y** : DataFrame instance

pandas.io.stata.StataReader.data_label

StataReader.**data_label**()

Returns data label of Stata file

pandas.io.stata.StataReader.value_labels

StataReader.**value_labels**()

Returns a dict, associating each variable name a dict, associating each value its corresponding label

pandas.io.stata.StataReader.variable_labels

StataReader.**variable_labels**()

Returns variable labels as a dict, associating each variable name with corresponding label

pandas.io.stata.StataWriter.write_file

StataWriter.**write_file**()

25.2 General functions

25.2.1 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*, *dropna=True*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

Parameters **data** : DataFrame

values : column to aggregate, optional

rows : list of column names or arrays to group on

Keys to group on the x-axis of the pivot table

cols : list of column names or arrays to group on

Keys to group on the y-axis of the pivot table

aggfunc : function, default numpy.mean, or list of functions

If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves)

fill_value : scalar, default None

Value to replace missing values with

margins : boolean, default False

Add all row / columns (e.g. for subtotal / grand totals)

dropna : boolean, default True

Do not include columns whose entries are all NaN

Returns **table** : DataFrame

Examples

```
>>> df
   A   B   C      D
0  foo one small  1
1  foo one large  2
2  foo one large  2
3  foo two small  3
4  foo two small  3
5  bar one large  4
6  bar one small  5
7  bar two small  6
8  bar two large  7
```

```
>>> table = pivot_table(df, values='D', rows=['A', 'B'],
...                     cols=['C'], aggfunc=np.sum)
>>> table
          small  large
foo  one  1      4
     two  6      NaN
bar  one  5      4
     two  6      7
```

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=False*, *suffixes=('_x', '_y')*, *copy=True*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters **left** : DataFrame

right : DataFrame

how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

on : label or list

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

left_on : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

right_on : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left_on docs

left_index : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left_index

sort : boolean, default False

Sort the join keys lexicographically in the result DataFrame

suffixes : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

copy : boolean, default True

If False, do not copy data unnecessarily

Returns **merged** : DataFrame

Examples

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  bar   2        bar   6
1  bar   2        bar   8
2  baz   3        NaN   NaN
3  foo   1        foo   5
4  foo   4        foo   5
5  NaN   NaN      qux   7
```

pandas.tools.merge.concat

pandas.tools.merge.**concat**(*objs*, *axis=0*, *join='outer'*, *join_axes=None*, *ignore_index=False*, *keys=None*, *levels=None*, *names=None*, *verify_integrity=False*)

Concatenate pandas objects along a particular axis with optional set logic along the other axes. Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number

Parameters **objs** : list or dict of Series, DataFrame, or Panel objects

If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case an Exception will be raised

axis : {0, 1, ...}, default 0

The axis to concatenate along

join : {'inner', 'outer'}, default 'outer'

How to handle indexes on other axis(es)

join_axes : list of Index objects

Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic

verify_integrity : boolean, default False

Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation

keys : sequence, default None

If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level

levels : list of sequences, default None

Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys

names : list, default None

Names for the levels in the resulting hierarchical index

ignore_index : boolean, default False

If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the the index values on the other axes are still respected in the join.

Returns **concatenated** : type of objects

Notes

The keys, levels, and names arguments are all optional

25.2.2 Top-level missing data

isnull(obj)	Detect missing values (NaN in numeric arrays, None/NaN in object arrays)
notnull(obj)	Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

pandas.core.common.isnull

pandas.core.common.**isnull**(*obj*)

Detect missing values (NaN in numeric arrays, None/NaN in object arrays)

Parameters **arr** : ndarray or object value

Object to check for null-ness

Returns **isnulled** : array-like of bool or bool

Array or bool indicating whether an object is null or if an array is given which of the element is null.

pandas.core.common.notnull

pandas.core.common.**notnull**(*obj*)

Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

Parameters **arr** : ndarray or object value

Object to check for *not*-null-ness

Returns **isnulled** : array-like of bool or bool

Array or bool indicating whether an object is *not* null or if an array is given which of the element is *not* null.

25.2.3 Top-level dealing with datetimes

to_datetime(arg[, errors, dayfirst, utc, ...])	Convert argument to datetime

pandas.tseries.tools.to_datetime

pandas.tseries.tools.**to_datetime**(*arg*, *errors='ignore'*, *dayfirst=False*, *utc=None*, *box=True*, *format=None*, *coerce=False*, *unit='ns'*)

Convert argument to datetime

Parameters **arg** : string, datetime, array of strings (with possible NAs)

errors : {'ignore', 'raise'}, default 'ignore'

Errors are ignored by default (values left untouched)

dayfirst : boolean, default False

If True parses dates with the day first, eg 20/01/2005 Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug).

utc : boolean, default None

Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well)

box : boolean, default True

If True returns a DatetimeIndex, if False returns ndarray of values

format : string, default None

strftime to parse time, eg "%d/%m/%Y"

coerce : force errors to NaT (False by default)

unit : unit of the arg (D,s,ms,us,ns) denote the unit in epoch

(e.g. a unix timestamp), which is an integer/float number

Returns **ret** : datetime if parsing succeeded

25.2.4 Standard moving window functions

rolling_count(arg, window[, freq, center, ...])	Rolling count of number of non-NaN observations inside provided window.
rolling_sum(arg, window[, min_periods, ...])	Moving sum
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*, *freq=None*, *center=False*, *time_rule=None*)

Rolling count of number of non-NaN observations inside provided window.

Parameters **arg** : DataFrame or numpy ndarray-like

window : Number of observations used for calculating statistic

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **rolling_count** : type of caller

pandas.stats.moments.rolling_sum

pandas.stats.moments.**rolling_sum**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Moving sum

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_mean

pandas.stats.moments.**rolling_mean**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Moving mean

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_median

pandas.stats.moments.**rolling_median**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

O(N log(window)) implementation using skip list

Moving median

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_var

pandas.stats.moments.**rolling_var**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased moving variance

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_std

pandas.stats.moments.**rolling_std**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased moving standard deviation

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_corr

pandas.stats.moments.**rolling_corr**(*arg1*, *arg2*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*)

Moving sample correlation

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series -> Computes result for each column Series / Series -> Series

pandas.stats.moments.rolling_cov

pandas.stats.moments.**rolling_cov**(*arg1*, *arg2*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*)

Unbiased moving covariance

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series -> Computes result for each column Series / Series -> Series

pandas.stats.moments.rolling_skew

pandas.stats.moments.**rolling_skew**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased moving skewness

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_kurt

pandas.stats.moments.**rolling_kurt**(*arg*, *window*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased moving kurtosis

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_apply

pandas.stats.moments.**rolling_apply**(*arg*, *window*, *func*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*)

Generic moving function application

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

func : function

Must produce a single value from an ndarray input

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.rolling_quantile

pandas.stats.moments.**rolling_quantile**(*arg*, *window*, *quantile*, *min_periods=None*, *freq=None*, *center=False*, *time_rule=None*)

Moving quantile

Parameters **arg** : Series, DataFrame

window : Number of observations used for calculating statistic

quantile : 0 <= quantile <= 1

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **y** : type of input argument

25.2.5 Standard expanding window functions

expanding_count(arg[, freq, center, time_rule])	Expanding count of number of non-NaN observations.
expanding_sum(arg[, min_periods, freq, ...])	Expanding sum
expanding_mean(arg[, min_periods, freq, ...])	Expanding mean
expanding_median(arg[, min_periods, freq, ...])	O(N log(window)) implementation using skip list
expanding_var(arg[, min_periods, freq, ...])	Unbiased expanding variance
expanding_std(arg[, min_periods, freq, ...])	Unbiased expanding standard deviation
expanding_corr(arg1, arg2[, min_periods, ...])	Expanding sample correlation
expanding_cov(arg1, arg2[, min_periods, ...])	Unbiased expanding covariance
expanding_skew(arg[, min_periods, freq, ...])	Unbiased expanding skewness
expanding_kurt(arg[, min_periods, freq, ...])	Unbiased expanding kurtosis
expanding_apply(arg, func[, min_periods, ...])	Generic expanding function application
expanding_quantile(arg, quantile[, ...])	Expanding quantile

pandas.stats.moments.expanding_count

pandas.stats.moments.**expanding_count**(*arg*, *freq=None*, *center=False*, *time_rule=None*)

Expanding count of number of non-NaN observations.

Parameters **arg** : DataFrame or numpy ndarray-like

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **expanding_count** : type of caller

pandas.stats.moments.expanding_sum

pandas.stats.moments.**expanding_sum**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Expanding sum

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_mean

pandas.stats.moments.**expanding_mean**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Expanding mean

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_median

pandas.stats.moments.**expanding_median**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

O(N log(window)) implementation using skip list

Expanding median

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_var

pandas.stats.moments.**expanding_var**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased expanding variance

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_std

pandas.stats.moments.**expanding_std**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased expanding standard deviation

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_corr

pandas.stats.moments.**expanding_corr**(*arg1*, *arg2*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*)

Expanding sample correlation

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series -> Computes result for each column Series / Series -> Series

pandas.stats.moments.expanding_cov

pandas.stats.moments.**expanding_cov**(*arg1*, *arg2*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*)

Unbiased expanding covariance

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type depends on inputs

DataFrame / DataFrame -> DataFrame (matches on columns) DataFrame / Series -> Computes result for each column Series / Series -> Series

pandas.stats.moments.expanding_skew

pandas.stats.moments.**expanding_skew**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased expanding skewness

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_kurt

pandas.stats.moments.**expanding_kurt**(*arg*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*, ***kwargs*)

Unbiased expanding kurtosis

Parameters **arg** : Series, DataFrame

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

Returns **y** : type of input argument

pandas.stats.moments.expanding_apply

pandas.stats.moments.**expanding_apply**(*arg*, *func*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*)

Generic expanding function application

Parameters **arg** : Series, DataFrame

func : function

Must produce a single value from an ndarray input

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **y** : type of input argument

pandas.stats.moments.expanding_quantile

pandas.stats.moments.**expanding_quantile**(*arg*, *quantile*, *min_periods=1*, *freq=None*, *center=False*, *time_rule=None*)

Expanding quantile

Parameters **arg** : Series, DataFrame

quantile : 0 <= quantile <= 1

min_periods : int

Minimum number of observations in window required to have a value

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

center : boolean, default False

Whether the label should correspond with center of window

time_rule : Legacy alias for freq

Returns **y** : type of input argument

25.2.6 Exponentially-weighted moving window functions

ewma(arg[, com, span, min_periods, freq, ...])	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*, *freq=None*, *time_rule=None*, *adjust=True*)

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)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

adjust : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

Returns **y** : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a "span" parameter s, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

adjust : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

bias : boolean, default False

Use a standard estimation bias correction

Returns **y** : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a "span" parameter s, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s+1) = c/(1+c)$

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*, *freq=None*, *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)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

adjust : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

bias : boolean, default False

Use a standard estimation bias correction

Returns **y** : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a "span" parameter s, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s+1) = c/(1+c)$

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*, *freq=None*, *time_rule=None*)

Exponentially-weighted moving correlation

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

com : float. optional

Center of mass: alpha = 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)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

adjust : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

Returns **y** : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a "span" parameter s, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

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*, *freq=None*, *time_rule=None*)

Exponentially-weighted moving covariance

Parameters **arg1** : Series, DataFrame, or ndarray

arg2 : Series, DataFrame, or ndarray

com : float. optional

Center of mass: alpha = 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)

freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

adjust : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

Returns **y** : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a "span" parameter s, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s+1) = c/(1+c)$

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.

25.3 Series

25.3.1 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)	Detect missing values (NaN in numeric arrays, None/NaN in 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*)

Detect missing values (NaN in numeric arrays, None/NaN in object arrays)

Parameters **arr** : ndarray or object value

Object to check for null-ness

Returns **isnulled** : array-like of bool or bool

Array or bool indicating whether an object is null or if an array is given which of the element is null.

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

Object to check for *not*-null-ness

Returns **isnulled** : array-like of bool or bool

Array or bool indicating whether an object is *not* null or if an array is given which of the element is *not* null.

25.3.2 Conversion / Constructors

Series.__init__([data, index, dtype, name, copy])	
Series.astype(dtype)	See numpy.ndarray.astype
Series.copy([order])	Return new Series with copy of underlying values

pandas.Series.__init__

`Series.`**`__init__`**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*)

pandas.Series.astype

`Series.`**`astype`**(*dtype*)

See numpy.ndarray.astype

pandas.Series.copy

Series.**copy**(*order='C'*)
Return new Series with copy of underlying values

Returns **cp** : Series

25.3.3 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()	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**()
Lazily iterate over (index, value) tuples

25.3.4 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

Continued on next page

Table 25.19 – continued from previous page

Series.combine_first(other)	Combine Series values, choosing the calling Series's values
Series.round([decimals, out])	Return *a* with each element rounded to the given number of decimals.

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

pandas.Series.round

Series.**round**(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See Also:

numpy.around equivalent function

Continued on next page

Table 25.20 – continued from previous page

25.3.5 Function application, GroupBy

Series.apply(func[, convert_dtype, args])	Invoke function on values of Series. Can be ufunc (a NumPy function
Series.map(arg[, na_action])	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*, *convert_dtype=True*, *args=()*, ***kwds*)

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

Parameters **func** : function

convert_dtype : boolean, default True

Try to find better dtype for elementwise function results. If False, leave as dtype=object

Returns **y** : Series or DataFrame if func returns a Series

See Also:

Series.map For element-wise operations

pandas.Series.map

Series.**map**(*arg*, *na_action=None*)

Map values of Series using input correspondence (which can be a dict, Series, or function)

Parameters **arg** : function, dict, or Series

na_action : {None, 'ignore'}

If 'ignore', propagate NA values

Returns **y** : Series

same index as caller

Examples

```
>>> x
one   1
two   2
three 3

>>> y
1  foo
2  bar
3  baz
```

```
>>> x.map(y)
one   foo
two   bar
three baz
```

pandas.Series.groupby

Series.**groupby**(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns

Parameters **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

axis : int, default 0

level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort : boolean, default True

Sort group keys. Get better performance by turning this off

group_keys : boolean, default True

When calling apply, add group keys to index to identify pieces

squeeze : boolean, default False

reduce the dimensionaility of the return type if possible, otherwise return a consistent type

Returns **GroupBy object** :

Examples

DataFrame result >>> data.groupby(func, axis=0).mean()

DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()

DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()

25.3.6 Computations / Descriptive Stats

Series.abs()	Return an object with absolute value taken.
Series.any([axis, out])	Returns True if any of the elements of *a* evaluate to True.
Series.autocorr()	Lag-1 autocorrelation

Continued on next page

Table 25.21 – continued from previous page

Series.between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right. NA values
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, min_periods])	Compute correlation with *other* Series, excluding missing values
Series.count([level])	Return number of non-NA/null observations in the Series
Series.cov(other[, min_periods])	Compute covariance with Series, excluding missing values
Series.cummax([axis, dtype, out, skipna])	Cumulative max of values.
Series.cummin([axis, dtype, out, skipna])	Cumulative min of values.
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.kurt([skipna, level])	Return unbiased kurtosis of values
Series.mad([skipna, level])	Return mean absolute deviation of values
Series.max([axis, out, skipna, level])	
Series.mean([axis, dtype, out, skipna, level])	Return mean of values
Series.median([axis, dtype, out, skipna, level])	Return median of values
Series.min([axis, out, skipna, level])	
Series.nunique()	Return count of unique elements in the Series
Series.pct_change([periods, fill_method, ...])	Percent change over given number of periods
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.rank([method, na_option, ascending])	Compute data ranks (1 through n).
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
Series.unique()	Return array of unique values in the Series. Significantly faster than
Series.var([axis, dtype, out, ddof, skipna, ...])	Return variance of values
Series.value_counts([normalize])	Returns Series containing counts of unique values. The resulting Series

pandas.Series.abs

Series.**abs**()

Return an object with absolute value taken. Only applicable to objects that are all numeric

Returns **abs: type of caller** :

pandas.Series.any

Series.**any**(*axis=None*, *out=None*)

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See Also:

numpy.any equivalent function

pandas.Series.autocorr

Series.**autocorr**()
Lag-1 autocorrelation

Returns **autocorr** : float

pandas.Series.between

Series.**between**(*left*, *right*, *inclusive=True*)
Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

Parameters **left** : scalar

Left boundary

right : scalar

Right boundary

Returns **is_between** : Series

pandas.Series.clip

Series.**clip**(*lower=None*, *upper=None*, *out=None*)
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'*, *min_periods=None*)

Compute correlation with *other* Series, excluding missing values

Parameters **other** : Series

method : {'pearson', 'kendall', 'spearman'}

pearson : standard correlation coefficient kendall : Kendall Tau correlation coefficient spearman : Spearman rank correlation

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns **correlation** : float

pandas.Series.count

Series.**count**(*level=None*)

Return number of non-NA/null observations in the Series

Parameters **level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

Returns **nobs** : int or Series (if level specified)

pandas.Series.cov

Series.**cov**(*other*, *min_periods=None*)

Compute covariance with Series, excluding missing values

Parameters **other** : Series

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns **covariance** : float

Normalized by N-1 (unbiased estimator). :

pandas.Series.cummax

Series.**cummax**(*axis=0*, *dtype=None*, *out=None*, *skipna=True*)

Cumulative max of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **cummax** : Series

pandas.Series.cummin

Series.**cummin**(*axis=0*, *dtype=None*, *out=None*, *skipna=True*)

Cumulative min of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **cummin** : Series

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.kurt

`Series.`**`kurt`**(*skipna=True*, *level=None*)

Return unbiased kurtosis 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 **kurt** : float (or Series if level specified)

pandas.Series.mad

`Series.`**`mad`**(*skipna=True*, *level=None*)

Return mean absolute 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 **mad** : float (or Series if level specified)

pandas.Series.max

`Series.`**`max`**(*axis=None*, *out=None*, *skipna=True*, *level=None*)

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)

See Also:

`Return`, `NA`

Notes

This method returns the maximum of the values in the Series. If you want the *index* of the maximum, use `Series.idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

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`**(*axis=0*, *dtype=None*, *out=None*, *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*)

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)

See Also:

`Return`, `NA`

Notes

This method returns the minimum of the values in the Series. If you want the *index* of the minimum, use `Series.idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

pandas.Series.nunique

Series.**nunique**()
Return count of unique elements in the Series

Returns **nunique** : int

pandas.Series.pct_change

Series.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwds*)
Percent change over given number of periods

Parameters **periods** : int, default 1

Periods to shift for forming percent change

fill_method : str, default 'pad'

How to handle NAs before computing percent changes

limit : int, default None

The number of consecutive NAs to fill before stopping

freq : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay())

Returns **chg** : Series or DataFrame

pandas.Series.prod

Series.**prod**(*axis=0*, *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 **prod** : 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.rank

Series.**rank**(*method='average'*, *na_option='keep'*, *ascending=True*)

Compute data ranks (1 through n). Equal values are assigned a rank that is the average of the ranks of those values

Parameters **method** : {'average', 'min', 'max', 'first'}

average: average rank of group min: lowest rank in group max: highest rank in group first: ranks assigned in order they appear in the array

na_option : {'keep'}

keep: leave NA values where they are

ascending : boolean, default True

False for ranks by high (1) to low (N)

Returns **ranks** : Series

pandas.Series.skew

Series.**skew**(*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)

Normalized by N-1 (unbiased estimator).

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.unique

`Series.`**`unique`**`()`

Return array of unique values in the Series. Significantly faster than numpy.unique

Returns **uniques** : ndarray

pandas.Series.var

`Series.`**`var`**`(`*axis=None*, *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)

Normalized by N-1 (unbiased estimator).

pandas.Series.value_counts

`Series.`**`value_counts`**`(`*normalize=False*`)`

Returns Series containing counts of unique values. The resulting Series will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values

Parameters **normalize: boolean, default False** :

If True then the Series returned will contain the relative frequencies of the unique values.

Returns **counts** : Series

25.3.7 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.first`(offset)	Convenience method for subsetting initial periods of time series data
`Series.head`([n])	Returns first n rows of Series
`Series.idxmax`([axis, out, skipna])	Index of first occurrence of maximum of values.
`Series.idxmin`([axis, out, skipna])	Index of first occurrence of minimum of values.

Continued on next page

Table 25.22 – continued from previous page

Series.isin(values)	Return boolean vector showing whether each element in the Series is
Series.last(offset)	Convenience method for subsetting final periods of time series data
Series.reindex([index, method, level, ...])	Conform Series to new index with optional filling logic, placing
Series.reindex_like(other[, method, limit, ...])	Reindex Series to match index of another Series, optionally with
Series.rename(mapper[, inplace])	Alter Series index using dict or function
Series.reset_index([level, drop, name, inplace])	Analogous to the DataFrame.reset_index function, see docstring there.
Series.select(crit[, axis])	Return data corresponding to axis labels matching criteria
Series.take(indices[, axis, convert])	Analogous to ndarray.take, return Series corresponding to requested
Series.tail([n])	Returns last n rows of Series
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*, *limit=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'

limit : int, default None

fill_value, method, inplace, limit are passed to fillna

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.first

Series.**first**(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('10D') -> First 10 days

pandas.Series.head

Series.**head**(*n=5*)

Returns first n rows of Series

pandas.Series.idxmax

Series.**idxmax**(*axis=None*, *out=None*, *skipna=True*)

Index of first occurrence of maximum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmax** : Index of minimum of values

See Also:

DataFrame.idxmax

Notes

This method is the Series version of ndarray.argmax.

pandas.Series.idxmin

Series.**idxmin**(*axis=None*, *out=None*, *skipna=True*)

Index of first occurrence of minimum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmin** : Index of minimum of values

See Also:

DataFrame.idxmin

Notes

This method is the Series version of ndarray.argmin.

pandas.Series.isin

Series.**isin**(*values*)

Return boolean vector showing whether each element in the Series is exactly contained in the passed sequence of values

Parameters **values** : sequence

Returns **isin** : Series (boolean dtype)

pandas.Series.last

Series.**last**(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

pandas.Series.reindex

Series.**reindex**(*index=None*, *method=None*, *level=None*, *fill_value=nan*, *limit=None*, *copy=True*, *takeable=False*)

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 NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit : int, default None

Maximum size gap to forward or backward fill

takeable : the labels are locations (and not labels)

Returns **reindexed** : Series

pandas.Series.reindex_like

Series.**reindex_like**(*other*, *method=None*, *limit=None*, *fill_value=nan*)

Reindex Series to match index of another Series, optionally with filling logic

Parameters **other** : Series

method : string or None

See Series.reindex docstring

limit : int, default None

Maximum size gap to forward or backward fill

Returns **reindexed** : Series

Notes

Like calling s.reindex(other.index, method=...)

pandas.Series.rename

Series.**rename**(*mapper*, *inplace=False*)

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.reset_index

Series.**reset_index**(*level=None*, *drop=False*, *name=None*, *inplace=False*)

Analogous to the DataFrame.reset_index function, see docstring there.

Parameters **level** : int, str, tuple, or list, default None

Only remove the given levels from the index. Removes all levels by default

drop : boolean, default False

Do not try to insert index into dataframe columns

name : object, default None

The name of the column corresponding to the Series values

inplace : boolean, default False

Modify the Series in place (do not create a new object)

Returns **resetted** : DataFrame, or Series if drop == True

pandas.Series.select

Series.**select**(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

pandas.Series.take

Series.**take**(*indices*, *axis=0*, *convert=True*)

Analogous to ndarray.take, return Series corresponding to requested indices

Parameters **indices** : list / array of ints

convert : translate negative to positive indices (default)

Returns **taken** : Series

pandas.Series.tail

Series.**tail**(*n=5*)

Returns last n rows of 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 copy : boolean, default True

Returns **truncated** : type of caller

25.3.8 Missing data handling

Series.dropna()	Return Series without null values
Series.fillna([value, method, inplace, limit])	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=None*, *inplace=False*, *limit=None*)

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

limit : int, default None

Maximum size gap to forward or backward fill

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', 'values'}

Interpolation method. 'time' interpolation works on daily and higher resolution data to interpolate given length of interval 'values' using the actual index numeric values

Returns **interpolated** : Series

25.3.9 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.reorder_levels(order)	Rearrange index levels using input order.
Series.sort([axis, kind, order, ascending])	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.swaplevel(i, j[, copy])	Swap levels i and j in a MultiIndex
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, with -1 indicated where nan values are present

pandas.Series.order

Series.**order**(*na_last=True*, *ascending=True*, *kind='mergesort'*)

Sorts Series object, by value, maintaining index-value link

Parameters **na_last** : boolean (optional, default=True)

Put NaN's at beginning or end

ascending : boolean, default True

Sort ascending. Passing False sorts descending

kind : {'mergesort', 'quicksort', 'heapsort'}, default 'mergesort'

Choice of sorting algorithm. See np.sort for more information. 'mergesort' is the only stable algorithm

Returns **y** : Series

pandas.Series.reorder_levels

Series.**reorder_levels**(*order*)

Rearrange index levels using input order. May not drop or duplicate levels

Parameters **order: list of int representing new level order.** :

(reference level by number not by key)

axis: where to reorder levels :

Returns type of caller (new object) :

pandas.Series.sort

`Series.`**`sort`**(*axis=0*, *kind='quicksort'*, *order=None*, *ascending=True*)

Sort values and index labels by value, in place. For compatibility with ndarray API. No return value

Parameters **axis** : int (can only be zero)

kind : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'

Choice of sorting algorithm. See np.sort for more information. 'mergesort' is the only stable algorithm

order : ignored

ascending : boolean, default True

Sort ascending. Passing False sorts descending

See Also:

`pandas.Series.order`

pandas.Series.sort_index

`Series.`**`sort_index`**(*ascending=True*)

Sort object by labels (along an axis)

Parameters **ascending** : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

Returns **sorted_obj** : Series

Examples

```
>>> result1 = s.sort_index(ascending=False)
>>> result2 = s.sort_index(ascending=[1, 0])
```

pandas.Series.sortlevel

`Series.`**`sortlevel`**(*level=0*, *ascending=True*)

Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

Parameters **level** : int

ascending : bool, default True

Returns **sorted** : Series

pandas.Series.swaplevel

Series.**swaplevel**(*i*, *j*, *copy=True*)
Swap levels i and j in a MultiIndex

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **swapped** : Series

pandas.Series.unstack

Series.**unstack**(*level=-1*)
Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame

Parameters **level** : int, string, or list of these, default last level

Level(s) to unstack, can pass level name

Returns **unstacked** : DataFrame

Examples

```
>>> s
one  a   1.
one  b   2.
two  a   3.
two  b   4.

>>> s.unstack(level=-1)
     a   b
one  1.  2.
two  3.  4.

>>> s.unstack(level=0)
   one  two
a  1.   2.
b  3.   4.
```

25.3.10 Combining / joining / merging

Series.append(to_append[, verify_integrity])	Concatenate two or more Series. The indexes must not overlap
Series.replace(to_replace[, value, method, ...])	Replace arbitrary values in a Series
Series.update(other)	Modify Series in place using non-NA values from passed

pandas.Series.append

Series.**append**(*to_append*, *verify_integrity=False*)
Concatenate two or more Series. The indexes must not overlap

Parameters **to_append** : Series or list/tuple of Series

verify_integrity : boolean, default False

If True, raise Exception on creating index with duplicates

Returns **appended** : Series

pandas.Series.replace

`Series.`**`replace`**(*to_replace*, *value=None*, *method='pad'*, *inplace=False*, *limit=None*)

Replace arbitrary values in a Series

Parameters **to_replace** : list or dict

list of values to be replaced or dict of replacement values

value : anything

if to_replace is a list then value is the replacement value

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

limit : int, default None

Maximum size gap to forward or backward fill

Returns **replaced** : Series

See Also:

`fillna`, `reindex`, `asfreq`

Notes

replace does not distinguish between NaN and None

pandas.Series.update

`Series.`**`update`**(*other*)

Modify Series in place using non-NA values from passed Series. Aligns on index

Parameters **other** : Series

25.3.11 Time series-related

`Series.asfreq`(freq[, method, how, normalize])	Convert all TimeSeries inside to specified frequency using DateOffset
`Series.asof`(where)	Return last good (non-NaN) value in TimeSeries if value is NaN for
`Series.shift`([periods, freq, copy])	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

Table 25.26 – continued from previous page

Series.weekday	
Series.resample(rule[, how, axis, ...])	Convenience method for frequency conversion and resampling of regular time-
Series.tz_convert(tz[, copy])	Convert TimeSeries to target time zone
Series.tz_localize(tz[, copy])	Localize tz-naive TimeSeries to target time zone

pandas.Series.asfreq

Series.**asfreq**(*freq*, *method=None*, *how=None*, *normalize=False*)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters **freq** : DateOffset object, or string

method : {'backfill', 'bfill', 'pad', 'ffill', None}

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill methdo

how : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

normalize : bool, default False

Whether to reset output index to midnight

Returns **converted** : type of caller

pandas.Series.asof

Series.**asof**(*where*)

Return last good (non-NaN) value in TimeSeries if value is NaN for requested date.

If there is no good value, NaN is returned.

Parameters **where** : date or array of dates

Returns **value or NaN** :

Notes

Dates are assumed to be sorted

pandas.Series.shift

Series.**shift**(*periods=1*, *freq=None*, *copy=True*, ***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

freq : DateOffset, timedelta, or offset alias 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**

pandas.Series.resample

Series.**resample**(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*)
Convenience method for frequency conversion and resampling of regular time-series data.

Parameters **rule** : the offset string or object representing target conversion

how : string, method for down- or re-sampling, default to 'mean' for downsampling

axis : int, optional, default 0

fill_method : string, fill_method for upsampling, default None

closed : {'right', 'left'}
Which side of bin interval is closed

label : {'right', 'left'}
Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

kind: "period"/"timestamp" :

loffset: timedelta :
Adjust the resampled time labels

limit: int, default None :
Maximum size gap to when reindexing with fill_method

base : int, default 0
For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

pandas.Series.tz_convert

Series.**tz_convert**(*tz*, *copy=True*)

Convert TimeSeries to target time zone

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

Returns **converted** : TimeSeries

pandas.Series.tz_localize

Series.**tz_localize**(*tz*, *copy=True*)

Localize tz-naive TimeSeries to target time zone Entries will retain their "naive" value but will be annotated as being relative to the specified tz.

After localizing the TimeSeries, you may use tz_convert() to get the Datetime values recomputed to a different tz.

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

Returns **localized** : TimeSeries

25.3.12 Plotting

Series.hist([by, ax, grid, xlabelsize, ...])	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**(*by=None*, *ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *figsize=None*, ***kwds*)

Draw histogram of the input series using matplotlib

Parameters **by** : object, optional

If passed, then used to form histograms for separate groups

ax : matplotlib axis object

If not passed, uses gca()

grid : boolean, default True

Whether to show axis grid lines

xlabelsize : int, default None

If specified changes the x-axis label size

xrot : float, default None

rotation of x axis labels

ylabelsize : int, default None

If specified changes the y-axis label size

yrot : float, default None

rotation of y axis labels

figsize : tuple, default None

figure size in inches by default

kwds : keywords

To be passed to the actual plotting function

Notes

See matplotlib documentation online for more on this

pandas.Series.plot

`Series.`**`plot`**(*series*, *label=None*, *kind='line'*, *use_index=True*, *rot=None*, *xticks=None*, *yticks=None*, *xlim=None*, *ylim=None*, *ax=None*, *style=None*, *grid=None*, *legend=False*, *logx=False*, *logy=False*, *secondary_y=False*, ***kwds*)

Plot the input series with the index on the x-axis using matplotlib

Parameters **label** : label argument to provide to plot

kind : {'line', 'bar', 'barh', 'kde', 'density'}

bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estimation plot

use_index : boolean, default True

Plot index as axis tick labels

rot : int, default None

Rotation for tick labels

xticks : sequence

Values to use for the xticks

yticks : sequence

Values to use for the yticks

xlim : 2-tuple/list

ylim : 2-tuple/list

ax : matplotlib axis object

If not passed, uses gca()

style : string, default matplotlib default

matplotlib line style to use

grid : matplotlib grid

legend: matplotlib legend :

logx : boolean, default False

For line plots, use log scaling on x axis

logy : boolean, default False

For line plots, use log scaling on y axis

secondary_y : boolean or sequence of ints, default False

If True then y-axis will be on the right

figsize : a tuple (width, height) in inches

kwds : keywords

Options to pass to matplotlib plotting method

Notes

See matplotlib documentation online for more on this subject

25.3.13 Serialization / IO / Conversion

Series.from_csv(path[, sep, parse_dates, ...])	Read delimited file into Series
Series.to_pickle(path)	Pickle (serialize) object to input file 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
Series.to_sparse([kind, fill_value])	Convert Series to SparseSeries
Series.to_string([buf, na_rep, ...])	Render a string representation of the Series
Series.to_clipboard()	Attempt to write text representation of object to the system clipboard ..

pandas.Series.from_csv

classmethod Series.**from_csv**(*path*, *sep=','*, *parse_dates=True*, *header=None*, *index_col=0*, *encoding=None*)

Read delimited file into Series

Parameters **path** : string file path or file handle / StringIO

sep : string, default ','

Field delimiter

parse_dates : boolean, default True

Parse dates. Different default from read_table

header : int, default 0

Row to use at header (skip prior rows)

index_col : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

Returns **y** : Series

pandas.Series.to_pickle

`Series.`**`to_pickle`**(*path*)

Pickle (serialize) object to input file path

Parameters **path** : string

File path

pandas.Series.to_csv

`Series.`**`to_csv`**(*path*, *index=True*, *sep=','*, *na_rep=''*, *float_format=None*, *header=False*, *index_label=None*, *mode='w'*, *nanRep=None*, *encoding=None*)

Write Series to a comma-separated values (csv) file

Parameters **path** : string file path or file handle / StringIO

na_rep : string, default ''

Missing data representation

float_format : string, default None

Format string for floating point numbers

header : boolean, default False

Write out series name

index : boolean, default True

Write row names (index)

index_label : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

mode : Python write mode, default 'w'

sep : character, default ","

Field delimiter for the output file.

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

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

pandas.Series.to_string

Series.**to_string**(*buf=None*, *na_rep='NaN'*, *float_format=None*, *nanRep=None*, *length=False*, *dtype=False*, *name=False*)

Render a string representation of the Series

Parameters **buf** : StringIO-like, optional

buffer to write to

na_rep : string, optional

string representation of NAN to use, default 'NaN'

float_format : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

length : boolean, default False

Add the Series length

dtype : boolean, default False

Add the Series dtype

name : boolean, default False

Add the Series name (which may be None)

Returns **formatted** : string (if not buffer passed)

pandas.Series.to_clipboard

Series.**to_clipboard**()

Attempt to write text representation of object to the system clipboard

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows:
- OS X:

25.4 DataFrame

25.4.1 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()	return the counts of dtypes in this frame
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.

NOTE: the dtype will be a lower-common-denominator dtype (implicit upcasting) that is to say if the dtypes (even of numeric types) are mixed, the one that accomodates all will be chosen use this with care if you are not dealing with the blocks

e.g. if the dtypes are float16,float32 -> float32 float16,float32,float64 -> float64 int32,uint8 -> int32

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**()

return the counts of dtypes in this frame

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.

NOTE: the dtype will be a lower-common-denominator dtype (implicit upcasting) that is to say if the dtypes (even of numeric types) are mixed, the one that accomodates all will be chosen use this with care if you are not dealing with the blocks

e.g. if the dtypes are float16,float32 -> float32 float16,float32,float64 -> float64 int32,uint8 -> int32

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`**

25.4.2 Conversion / Constructors

`DataFrame.__init__`([data, index, columns, ...])	
`DataFrame.astype`(dtype[, copy, raise_on_error])	Cast object to input numpy.dtype
`DataFrame.convert_objects`([convert_dates, ...])	Attempt to infer better dtype for object columns
`DataFrame.copy`([deep])	Make a copy of this object

pandas.DataFrame.__init__

`DataFrame.`**`__init__`**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

pandas.DataFrame.astype

`DataFrame.`**`astype`**(*dtype*, *copy=True*, *raise_on_error=True*)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **dtype** : numpy.dtype or Python type

raise_on_error : raise on invalid input

Returns **casted** : type of caller

pandas.DataFrame.convert_objects

DataFrame.**convert_objects**(*convert_dates=True*, *convert_numeric=False*, *copy=True*)

Attempt to infer better dtype for object columns

Parameters **convert_dates** : if True, attempt to soft convert_dates, if 'coerce', force conversion (and non-convertibles get NaT)

convert_numeric : if True attempt to coerce to numerbers (including strings), non-convertibles get NaN

copy : boolean, return a copy if True (True by default)

Returns **converted** : DataFrame

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

25.4.3 Indexing, iteration

DataFrame.head([n])	Returns first n rows of DataFrame
DataFrame.ix	
DataFrame.insert(loc, column, value[, ...])	Insert column into DataFrame at specified location.
DataFrame.__iter__()	Iterate over columns of the frame.
DataFrame.iteritems()	Iterator over (column, series) pairs
DataFrame.iterrows()	Iterate over rows of DataFrame as (index, Series) pairs.
DataFrame.itertuples([index])	Iterate over rows of DataFrame as tuples, with index value
DataFrame.lookup(row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame. Given
DataFrame.pop(item)	Return column and drop from frame.
DataFrame.tail([n])	Returns last n rows of DataFrame
DataFrame.xs(key[, axis, level, copy])	Returns a cross-section (row(s) or column(s)) from the DataFrame.

pandas.DataFrame.head

DataFrame.**head**(*n=5*)

Returns first n rows of DataFrame

pandas.DataFrame.ix

DataFrame.**ix**

pandas.DataFrame.insert

DataFrame.**insert**(*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location. if allow_duplicates is False, Raises Exception if column is

already contained in the DataFrame

Parameters **loc** : int

Must have 0 <= loc <= len(columns)

column : object

value : int, Series, or array-like

pandas.DataFrame.__iter__

`DataFrame.__iter__()`
Iterate over columns of the frame.

pandas.DataFrame.iteritems

`DataFrame.iteritems()`
Iterator over (column, series) pairs

pandas.DataFrame.iterrows

`DataFrame.iterrows()`
Iterate over rows of DataFrame as (index, Series) pairs.

Returns **it** : generator

A generator that iterates over the rows of the frame.

Notes

- `iterrows` does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = DataFrame([[1, 1.0]], columns=['x', 'y'])
>>> row = next(df.iterrows())[1]
>>> print row['x'].dtype
float64
>>> print df['x'].dtype
int64
```

pandas.DataFrame.itertuples

`DataFrame.itertuples(index=True)`
Iterate over rows of DataFrame as tuples, with index value as first element of the tuple

pandas.DataFrame.lookup

`DataFrame.lookup(row_labels, col_labels)`
Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

Parameters **row_labels** : sequence

The row labels to use for lookup

col_labels : sequence

The column labels to use for lookup

Notes

Akin to

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

Examples

values [ndarray] The found values

pandas.DataFrame.pop

`DataFrame.`**`pop`**(*item*)

Return column and drop from frame. Raise KeyError if not found.

Returns **column** : Series

pandas.DataFrame.tail

`DataFrame.`**`tail`**(*n=5*)

Returns last n rows of DataFrame

pandas.DataFrame.xs

`DataFrame.`**`xs`**(*key*, *axis=0*, *level=None*, *copy=True*)

Returns a cross-section (row(s) or column(s)) from the DataFrame. Defaults to cross-section on the rows (axis=0).

Parameters **key** : object

Some label contained in the index, or partially in a MultiIndex

axis : int, default 0

Axis to retrieve cross-section on

level : object, defaults to first n levels (n=1 or len(key))

In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

copy : boolean, default True

Whether to make a copy of the data

Returns **xs** : Series or DataFrame

Examples

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
>>> s = df.xs('a', copy=False)
>>> s['A'] = 100
>>> df
     A  B  C
a  100  5  2
b    4  0  9
c    9  7  3

>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
second
three   5  3  5  3
```

25.4.4 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
DataFrame.mul(other[, axis, level, fill_value])	Binary operator multiply with support to substitute a fill_value for missing da
DataFrame.sub(other[, axis, level, fill_value])	Binary operator subtract with support to substitute a fill_value for missing dat
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

Continued on next

Table 25.32 – continued from previous page

DataFrame.rmul(other[, axis, level, fill_value])	Binary operator rmultiply with support to substitute a fill_value for missing da
DataFrame.rsub(other[, axis, level, fill_value])	Binary operator rsubtract with support to substitute a fill_value for missing da
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*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

Parameters **other** : DataFrame

func : function

fill_value : scalar value

overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

Returns **result** : DataFrame

pandas.DataFrame.combineAdd

DataFrame.**combineAdd**(*other*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

Parameters **other** : DataFrame

Returns **DataFrame** :

pandas.DataFrame.combine_first

DataFrame.**combine_first**(*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

Parameters **other** : DataFrame

Returns **combined** : DataFrame

Examples

```
>>> a.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** :

25.4.5 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

See Also:

`DataFrame.applymap` For elementwise operations

Examples

```
>>> df.apply(numpy.sqrt) # returns DataFrame
>>> df.apply(numpy.sum, axis=0) # equiv to df.sum(0)
>>> df.apply(numpy.sum, axis=1) # equiv to df.sum(1)
```

pandas.DataFrame.applymap

DataFrame.**applymap**(*func*)

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing map(func, series) for each series in the DataFrame

Parameters **func** : function

Python function, returns a single value from a single value

Returns **applied** : DataFrame

pandas.DataFrame.groupby

DataFrame.**groupby**(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns

Parameters **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

axis : int, default 0

level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort : boolean, default True

Sort group keys. Get better performance by turning this off

group_keys : boolean, default True

When calling apply, add group keys to index to identify pieces

squeeze : boolean, default False

reduce the dimensionaility of the return type if possible, otherwise return a consistent type

Returns **GroupBy object** :

Examples

DataFrame result >>> data.groupby(func, axis=0).mean()

DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()

DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()

25.4.6 Computations / Descriptive Stats

DataFrame.abs()	Return an object with absolute value taken.
DataFrame.any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis.
DataFrame.clip([lower, upper])	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, min_periods])	Compute pairwise correlation of columns, excluding NA/null values
DataFrame.corrwith(other[, axis, drop])	Compute pairwise correlation between rows or columns of two DataFrame
DataFrame.count([axis, level, numeric_only])	Return Series with number of non-NA/null observations over requested
DataFrame.cov([min_periods])	Compute pairwise covariance of columns, excluding NA/null values
DataFrame.cummax([axis, skipna])	Return DataFrame of cumulative max over requested axis.
DataFrame.cummin([axis, skipna])	Return DataFrame of cumulative min over requested axis.
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.kurt([axis, skipna, level])	Return unbiased kurtosis over requested axis.
DataFrame.mad([axis, skipna, level])	Return mean absolute deviation over requested axis.
DataFrame.max([axis, skipna, level])	
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])	
DataFrame.pct_change([periods, fill_method, ...])	Percent change over given number of periods
DataFrame.prod([axis, skipna, level])	Return product over requested axis.
DataFrame.quantile([q, axis, numeric_only])	Return values at the given quantile over requested axis, a la
DataFrame.rank([axis, numeric_only, method, ...])	Compute numerical data ranks (1 through n) along axis.
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.abs

DataFrame.**abs**()

Return an object with absolute value taken. Only applicable to objects that are all numeric

Returns **abs: type of caller** :

pandas.DataFrame.any

DataFrame.**any**(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*)

Return whether any element is True over requested axis. %(na_action)s

Parameters **axis** : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

bool_only : boolean, default None

Only include boolean data.

Returns **any** : Series (or DataFrame if level specified)

pandas.DataFrame.clip

DataFrame.**clip**(*lower=None*, *upper=None*)

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'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

Parameters **method** : {'pearson', 'kendall', 'spearman'}

pearson : standard correlation coefficient kendall : Kendall Tau correlation coefficient spearman : Spearman rank correlation

min_periods : int, optional

Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

Returns **y** : DataFrame

pandas.DataFrame.corrwith

DataFrame.**corrwith**(*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

Parameters **other** : DataFrame

axis : {0, 1}

0 to compute column-wise, 1 for row-wise

drop : boolean, default False

Drop missing indices from result, default returns union of all

Returns **correls** : Series

pandas.DataFrame.count

DataFrame.**count**(*axis=0*, *level=None*, *numeric_only=False*)

Return Series with number of non-NA/null observations over requested axis. Works with non-floating point data as well (detects NaN and None)

Parameters **axis** : {0, 1}

0 for row-wise, 1 for column-wise

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only : boolean, default False

Include only float, int, boolean data

Returns **count** : Series (or DataFrame if level specified)

pandas.DataFrame.cov

DataFrame.**cov**(*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values

Parameters **min_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result.

Returns **y** : DataFrame

y contains the covariance matrix of the DataFrame's time series. :

The covariance is normalized by N-1 (unbiased estimator). :

pandas.DataFrame.cummax

DataFrame.**cummax**(*axis=None*, *skipna=True*)

Return DataFrame of cumulative max 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.cummin

DataFrame.**cummin**(*axis=None*, *skipna=True*)

Return DataFrame of cumulative min 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.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.kurt

DataFrame.**kurt**(*axis=0*, *skipna=True*, *level=None*)

Return unbiased kurtosis 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 **kurt** : Series (or DataFrame if level specified)

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*)

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)

See Also:

`Return`, `NA`

Notes

This method returns the maximum of the values in the DataFrame. If you want the *index* of the maximum, use `DataFrame.idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

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*)

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)

See Also:

Return, NA

Notes

This method returns the minimum of the values in the DataFrame. If you want the *index* of the minimum, use DataFrame.idxmin. This is the equivalent of the numpy.ndarray method argmin.

pandas.DataFrame.pct_change

DataFrame.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwds*)

Percent change over given number of periods

Parameters **periods** : int, default 1

Periods to shift for forming percent change

fill_method : str, default 'pad'

How to handle NAs before computing percent changes

limit : int, default None

The number of consecutive NAs to fill before stopping

freq : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay())

Returns **chg** : Series or DataFrame

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*, *numeric_only=True*)

Return values at the given quantile over requested axis, a la scoreatpercentile in scipy.stats

Parameters **q** : quantile, default 0.5 (50% quantile)

0 <= q <= 1

axis : {0, 1}

0 for row-wise, 1 for column-wise

Returns **quantiles** : Series

pandas.DataFrame.rank

DataFrame.**rank**(*axis=0*, *numeric_only=None*, *method='average'*, *na_option='keep'*, *ascending=True*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

Parameters **axis** : {0, 1}, default 0

Ranks over columns (0) or rows (1)

numeric_only : boolean, default None

Include only float, int, boolean data

method : {'average', 'min', 'max', 'first'}

average: average rank of group min: lowest rank in group max: highest rank in group first: ranks assigned in order they appear in the array

na_option : {'keep', 'top', 'bottom'}

keep: leave NA values where they are top: smallest rank if ascending bottom: smallest rank if descending

ascending : boolean, default True

False for ranks by high (1) to low (N)

Returns **ranks** : DataFrame

pandas.DataFrame.skew

DataFrame.**skew**(*axis=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)

Normalized by N-1 (unbiased estimator).

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)

Normalized by N-1 (unbiased estimator).

25.4.7 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.drop_duplicates([cols, take_last, ...])	Return DataFrame with duplicate rows removed, optionally only
DataFrame.duplicated([cols, take_last])	Return boolean Series denoting duplicate rows, optionally only
DataFrame.filter([items, like, regex])	Restrict frame's columns to set of items or wildcard
DataFrame.first(offset)	Convenience method for subsetting initial periods of time series data
DataFrame.head([n])	Returns first n rows of DataFrame
DataFrame.idxmax([axis, skipna])	Return index of first occurrence of maximum over requested axis.
DataFrame.idxmin([axis, skipna])	Return index of first occurrence of minimum over requested axis.
DataFrame.last(offset)	Convenience method for subsetting final periods of time series data
DataFrame.reindex([index, columns, method, ...])	Conform DataFrame to new index with optional filling logic, placing
DataFrame.reindex_axis(labels[, axis, ...])	Conform DataFrame to new index with optional filling logic, placing
DataFrame.reindex_like(other[, method, ...])	Reindex DataFrame to match indices of another DataFrame, optionally
DataFrame.rename([index, columns, copy, inplace])	Alter index and / or columns using input function or functions.
DataFrame.reset_index([level, drop, ...])	For DataFrame with multi-level index, return new DataFrame with
DataFrame.select(crit[, axis])	Return data corresponding to axis labels matching criteria
DataFrame.set_index(keys[, drop, append, ...])	Set the DataFrame index (row labels) using one or more existing

Continued on next page

Table 25.35 – continued from previous page

DataFrame.tail([n])	Returns last n rows of DataFrame
DataFrame.take(indices[, axis, convert])	Analogous to ndarray.take, return DataFrame corresponding to requested
DataFrame.truncate([before, after, copy])	Function truncate a sorted DataFrame / Series before and/or after

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*, *limit=None*, *fill_axis=0*)

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

limit : int, default None

fill_axis : {0, 1}, default 0

Filling axis, method and limit

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.drop_duplicates

DataFrame.**drop_duplicates**(*cols=None*, *take_last=False*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

Parameters **cols** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

take_last : boolean, default False

Take the last observed row in a row. Defaults to the first row

inplace : boolean, default False

Whether to drop duplicates in place or to return a copy

Returns **deduplicated** : DataFrame

pandas.DataFrame.duplicated

DataFrame.**duplicated**(*cols=None*, *take_last=False*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

Parameters **cols** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

take_last : boolean, default False

Take the last observed row in a row. Defaults to the first row

Returns **duplicated** : Series

pandas.DataFrame.filter

DataFrame.**filter**(*items=None*, *like=None*, *regex=None*)

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.first

DataFrame.**first**(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('10D') -> First 10 days

pandas.DataFrame.head

DataFrame.**head**(*n=5*)

Returns first n rows of DataFrame

pandas.DataFrame.idxmax

DataFrame.**idxmax**(*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

Parameters **axis** : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be first index.

Returns **idxmax** : Series

See Also:

Series.idxmax

Notes

This method is the DataFrame version of ndarray.argmax.

pandas.DataFrame.idxmin

DataFrame.**idxmin**(*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

Parameters **axis** : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **idxmin** : Series

See Also:

Series.idxmin

Notes

This method is the DataFrame version of `ndarray.argmin`.

pandas.DataFrame.last

DataFrame.**last**(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

pandas.DataFrame.reindex

DataFrame.**reindex**(*index=None*, *columns=None*, *method=None*, *level=None*, *fill_value=nan*, *limit=None*, *copy=True*, *takeable=False*)

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

limit : int, default None

Maximum size gap to forward or backward fill

takeable : the labels are locations (and not labels)

Returns **reindexed** : same type as calling instance

Examples

```
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

pandas.DataFrame.reindex_axis

DataFrame.**reindex_axis**(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform 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

axis : {0, 1}

0 -> index (rows) 1 -> columns

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

limit : int, default None

Maximum size gap to forward or backward fill

Returns **reindexed** : same type as calling instance

See Also:

DataFrame.reindex, DataFrame.reindex_like

Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

pandas.DataFrame.reindex_like

DataFrame.**reindex_like**(*other*, *method=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Reindex DataFrame to match indices of another DataFrame, optionally with filling logic

Parameters **other** : DataFrame

method : string or None

copy : boolean, default True

limit : int, default None

Maximum size gap to forward or backward fill

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*, *inplace=False*)

Alter index and / or columns using input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

Parameters **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

inplace : boolean, default False

Whether to return a new DataFrame. If True then value of copy is ignored.

Returns **renamed** : DataFrame (new object)

See Also:

Series.rename

pandas.DataFrame.reset_index

DataFrame.**reset_index**(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

Parameters **level** : int, str, tuple, or list, default None

Only remove the given levels from the index. Removes all levels by default

drop : boolean, default False

Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace : boolean, default False

Modify the DataFrame in place (do not create a new object)

col_level : int or str, default 0

If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill : object, default ''

If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

Returns **resetted** : DataFrame

pandas.DataFrame.select

DataFrame.**select**(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

pandas.DataFrame.set_index

DataFrame.**set_index**(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

Parameters **keys** : column label or list of column labels / arrays

drop : boolean, default True

Delete columns to be used as the new index

append : boolean, default False

Whether to append columns to existing index

inplace : boolean, default False

Modify the DataFrame in place (do not create a new object)

verify_integrity : boolean, default False

Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

Returns **dataframe** : DataFrame

Examples

```
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])
```

pandas.DataFrame.tail

DataFrame.**tail**(*n=5*)

Returns last n rows of DataFrame

pandas.DataFrame.take

DataFrame.**take**(*indices*, *axis=0*, *convert=True*)

Analogous to ndarray.take, return DataFrame corresponding to requested indices along an axis

Parameters **indices** : list / array of ints

axis : {0, 1}

convert : convert indices for negative values, check bounds, default True

mainly useful for an user routine calling

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 copy : boolean, default True

Returns **truncated** : type of caller

25.4.8 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, ...])	Fill NA/NaN values using the specified method
DataFrame.replace([to_replace, value, ...])	Replace values given in 'to_replace' with 'value'.

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}, or tuple/list thereof

Pass tuple or list to drop on multiple axes

how : {'any', 'all'}

any : if any NA values are present, drop that label all : if all values are NA, drop that label

thresh : int, default None

int value : require that many non-NA values

subset : array-like

Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include

Returns **dropped** : DataFrame

pandas.DataFrame.fillna

DataFrame.**fillna**(*value=None*, *method=None*, *axis=0*, *inplace=False*, *limit=None*, *downcast=None*)

Fill NA/NaN values using the specified method

Parameters **method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

value : scalar 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). This value cannot be a list.

axis : {0, 1}, default 0

0: fill column-by-column 1: fill row-by-row

inplace : boolean, default False

If True, fill 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

limit : int, default None

Maximum size gap to forward or backward fill

downcast : dict, default is None, a dict of item->dtype of what to

downcast if possible

Returns **filled** : DataFrame

See Also:

reindex, asfreq

pandas.DataFrame.replace

DataFrame.**replace**(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method=None*, *axis=None*)

Replace values given in 'to_replace' with 'value'.

Parameters **to_replace** : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- None:
  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or `Series`.

See the examples section for examples of each of these.

value : scalar, dict, list, str, regex, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace : boolean, default False

If True, 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

limit : int, default None

Maximum size gap to forward or backward fill

regex : bool or same types as *to_replace*, default False

Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

Returns **filled** : DataFrame

Raises **AssertionError** :

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError :

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.

ValueError :

- If *to_replace* and *value* are `list` s or `ndarray` s, but they are not the same length.

See Also:

`reindex`, `asfreq`, `fillna`

Notes

•Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

•Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.

•This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

25.4.9 Reshaping, sorting, transposing

`DataFrame.delevel`(*args, **kwargs)	
`DataFrame.pivot`([index, columns, values])	Reshape data (produce a "pivot" table) based on column values.
`DataFrame.reorder_levels`(order[, axis])	Rearrange index levels using input order.
`DataFrame.sort`([columns, column, axis, ...])	Sort DataFrame either by labels (along either axis) or by the values in
`DataFrame.sort_index`([axis, by, ascending, ...])	Sort DataFrame either by labels (along either axis) or by the values in
`DataFrame.sortlevel`([level, axis, ...])	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.to_panel`()	Transform long (stacked) format (DataFrame) into wide (3D, Panel)
`DataFrame.transpose`()	Returns a DataFrame with the rows/columns switched. If the DataFrame is

pandas.DataFrame.delevel

`DataFrame.`**`delevel`**(**args*, ***kwargs*)

pandas.DataFrame.pivot

`DataFrame.`**`pivot`**(*index=None*, *columns=None*, *values=None*)

Reshape data (produce a "pivot" table) based on column values. Uses unique values from index / columns to form axes and return either DataFrame or Panel, depending on whether you request a single value column (DataFrame) or all columns (Panel)

Parameters **index** : string or object

Column name to use to make new frame's index

columns : string or object

Column name to use to make new frame's columns

values : string or object, optional

Column name to use for populating new frame's values

Returns **pivoted** : DataFrame

If no values column specified, will have hierarchically indexed columns

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods

Examples

```
>>> df
    foo   bar  baz
0   one   A    1.
1   one   B    2.
2   one   C    3.
3   two   A    4.
4   two   B    5.
5   two   C    6.

>>> df.pivot('foo', 'bar', 'baz')
     A   B   C
one  1   2   3
two  4   5   6

>>> df.pivot('foo', 'bar')['baz']
     A   B   C
one  1   2   3
two  4   5   6
```

pandas.DataFrame.reorder_levels

DataFrame.**reorder_levels**(*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

Parameters **order: list of int representing new level order.** :

(reference level by number not by key)

axis: where to reorder levels :

Returns **type of caller (new object)** :

pandas.DataFrame.sort

DataFrame.**sort**(*columns=None*, *column=None*, *axis=0*, *ascending=True*, *inplace=False*)

Sort DataFrame either by labels (along either axis) or by the values in column(s)

Parameters **columns** : object

Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

ascending : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

axis : {0, 1}

Sort index/rows versus columns

inplace : boolean, default False

Sort the DataFrame without creating a new instance

Returns **sorted** : DataFrame

Examples

```
>>> result = df.sort(['A', 'B'], ascending=[1, 0])
```

pandas.DataFrame.sort_index

DataFrame.**sort_index**(*axis=0*, *by=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*)

Sort DataFrame either by labels (along either axis) or by the values in a column

Parameters **axis** : {0, 1}

Sort index/rows versus columns

by : object

Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

ascending : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

inplace : boolean, default False

Sort the DataFrame without creating a new instance

Returns **sorted** : DataFrame

Examples

```
>>> result = df.sort_index(by=['A', 'B'], ascending=[1, 0])
```

pandas.DataFrame.sortlevel

DataFrame.**sortlevel**(*level=0*, *axis=0*, *ascending=True*, *inplace=False*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

Parameters **level** : int

axis : {0, 1}

ascending : bool, default True

inplace : boolean, default False

Sort the DataFrame without creating a new instance

Returns **sorted** : DataFrame

pandas.DataFrame.swaplevel

DataFrame.**swaplevel**(*i*, *j*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **swapped** : type of caller (new object)

pandas.DataFrame.stack

DataFrame.**stack**(*level=-1*, *dropna=True*)

Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.

Parameters **level** : int, string, or list of these, default last level

Level(s) to stack, can pass level name

dropna : boolean, default True

Whether to drop rows in the resulting Frame/Series with no valid values

Returns **stacked** : DataFrame or Series

Examples

```
>>> s
     a   b
one  1.  2.
two  3.  4.

>>> s.stack()
one a    1
    b    2
two a    3
    b    4
```

pandas.DataFrame.unstack

DataFrame.**unstack**(*level=-1*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex)

Parameters **level** : int, string, or list of these, default 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.to_panel

`DataFrame.`**`to_panel`**`()`

Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns **panel** : Panel

pandas.DataFrame.transpose

`DataFrame.`**`transpose`**`()`

Returns a DataFrame with the rows/columns switched. If the DataFrame is homogeneously-typed, the data is not copied

25.4.10 Combining / joining / merging

DataFrame.append(other[, ignore_index, ...])	Append columns of other to end of this frame's columns and index, returning a
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.update(other[, join, overwrite, ...])	Modify DataFrame in place using non-NA values from passed

pandas.DataFrame.append

DataFrame.**append**(*other*, *ignore_index=False*, *verify_integrity=False*)

Append columns of other to end of this frame's columns and index, returning a new object. Columns not in this frame are added as new columns.

Parameters **other** : DataFrame or list of Series/dict-like objects

ignore_index : boolean, default False

If True do not use the index labels. Useful for gluing together record arrays

verify_integrity : boolean, default False

If True, raise Exception on creating index with duplicates

Returns **appended** : DataFrame

Notes

If a list of dict is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged

pandas.DataFrame.join

DataFrame.**join**(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

Parameters **other** : DataFrame, Series with name field set, or list of DataFrame

Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on : column name, tuple/list of column names, or array-like

Column(s) to use for joining, otherwise join on index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how : {'left', 'right', 'outer', 'inner'}

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise * left: use calling frame's index * right: use input frame's index * outer: form union of indexes * inner: use intersection of indexes

lsuffix : string

Suffix to use from left frame's overlapping columns

rsuffix : string

Suffix to use from right frame's overlapping columns

sort : boolean, default False

Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

Returns **joined** : DataFrame

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

pandas.DataFrame.merge

DataFrame.**merge**(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters **right** : DataFrame

how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

on : label or list

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

left_on : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

right_on : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left_on docs

left_index : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left_index

sort : boolean, default False

Sort the join keys lexicographically in the result DataFrame

suffixes : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

copy : boolean, default True

If False, do not copy data unnecessarily

Returns **merged** : DataFrame

Examples

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  bar   2        bar   6
1  bar   2        bar   8
2  baz   3        NaN   NaN
3  foo   1        foo   5
4  foo   4        foo   5
5  NaN   NaN      qux   7
```

pandas.DataFrame.update

DataFrame.**update**(*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify DataFrame in place using non-NA values from passed DataFrame. Aligns on indices

Parameters **other** : DataFrame, or object coercible into a DataFrame

join : {'left', 'right', 'outer', 'inner'}, default 'left'

overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

filter_func : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict : bool

If True, will raise an error if the DataFrame and other both contain data in the same place.

25.4.11 Time series-related

DataFrame.asfreq(freq[, method, how, normalize])	Convert all TimeSeries inside to specified frequency using DateOffset
DataFrame.shift([periods, freq])	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
DataFrame.resample(rule[, how, axis, ...])	Convenience method for frequency conversion and resampling of regular t
DataFrame.to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex with desired
DataFrame.to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period
DataFrame.tz_convert(tz[, axis, copy])	Convert TimeSeries to target time zone. If it is time zone naive, it
DataFrame.tz_localize(tz[, axis, copy])	Localize tz-naive TimeSeries to target time zone

pandas.DataFrame.asfreq

DataFrame.**asfreq**(*freq*, *method=None*, *how=None*, *normalize=False*)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters **freq** : DateOffset object, or string

method : {'backfill', 'bfill', 'pad', 'ffill', None}

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill methdo

how : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

normalize : bool, default False

Whether to reset output index to midnight

Returns **converted** : type of caller

pandas.DataFrame.shift

DataFrame.**shift**(*periods=1*, *freq=None*, ***kwds*)

Shift the index of the DataFrame by desired number of periods with an optional time freq

Parameters **periods** : int

Number of periods to move, can be positive or negative

freq : DateOffset, timedelta, or time rule string, optional

Increment to use from datetools module or time rule (e.g. 'EOM')

Returns **shifted** : DataFrame

Notes

If freq is specified then the index values are shifted but the data if not realigned

pandas.DataFrame.first_valid_index

DataFrame.**first_valid_index**()

Return label for first non-NA/null value

pandas.DataFrame.last_valid_index

DataFrame.**last_valid_index**()

Return label for last non-NA/null value

pandas.DataFrame.resample

DataFrame.**resample**(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*)

Convenience method for frequency conversion and resampling of regular time-series data.

Parameters **rule** : the offset string or object representing target conversion

how : string, method for down- or re-sampling, default to 'mean' for

downsampling

axis : int, optional, default 0

fill_method : string, fill_method for upsampling, default None

closed : {'right', 'left'}

Which side of bin interval is closed

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

kind: "period"/"timestamp" :

loffset: timedelta :

Adjust the resampled time labels

limit: int, default None :

Maximum size gap to when reindexing with fill_method

base : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

pandas.DataFrame.to_period

DataFrame.**to_period**(*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters **freq** : string, default

axis : {0, 1}, default 0

The axis to convert (the index by default)

copy : boolean, default True

If False then underlying input data is not copied

Returns **ts** : TimeSeries with PeriodIndex

pandas.DataFrame.to_timestamp

DataFrame.**to_timestamp**(*freq=None*, *how='start'*, *axis=0*, *copy=True*)

Cast to DatetimeIndex of timestamps, at *beginning* of period

Parameters **freq** : string, default frequency of PeriodIndex

Desired frequency

how : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

axis : {0, 1} default 0

The axis to convert (the index by default)

copy : boolean, default True

If false then underlying input data is not copied

Returns **df** : DataFrame with DatetimeIndex

pandas.DataFrame.tz_convert

`DataFrame.`**`tz_convert`**(*tz*, *axis=0*, *copy=True*)

Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

pandas.DataFrame.tz_localize

`DataFrame.`**`tz_localize`**(*tz*, *axis=0*, *copy=True*)

Localize tz-naive TimeSeries to target time zone

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

25.4.12 Plotting

`DataFrame.boxplot`([column, by, ax, ...])	Make a box plot from DataFrame column/columns optionally grouped
`DataFrame.hist`(data[, column, by, grid, ...])	Draw Histogram the DataFrame's series using matplotlib / pylab.
`DataFrame.plot`([frame, x, y, subplots, ...])	Make line or bar plot of DataFrame's series with the index on the x-axis

pandas.DataFrame.boxplot

`DataFrame.`**`boxplot`**(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, ***kwds*)

Make a box plot from DataFrame column/columns optionally grouped (stratified) by one or more columns

Parameters **data** : DataFrame

column : column names or list of names, or vector

Can be any valid input to groupby

by : string or sequence

Column in the DataFrame to group by ax : matplotlib axis object, default None

fontsize : int or string

rot : int, default None Rotation for ticks grid : boolean, default None (matlab style default) Axis grid lines

Returns **ax** : matplotlib.axes.AxesSubplot

pandas.DataFrame.hist

`DataFrame.`**`hist`**(*data*, *column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, ***kwds*)

Draw Histogram the DataFrame's series using matplotlib / pylab.

Parameters **data** : DataFrame

column : string or sequence

If passed, will be used to limit data to a subset of columns

by : object, optional

If passed, then used to form histograms for separate groups

grid : boolean, default True

Whether to show axis grid lines

xlabelsize : int, default None

If specified changes the x-axis label size

xrot : float, default None

rotation of x axis labels

ylabelsize : int, default None

If specified changes the y-axis label size

yrot : float, default None

rotation of y axis labels

ax : matplotlib axes object, default None

sharex : bool, if True, the X axis will be shared amongst all subplots.

sharey : bool, if True, the Y axis will be shared amongst all subplots.

figsize : tuple

The size of the figure to create in inches by default

layout: (optional) a tuple (rows, columns) for the layout of the histograms :

kwds : other plotting keyword arguments

To be passed to hist function

pandas.DataFrame.plot

DataFrame.**plot**(*frame=None*, *x=None*, *y=None*, *subplots=False*, *sharex=True*, *sharey=False*, *use_index=True*, *figsize=None*, *grid=None*, *legend=True*, *rot=None*, *ax=None*, *style=None*, *title=None*, *xlim=None*, *ylim=None*, *logx=False*, *logy=False*, *xticks=None*, *yticks=None*, *kind='line'*, *sort_columns=False*, *fontsize=None*, *secondary_y=False*, ***kwds*)

Make line or bar plot of DataFrame's series with the index on the x-axis using matplotlib / pylab.

Parameters **frame** : DataFrame

x : label or position, default None

y : label or position, default None

Allows plotting of one column versus another

subplots : boolean, default False

Make separate subplots for each time series

sharex : boolean, default True

In case subplots=True, share x axis

sharey : boolean, default False

In case subplots=True, share y axis

use_index : boolean, default True

Use index as ticks for x axis

stacked : boolean, default False

If True, create stacked bar plot. Only valid for DataFrame input

sort_columns: boolean, default False :

Sort column names to determine plot ordering

title : string

Title to use for the plot

grid : boolean, default None (matlab style default)

Axis grid lines

legend : boolean, default True

Place legend on axis subplots

ax : matplotlib axis object, default None

style : list or dict

matplotlib line style per column

kind : {'line', 'bar', 'barh', 'kde', 'density'}

bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estimation plot

logx : boolean, default False

For line plots, use log scaling on x axis

logy : boolean, default False

For line plots, use log scaling on y axis

xticks : sequence

Values to use for the xticks

yticks : sequence

Values to use for the yticks

xlim : 2-tuple/list

ylim : 2-tuple/list

rot : int, default None

Rotation for ticks

secondary_y : boolean or sequence, default False

Whether to plot on the secondary y-axis If a list/tuple, which columns to plot on secondary y-axis

mark_right: boolean, default True :

When using a secondary_y axis, should the legend label the axis of the various columns automatically

colormap : str or matplotlib colormap object, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

kwds : keywords

Options to pass to matplotlib plotting method

Returns **ax_or_axes** : matplotlib.AxesSubplot or list of them

25.4.13 Serialization / IO / Conversion

DataFrame.from_csv(path[, header, sep, ...])	Read delimited file into DataFrame
DataFrame.from_dict(data[, orient, dtype])	Construct DataFrame from dict of array-like or dicts
DataFrame.from_items(items[, columns, orient])	Convert (key, value) pairs to DataFrame. The keys will be the axis
DataFrame.from_records(data[, index, ...])	Convert structured or record ndarray to DataFrame
DataFrame.info([verbose, buf, max_cols])	Concise summary of a DataFrame, used in __repr__ when very large.
DataFrame.to_pickle(path)	Pickle (serialize) object to input file path
DataFrame.to_csv(path_or_buf[, sep, na_rep, ...])	Write DataFrame to a comma-separated values (csv) file
DataFrame.to_hdf(path_or_buf, key, **kwargs)	activate the HDFStore
DataFrame.to_dict([outtype])	Convert DataFrame to dictionary.
DataFrame.to_excel(excel_writer[, ...])	Write DataFrame to a excel sheet
DataFrame.to_json([path_or_buf, orient, ...])	Convert the object to a JSON string.
DataFrame.to_html([buf, columns, col_space, ...])	to_html-specific options
DataFrame.to_stata(fname[, convert_dates, ...])	A class for writing Stata binary dta files from array-like objects
DataFrame.to_records([index, convert_datetime64])	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.to_clipboard()	Attempt to write text representation of object to the system clipboard ..

pandas.DataFrame.from_csv

classmethod DataFrame.**from_csv**(*path*, *header=0*, *sep=', '*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=False*)

Read delimited file into DataFrame

Parameters **path** : string file path or file handle / StringIO

header : int, default 0

Row to use at header (skip prior rows)

sep : string, default ','

Field delimiter

index_col : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates : boolean, default True

Parse dates. Different default from read_table

tupleize_cols : boolean, default True

write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

Returns **y** : DataFrame

Notes

Preferable to use read_table for most general purposes but from_csv makes for an easy roundtrip to and from file, especially with a DataFrame of time series data

pandas.DataFrame.from_dict

classmethod DataFrame.**from_dict**(*data*, *orient='columns'*, *dtype=None*)

Construct DataFrame from dict of array-like or dicts

Parameters **data** : dict

{field : array-like} or {field : dict}

orient : {'columns', 'index'}, default 'columns'

The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

Returns **DataFrame** :

pandas.DataFrame.from_items

classmethod DataFrame.**from_items**(*items*, *columns=None*, *orient='columns'*)

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

Parameters **items** : sequence of (key, value) pairs

Values should be arrays or Series.

columns : sequence of column labels, optional

Must be passed if orient='index'.

orient : {'columns', 'index'}, default 'columns'

The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

Returns **frame** : DataFrame

pandas.DataFrame.from_records

classmethod DataFrame.**from_records**(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

Parameters **data** : ndarray (structured dtype), list of tuples, dict, or DataFrame

index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude: sequence, default None :

Columns or fields to exclude

columns : sequence, default None

Column names to use. If the passed data do not have named associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float : boolean, default False

Attempt to convert values to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

Returns **df** : DataFrame

pandas.DataFrame.info

DataFrame.**info**(*verbose=True*, *buf=None*, *max_cols=None*)

Concise summary of a DataFrame, 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

max_cols : int, default None

Determines whether full summary or short summary is printed

pandas.DataFrame.to_pickle

DataFrame.**to_pickle**(*path*)

Pickle (serialize) object to input file path

Parameters **path** : string

File path

pandas.DataFrame.to_csv

DataFrame.**to_csv**(*path_or_buf*, *sep=','*, *na_rep=''*, *float_format=None*, *cols=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *nanRep=None*, *encoding=None*, *quoting=None*, *line_terminator='n'*, *chunksize=None*, *tupleize_cols=True*, ***kwds*)

Write DataFrame to a comma-separated values (csv) file

Parameters **path_or_buf** : string or file handle / StringIO

File path

sep : character, default ","

Field delimiter for the output file.

na_rep : string, default ''

Missing data representation

float_format : string, default None

Format string for floating point numbers

cols : sequence, optional

Columns to write

header : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

index : boolean, default True

Write row names (index)

index_label : string or sequence, or False, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

nanRep : None

deprecated, use na_rep

mode : str

Python write mode, default 'w'

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

line_terminator : string, default '\n'

The newline character or character sequence to use in the output file

quoting : optional constant from csv module

defaults to csv.QUOTE_MINIMAL

chunksize : int or None

rows to write at a time

tupleize_cols : boolean, default True

write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

pandas.DataFrame.to_hdf

`DataFrame.`**`to_hdf`**(*path_or_buf*, *key*, ***kwargs*)

activate the HDFStore

pandas.DataFrame.to_dict

`DataFrame.`**`to_dict`**(*outtype='dict'*)

Convert DataFrame to dictionary.

Parameters **outtype** : str {'dict', 'list', 'series'}

Determines the type of the values of the dictionary. The default *dict* is a nested dictionary {column -> {index -> value}}. *list* returns {column -> list(values)}. *series* returns {column -> Series(values)}. Abbreviations are allowed.

Returns **result** : dict like {column -> {index -> value}}

pandas.DataFrame.to_excel

`DataFrame.`**`to_excel`**(*excel_writer*, *sheet_name='sheet1'*, *na_rep=''*, *float_format=None*, *cols=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*)

Write DataFrame to a excel sheet

Parameters **excel_writer** : string or ExcelWriter object

File path or existing ExcelWriter

sheet_name : string, default 'sheet1'

Name of sheet which will contain DataFrame

na_rep : string, default ''

Missing data representation

float_format : string, default None

Format string for floating point numbers

cols : sequence, optional

Columns to write

header : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

index : boolean, default True

Write row names (index)

index_label : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook >>> writer = ExcelWriter('output.xlsx') >>> df1.to_excel(writer,'sheet1') >>> df2.to_excel(writer,'sheet2') >>> writer.save()

pandas.DataFrame.to_json

`DataFrame.`**`to_json`**(*path_or_buf=None*, *orient=None*, *date_format='epoch'*, *double_precision=10*, *force_ascii=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters **path_or_buf** : the path or buffer to write the result string

if this is None, return a StringIO of the converted string

orient : string

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- The format of the JSON string
  - split : dict like {index -> [index], columns -> [columns], data -> [values]}
  - records : list like [{column -> value}, ... , {column -> value}]
  - index : dict like {index -> {column -> value}}
  - columns : dict like {column -> {index -> value}}
  - values : just the values array

date_format : type of date conversion (epoch = epoch milliseconds, iso = ISO8601)

default is epoch

double_precision : The number of decimal places to use when encoding

floating point values, default 10.

force_ascii : force encoded string to be ASCII, default True.

Returns **result** : a JSON compatible string written to the path_or_buf;

if the path_or_buf is none, return a StringIO of the result

pandas.DataFrame.to_html

`DataFrame.`**`to_html`**(*buf=None*, *columns=None*, *col_space=None*, *colSpace=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *force_unicode=None*, *bold_rows=True*, *classes=None*, *escape=True*)

to_html-specific options

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

Render a DataFrame as an HTML table.

Parameters **frame** : DataFrame

object to render

buf : StringIO-like, optional

buffer to write to

columns : sequence, optional

the subset of columns to write; default None writes all columns

col_space : int, optional

the minimum width of each column

header : bool, optional

whether to print column labels, default True

index : bool, optional

whether to print index (row) labels, default True

na_rep : string, optional

string representation of NAN to use, default 'NaN'

formatters : list or dict of one-parameter functions, optional

formatter functions to apply to columns' elements by position or name, default None, if the result is a string , it must be a unicode string. List must be of length equal to the number of columns.

float_format : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

sparsify : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

justify : {'left', 'right'}, default None

Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_printoptions), 'right' out of the box.

index_names : bool, optional

Prints the names of the indexes, default True

force_unicode : bool, default False

Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

Returns **formatted** : string (or unicode, depending on data and options)

pandas.DataFrame.to_stata

DataFrame.**to_stata**(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*)

A class for writing Stata binary dta files from array-like objects

Parameters **fname** : file path or buffer

Where to save the dta file.

convert_dates : dict

Dictionary mapping column of datetime types to the stata internal format that you want to use for the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either a number or a name.

encoding : str

Default is latin-1. Note that Stata does not support unicode.

byteorder : str

Can be ">", "<", "little", or "big". The default is None which uses *sys.byteorder*

Examples

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

Or with dates

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

pandas.DataFrame.to_records

DataFrame.**to_records**(*index=True*, *convert_datetime64=True*)

Convert DataFrame to record array. Index will be put in the 'index' field of the record array if requested

Parameters **index** : boolean, default True

Include index in resulting record array, stored in 'index' field

convert_datetime64 : boolean, default True

Whether to convert the index to datetime.datetime if it is a DatetimeIndex

Returns **y** : recarray

pandas.DataFrame.to_sparse

DataFrame.**to_sparse**(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

Parameters **fill_value** : float, default NaN

kind : {'block', 'integer'}

Returns **y** : SparseDataFrame

pandas.DataFrame.to_string

DataFrame.**to_string**(*buf=None*, *columns=None*, *col_space=None*, *colSpace=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *nanRep=None*, *index_names=True*, *justify=None*, *force_unicode=None*, *line_width=None*)

Render a DataFrame to a console-friendly tabular output.

Parameters **frame** : DataFrame

object to render

buf : StringIO-like, optional

buffer to write to

columns : sequence, optional

the subset of columns to write; default None writes all columns

col_space : int, optional

the minimum width of each column

header : bool, optional

whether to print column labels, default True

index : bool, optional

whether to print index (row) labels, default True

na_rep : string, optional

string representation of NAN to use, default 'NaN'

formatters : list or dict of one-parameter functions, optional

formatter functions to apply to columns' elements by position or name, default None, if the result is a string , it must be a unicode string. List must be of length equal to the number of columns.

float_format : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

sparsify : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

justify : {'left', 'right'}, default None

Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_printoptions), 'right' out of the box.

index_names : bool, optional

Prints the names of the indexes, default True

force_unicode : bool, default False

Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

Returns **formatted** : string (or unicode, depending on data and options)

pandas.DataFrame.to_clipboard

`DataFrame.`**`to_clipboard`**`()`

Attempt to write text representation of object to the system clipboard

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows:
- OS X:

25.5 Panel

25.5.1 Attributes and underlying data

Axes

- **items**: axis 0; each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1; the index (rows) of each of the DataFrames
- **minor_axis**: axis 2; the columns of each of the DataFrames

`Panel.values`
`Panel.axes`
`Panel.ndim`
`Panel.shape`

pandas.Panel.values

`Panel.`**`values`**

pandas.Panel.axes

`Panel.`**`axes`**

pandas.Panel.ndim

`Panel.`**`ndim`**

pandas.Panel.shape

`Panel.`**`shape`**

25.5.2 Conversion / Constructors

`Panel.__init__`([data, items, major_axis, ...])	
`Panel.astype`(dtype[, copy, raise_on_error])	Cast object to input numpy.dtype
`Panel.copy`([deep])	Make a copy of this object

pandas.Panel.__init__

`Panel.`**`__init__`**(*data=None*, *items=None*, *major_axis=None*, *minor_axis=None*, *copy=False*, *dtype=None*)

pandas.Panel.astype

`Panel.`**`astype`**(*dtype*, *copy=True*, *raise_on_error=True*)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **dtype** : numpy.dtype or Python type

raise_on_error : raise on invalid input

Returns **casted** : type of caller

pandas.Panel.copy

`Panel.`**`copy`**(*deep=True*)

Make a copy of this object

Parameters **deep** : boolean, default True

Make a deep copy, i.e. also copy data

Returns **copy** : type of caller

25.5.3 Getting and setting

`Panel.get_value`(*args)	Quickly retrieve single value at (item, major, minor) location
`Panel.set_value`(*args)	Quickly set single value at (item, major, minor) location

pandas.Panel.get_value

`Panel.`**`get_value`**(**args*)

Quickly retrieve single value at (item, major, minor) location

Parameters **item** : item label (panel item)

major : major axis label (panel item row)

minor : minor axis label (panel item column)

Returns **value** : scalar value

pandas.Panel.set_value

`Panel.`**`set_value`**`(*args)`

Quickly set single value at (item, major, minor) location

Parameters **item** : item label (panel item)

major : major axis label (panel item row)

minor : minor axis label (panel item column)

value : scalar

Returns **panel** : Panel

If label combo is contained, will be reference to calling Panel, otherwise a new object

25.5.4 Indexing, iteration, slicing

Panel.ix	
Panel.__iter__()	
Panel.iteritems()	
Panel.pop(item)	Return item slice from panel and delete from panel
Panel.xs(key[, axis, copy])	Return slice of panel along selected axis
Panel.major_xs(key[, copy])	Return slice of panel along major axis
Panel.minor_xs(key[, copy])	Return slice of panel along minor axis

pandas.Panel.ix

`Panel.`**`ix`**

pandas.Panel.__iter__

`Panel.`**`__iter__`**`()`

pandas.Panel.iteritems

`Panel.`**`iteritems`**`()`

pandas.Panel.pop

`Panel.`**`pop`**`(item)`

Return item slice from panel and delete from panel

Parameters **key** : object

Must be contained in panel's items

Returns **y** : DataFrame

pandas.Panel.xs

`Panel.`**`xs`**(*key*, *axis=1*, *copy=True*)

Return slice of panel along selected axis

Parameters **key** : object

Label

axis : {'items', 'major', 'minor}, default 1/'major'

Returns **y** : ndim(self)-1

pandas.Panel.major_xs

`Panel.`**`major_xs`**(*key*, *copy=True*)

Return slice of panel along major axis

Parameters **key** : object

Major axis label

copy : boolean, default False

Copy data

Returns **y** : DataFrame

index -> minor axis, columns -> items

pandas.Panel.minor_xs

`Panel.`**`minor_xs`**(*key*, *copy=True*)

Return slice of panel along minor axis

Parameters **key** : object

Minor axis label

copy : boolean, default False

Copy data

Returns **y** : DataFrame

index -> major axis, columns -> items

25.5.5 Binary operator functions

`Panel.add`(other[, axis])	Wrapper method for <built-in function add>
`Panel.div`(other[, axis])	Wrapper method for <built-in function div>
`Panel.mul`(other[, axis])	Wrapper method for <built-in function mul>
`Panel.sub`(other[, axis])	Wrapper method for <built-in function sub>

pandas.Panel.add

Panel.**add**(*other*, *axis=0*)
Wrapper method for <built-in function add>

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over :

Returns **Panel** :

pandas.Panel.div

Panel.**div**(*other*, *axis=0*)
Wrapper method for <built-in function div>

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over :

Returns **Panel** :

pandas.Panel.mul

Panel.**mul**(*other*, *axis=0*)
Wrapper method for <built-in function mul>

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over :

Returns **Panel** :

pandas.Panel.sub

Panel.**sub**(*other*, *axis=0*)
Wrapper method for <built-in function sub>

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over :

Returns **Panel** :

25.5.6 Function application, GroupBy

Panel.apply(func[, axis])	Apply
Panel.groupby(function[, axis])	Group data on given axis, returning GroupBy object

pandas.Panel.apply

Panel.**apply**(*func*, *axis='major'*)

Apply

Parameters **func** : numpy function

Signature should match numpy.{sum, mean, var, std} etc.

axis : {'major', 'minor', 'items'}

fill_value : boolean, default True

Replace NaN values with specified first

Returns **result** : DataFrame or Panel

pandas.Panel.groupby

Panel.**groupby**(*function*, *axis='major'*)

Group data on given axis, returning GroupBy object

Parameters **function** : callable

Mapping function for chosen access

axis : {'major', 'minor', 'items'}, default 'major'

Returns **grouped** : PanelGroupBy

25.5.7 Computations / Descriptive Stats

Panel.abs()	Return an object with absolute value taken.
Panel.count([axis])	Return number of observations over requested axis.
Panel.cummax([axis, skipna])	Return DataFrame of cumulative max over requested axis.
Panel.cummin([axis, skipna])	Return DataFrame of cumulative min over requested axis.
Panel.cumprod([axis, skipna])	Return cumulative product over requested axis as DataFrame
Panel.cumsum([axis, skipna])	Return DataFrame of cumulative sums over requested axis.
Panel.max([axis, skipna])	Return maximum over requested axis
Panel.mean([axis, skipna])	Return mean over requested axis
Panel.median([axis, skipna])	Return median over requested axis
Panel.min([axis, skipna])	Return minimum over requested axis
Panel.pct_change([periods, fill_method, ...])	Percent change over given number of periods
Panel.prod([axis, skipna])	Return product over requested axis
Panel.skew([axis, skipna])	Return unbiased skewness over requested axis
Panel.sum([axis, skipna])	Return sum over requested axis
Panel.std([axis, skipna])	Return unbiased standard deviation over requested axis
Panel.var([axis, skipna])	Return unbiased variance over requested axis

pandas.Panel.abs

Panel.**abs**()

Return an object with absolute value taken. Only applicable to objects that are all numeric

Returns **abs: type of caller** :

pandas.Panel.count

Panel.**count**(*axis='major'*)

Return number of observations over requested axis.

Parameters **axis** : {'items', 'major', 'minor'} or {0, 1, 2}

Returns **count** : DataFrame

pandas.Panel.cummax

Panel.**cummax**(*axis=None*, *skipna=True*)

Return DataFrame of cumulative max 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.Panel.cummin

Panel.**cummin**(*axis=None*, *skipna=True*)

Return DataFrame of cumulative min 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.Panel.cumprod

Panel.**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.Panel.cumsum

Panel.**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.Panel.max

`Panel.`**`max`**(*axis='major'*, *skipna=True*)

Return maximum over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **maximum** : DataFrame

pandas.Panel.mean

`Panel.`**`mean`**(*axis='major'*, *skipna=True*)

Return mean over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **mean** : DataFrame

pandas.Panel.median

`Panel.`**`median`**(*axis='major'*, *skipna=True*)

Return median over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **median** : DataFrame

pandas.Panel.min

`Panel.`**`min`**(*axis='major'*, *skipna=True*)

Return minimum over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **minimum** : DataFrame

pandas.Panel.pct_change

Panel.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwds*)

Percent change over given number of periods

Parameters **periods** : int, default 1

Periods to shift for forming percent change

fill_method : str, default 'pad'

How to handle NAs before computing percent changes

limit : int, default None

The number of consecutive NAs to fill before stopping

freq : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay())

Returns **chg** : Series or DataFrame

pandas.Panel.prod

Panel.**prod**(*axis='major'*, *skipna=True*)

Return product over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **prod** : DataFrame

pandas.Panel.skew

Panel.**skew**(*axis='major'*, *skipna=True*)

Return unbiased skewness over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **skew** : DataFrame

pandas.Panel.sum

Panel.**sum**(*axis='major'*, *skipna=True*)

Return sum over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **sum** : DataFrame

pandas.Panel.std

Panel.**std**(*axis='major'*, *skipna=True*)

Return unbiased standard deviation over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **stdev** : DataFrame

pandas.Panel.var

Panel.**var**(*axis='major'*, *skipna=True*)

Return unbiased variance over requested axis

Parameters **axis** : {items, major_axis, minor_axis} or {0, 1, 2}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **variance** : DataFrame

25.5.8 Reindexing / Selection / Label manipulation

Panel.add_prefix(prefix)	Concatenate prefix string with panel items names.
Panel.add_suffix(suffix)	Concatenate suffix string with panel items names
Panel.drop(labels[, axis, level])	Return new object with labels in requested axis removed
Panel.filter(items)	Restrict items in panel to input list
Panel.first(offset)	Convenience method for subsetting initial periods of time series data
Panel.last(offset)	Convenience method for subsetting final periods of time series data
Panel.reindex([major, minor, method, ...])	Conform panel to new axis or axes
Panel.reindex_axis(labels[, axis, method, ...])	Conform Panel to new index with optional filling logic, placing
Panel.reindex_like(other[, method])	return an object with matching indicies to myself
Panel.select(crit[, axis])	Return data corresponding to axis labels matching criteria
Panel.take(indices[, axis, convert])	Analogous to ndarray.take
Panel.truncate([before, after, axis])	Function truncates a sorted Panel before and/or after some

pandas.Panel.add_prefix

Panel.**add_prefix**(*prefix*)

Concatenate prefix string with panel items names.

Parameters **prefix** : string

Returns **with_prefix** : type of caller

pandas.Panel.add_suffix

Panel.**add_suffix**(*suffix*)

Concatenate suffix string with panel items names

Parameters **suffix** : string

Returns **with_suffix** : type of caller

pandas.Panel.drop

`Panel.`**`drop`**(*labels*, *axis=0*, *level=None*)

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.Panel.filter

`Panel.`**`filter`**(*items*)

Restrict items in panel to input list

Parameters **items** : sequence

Returns **y** : Panel

pandas.Panel.first

`Panel.`**`first`**(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('10D') -> First 10 days

pandas.Panel.last

`Panel.`**`last`**(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

pandas.Panel.reindex

Panel.**reindex**(*major=None*, *minor=None*, *method=None*, *major_axis=None*, *minor_axis=None*, *copy=True*, ***kwargs*)

Conform panel to new axis or axes

Parameters **major** : Index or sequence, default None

Can also use 'major_axis' keyword

items : Index or sequence, default None

minor : Index or sequence, default None

Can also use 'minor_axis' keyword

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series

pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

copy : boolean, default True

Return a new object, even if the passed indexes are the same

Returns **Panel (new object)** :

pandas.Panel.reindex_axis

Panel.**reindex_axis**(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*)

Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

Parameters **index** : array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0, 1}

0 -> index (rows) 1 -> columns

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

Returns **reindexed** : Panel

pandas.Panel.reindex_like

Panel.**reindex_like**(*other*, *method=None*)

return an object with matching indicies to myself

Parameters **other** : Panel

method : string or None

Returns **reindexed** : Panel

pandas.Panel.select

`Panel.`**`select`**(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

pandas.Panel.take

`Panel.`**`take`**(*indices*, *axis=0*, *convert=True*)

Analogous to ndarray.take

Parameters **indices** : list / array of ints

axis : int, default 0

convert : translate neg to pos indices (default)

Returns **taken** : type of caller

pandas.Panel.truncate

`Panel.`**`truncate`**(*before=None*, *after=None*, *axis='major'*)

Function truncates a sorted Panel before and/or after some particular values on the requested axis

Parameters **before** : date

Left boundary

after : date

Right boundary

axis : {'major', 'minor', 'items'}

Returns **Panel** :

25.5.9 Missing data handling

`Panel.dropna`([axis, how])	Drop 2D from panel, holding passed axis constant
`Panel.fillna`([value, method])	Fill NaN values using the specified method.

pandas.Panel.dropna

`Panel.`**`dropna`**(*axis=0*, *how='any'*)

Drop 2D from panel, holding passed axis constant

Parameters **axis** : int, default 0

Axis to hold constant. E.g. axis=1 will drop major_axis entries having a certain amount of NA data

how : {'all', 'any'}, default 'any'

'any': one or more values are NA in the DataFrame along the axis. For 'all' they all must be.

Returns **dropped** : Panel

pandas.Panel.fillna

Panel.**fillna**(*value=None*, *method=None*)

Fill NaN values using the specified method.

Member Series / TimeSeries are filled separately.

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

Returns **y** : DataFrame

See Also:

DataFrame.reindex, DataFrame.asfreq

25.5.10 Reshaping, sorting, transposing

Panel.sort_index([axis, ascending])	Sort object by labels (along an axis)
Panel.swaplevel(i, j[, axis])	Swap levels i and j in a MultiIndex on a particular axis
Panel.transpose(*args, **kwargs)	Permute the dimensions of the Panel
Panel.swapaxes([axis1, axis2, copy])	Interchange axes and swap values axes appropriately
Panel.conform(frame[, axis])	Conform input DataFrame to align with chosen axis pair.

pandas.Panel.sort_index

Panel.**sort_index**(*axis=0*, *ascending=True*)

Sort object by labels (along an axis)

Parameters **axis** : {0, 1}

Sort index/rows versus columns

ascending : boolean, default True

Sort ascending vs. descending

Returns **sorted_obj** : type of caller

pandas.Panel.swaplevel

Panel.**swaplevel**(*i*, *j*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **swapped** : type of caller (new object)

pandas.Panel.transpose

Panel.**transpose**(**args*, ***kwargs*)

Permute the dimensions of the Panel

Parameters **items** : int or one of {'items', 'major', 'minor'}

major : int or one of {'items', 'major', 'minor'}

minor : int or one of {'items', 'major', 'minor'}

copy : boolean, default False

Make a copy of the underlying data. Mixed-dtype data will always result in a copy

Returns **y** : Panel (new object)

Examples

```
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

pandas.Panel.swapaxes

Panel.**swapaxes**(*axis1='major'*, *axis2='minor'*, *copy=True*)

Interchange axes and swap values axes appropriately

Returns **y** : Panel (new object)

pandas.Panel.conform

Panel.**conform**(*frame*, *axis='items'*)

Conform input DataFrame to align with chosen axis pair.

Parameters **frame** : DataFrame

axis : {'items', 'major', 'minor'}

Axis the input corresponds to. E.g., if axis='major', then the frame's columns would be items, and the index would be values of the minor axis

Returns **DataFrame** :

25.5.11 Combining / joining / merging

Panel.join(other[, how, lsuffix, rsuffix])	Join items with other Panel either on major and minor axes column
Panel.update(other[, join, overwrite, ...])	Modify Panel in place using non-NA values from passed

pandas.Panel.join

Panel.**join**(*other*, *how='left'*, *lsuffix=''*, *rsuffix=''*)

Join items with other Panel either on major and minor axes column

Parameters **other** : Panel or list of Panels

Index should be similar to one of the columns in this one

how : {'left', 'right', 'outer', 'inner'}

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise * left: use calling frame's index * right: use input frame's index * outer: form union of indexes * inner: use intersection of indexes

lsuffix : string

Suffix to use from left frame's overlapping columns

rsuffix : string

Suffix to use from right frame's overlapping columns

Returns **joined** : Panel

pandas.Panel.update

Panel.**update**(*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

Parameters **other** : Panel, or object coercible to Panel

join : How to join individual DataFrames

{'left', 'right', 'outer', 'inner'}, default 'left'

overwrite : boolean, default True

If True then overwrite values for common keys in the calling panel

filter_func : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict : bool

If True, will raise an error if a DataFrame and other both contain data in the same place.

25.5.12 Time series-related

Panel.asfreq(freq[, method, how, normalize])	Convert all TimeSeries inside to specified frequency using DateOffset
Panel.shift(lags[, axis])	Shift major or minor axis by specified number of leads/lags.
Panel.resample(rule[, how, axis, ...])	Convenience method for frequency conversion and resampling of regular time-se
Panel.tz_convert(tz[, axis, copy])	Convert TimeSeries to target time zone. If it is time zone naive, it

Continued on

Table 25.53 – continued from previous page

Panel.tz_localize(tz[, axis, copy])	Localize tz-naive TimeSeries to target time zone

pandas.Panel.asfreq

Panel.**asfreq**(*freq*, *method=None*, *how=None*, *normalize=False*)

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

Parameters **freq** : DateOffset object, or string

method : {'backfill', 'bfill', 'pad', 'ffill', None}

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill methdo

how : {'start', 'end'}, default end

For PeriodIndex only, see PeriodIndex.asfreq

normalize : bool, default False

Whether to reset output index to midnight

Returns **converted** : type of caller

pandas.Panel.shift

Panel.**shift**(*lags*, *axis='major'*)

Shift major or minor axis by specified number of leads/lags. Drops periods right now compared with DataFrame.shift

Parameters **lags** : int

axis : {'major', 'minor'}

Returns **shifted** : Panel

pandas.Panel.resample

Panel.**resample**(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*)

Convenience method for frequency conversion and resampling of regular time-series data.

Parameters **rule** : the offset string or object representing target conversion

how : string, method for down- or re-sampling, default to 'mean' for

downsampling

axis : int, optional, default 0

fill_method : string, fill_method for upsampling, default None

closed : {'right', 'left'}

Which side of bin interval is closed

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

kind: "period"/"timestamp" :

loffset: timedelta :

Adjust the resampled time labels

limit: int, default None :

Maximum size gap to when reindexing with fill_method

base : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

pandas.Panel.tz_convert

`Panel.`**`tz_convert`**(*tz*, *axis=0*, *copy=True*)

Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

pandas.Panel.tz_localize

`Panel.`**`tz_localize`**(*tz*, *axis=0*, *copy=True*)

Localize tz-naive TimeSeries to target time zone

Parameters **tz** : string or pytz.timezone object

copy : boolean, default True

Also make a copy of the underlying data

25.5.13 Serialization / IO / Conversion

`Panel.from_dict`(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects
`Panel.to_pickle`(path)	Pickle (serialize) object to input file path
`Panel.to_excel`(path[, na_rep])	Write each DataFrame in Panel to a separate excel sheet
`Panel.to_sparse`([fill_value, kind])	Convert to SparsePanel
`Panel.to_frame`([filter_observations])	Transform wide format into long (stacked) format as DataFrame
`Panel.to_clipboard`()	Attempt to write text representation of object to the system clipboard ..

pandas.Panel.from_dict

classmethod `Panel.`**`from_dict`**(*data*, *intersect=False*, *orient='items'*, *dtype=None*)

Construct Panel from dict of DataFrame objects

Parameters **data** : dict

{field : DataFrame}

intersect : boolean

Intersect indexes of input DataFrames

orient : {'items', 'minor'}, default 'items'

The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

Returns **Panel** :

pandas.Panel.to_pickle

`Panel.`**`to_pickle`**(*path*)

Pickle (serialize) object to input file path

Parameters **path** : string

File path

pandas.Panel.to_excel

`Panel.`**`to_excel`**(*path*, *na_rep=''*)

Write each DataFrame in Panel to a separate excel sheet

Parameters **excel_writer** : string or ExcelWriter object

File path or existing ExcelWriter

na_rep : string, default ''

Missing data representation

pandas.Panel.to_sparse

`Panel.`**`to_sparse`**(*fill_value=None*, *kind='block'*)

Convert to SparsePanel

Parameters **fill_value** : float, default NaN

kind : {'block', 'integer'}

Returns **y** : SparseDataFrame

pandas.Panel.to_frame

`Panel.`**`to_frame`**(*filter_observations=True*)

Transform wide format into long (stacked) format as DataFrame

Parameters **filter_observations** : boolean, default True

Drop (major, minor) pairs without a complete set of observations across all the items

Returns **y** : DataFrame

pandas.Panel.to_clipboard

Panel.**to_clipboard**()
Attempt to write text representation of object to the system clipboard

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows:
- OS X:

CHAPTER

TWENTYSIX

RELEASE NOTES

This is the list of changes to pandas between each release. For full details, see the commit logs at http://github.com/pydata/pandas

26.1 What is it

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.

26.2 Where to get it

- Source code: http://github.com/pydata/pandas
- Binary installers on PyPI: http://pypi.python.org/pypi/pandas
- Documentation: http://pandas.pydata.org

26.2.1 pandas 0.13

Release date: not-yet-released

New features

Improvements to existing features

API Changes

Experimental Features

Bug Fixes

26.2.2 pandas 0.12

Release date: 2013-07-24

New features

- `pd.read_html()` can now parse HTML strings, files or urls and returns a list of `DataFrame` s courtesy of @cpcloud. (GH3477, GH3605, GH3606)
- Support for reading Amazon S3 files. (GH3504)
- Added module for reading and writing JSON strings/files: pandas.io.json includes `to_json` DataFrame/Series method, and a `read_json` top-level reader various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
- Added module for reading and writing Stata files: pandas.io.stata (GH1512) includes `to_stata` DataFrame method, and a `read_stata` top-level reader
- Added support for writing in `to_csv` and reading in `read_csv`, multi-index columns. The `header` option in `read_csv` now accepts a list of the rows from which to read the index. Added the option, `tupleize_cols` to provide compatiblity for the pre 0.12 behavior of writing and reading multi-index columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a multi-index column. Note: The default value will change in 0.12 to make the default *to* write and read multi-index columns in the new format. (GH3571, GH1651, GH3141)
- Add iterator to `Series.str` (GH3638)
- `pd.set_option()` now allows N option, value pairs (GH3667).
- Added keyword parameters for different types of scatter_matrix subplots
- A `filter` method on grouped Series or DataFrames returns a subset of the original (GH3680, GH919)
- Access to historical Google Finance data in pandas.io.data (GH3814)
- DataFrame plotting methods can sample column colors from a Matplotlib colormap via the `colormap` keyword. (GH3860)

Improvements to existing features

- Fixed various issues with internal pprinting code, the repr() for various objects including TimeStamp and Index now produces valid python code strings and can be used to recreate the object, (GH3038, GH3379, GH3251, GH3460)
- `convert_objects` now accepts a `copy` parameter (defaults to `True`)
- `HDFStore`
  - will retain index attributes (freq,tz,name) on recreation (GH3499,:issue:*4098*)
  - will warn with a `AttributeConflictWarning` if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
  - support datelike columns with a timezone as data_columns (GH2852)
  - table writing performance improvements.
  - support python3 (via `PyTables 3.0.0`) (GH3750)
- Add modulo operator to Series, DataFrame
- Add `date` method to DatetimeIndex
- Add `dropna` argument to pivot_table (:issue: *3820*)
- Simplified the API and added a describe method to Categorical
- `melt` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame (GH3649), thanks @hoechenberger. If `var_name` is not specified and `dataframe.columns.name` is not None, then this will be used as the `var_name` (GH4144). Also support for MultiIndex columns.

- clipboard functions use pyperclip (no dependencies on Windows, alternative dependencies offered for Linux) (GH3837).
- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have have a dtype of `object` (GH1818, GH3572, GH3911, GH3912), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which elimnates any spurious plots from showing up.
- Added Faq section on repr display options, to help users customize their setup.
- `where` operations that result in block splitting are much faster (GH3733)
- Series and DataFrame hist methods now take a `figsize` argument (GH3834)
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations (GH3877)
- Add `unit` keyword to `Timestamp` and `to_datetime` to enable passing of integers or floats that are in an epoch unit of `D, s, ms, us, ns`, thanks @mtkini (GH3969) (e.g. unix timestamps or epoch `s`, with fracional seconds allowed) (GH3540)
- DataFrame corr method (spearman) is now cythonized.
- Improved `network` test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. (GH3910, GH3914)
- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters
- Added `layout` keyword to DataFrame.hist() for more customizable layout (GH4050)
- Timestamp.min and Timestamp.max now represent valid Timestamp instances instead of the default datetime.min and datetime.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

API Changes

- `HDFStore`
  - When removing an object, `remove(key)` raises `KeyError` if the key is not a valid store object.
  - raise a `TypeError` on passing `where` or `columns` to select with a Storer; these are invalid parameters at this time (GH4189)
  - can now specify an `encoding` option to `append/put` to enable alternate encodings (GH3750)
  - enable support for `iterator/chunksize` with `read_hdf`
- The repr() for (Multi)Index now obeys display.max_seq_items rather then numpy threshold print options. (GH3426, GH3466)
- Added mangle_dupe_cols option to read_table/csv, allowing users to control legacy behaviour re dupe cols (A, A.1, A.2 vs A, A) (GH3468) Note: The default value will change in 0.12 to the "no mangle" behaviour, If your code relies on this behaviour, explicitly specify mangle_dupe_cols=True in your calls.
- Do not allow astypes on `datetime64[ns]` except to `object`, and `timedelta64[ns]` to `object/int` (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when perfomed on a `Series` and return an *empty* `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of `slice` objects: - sum, prod, mean, std, var, skew, kurt, corr, and cov

- Do not allow datetimelike/timedeltalike creation except with valid types (e.g. cannot pass `datetime64[ms]`) (GH3423)
- Add `squeeze` keyword to `groupby` to allow reduction from DataFrame -> Series if groups are unique. Regression from 0.10.1, partial revert on (GH2893) with (GH3596)
- Raise on `iloc` when boolean indexing with a label based indexer mask e.g. a boolean Series, even with integer labels, will raise. Since `iloc` is purely positional based, the labels on the Series are not alignable (GH3631)
- The `raise_on_error` option to plotting methods is obviated by GH3572, so it is removed. Plots now always raise when data cannot be plotted or the object being plotted has a dtype of `object`.
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead (GH3582, GH3675, GH3676).
- the `method` and `axis` arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. (GH3907)
- Deprecated display.height, display.width is now only a formatting option does not control triggering of summary, similar to < 0.11.0.
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if `True`, default is `False` (same as prior to 0.12) (GH3679)
- io API changes
  - added `pandas.io.api` for i/o imports
  - removed `Excel` support to `pandas.io.excel`
  - added top-level `pd.read_sql` and `to_sql` DataFrame methods
  - removed `clipboard` support to `pandas.io.clipboard`
  - replace top-level and instance methods `save` and `load` with top-level `read_pickle` and `to_pickle` instance method, `save` and `load` will give deprecation warning.
- the `method` and `axis` arguments of `DataFrame.replace()` are deprecated
- set FutureWarning to require data_source, and to replace year/month with expiry date in pandas.io options. This is in preparation to add options data from google (GH3822)
- the `method` and `axis` arguments of `DataFrame.replace()` are deprecated
- Implement `__nonzero__` for `NDFrame` objects (GH3691, GH3696)
- `as_matrix` with mixed signed and unsigned dtypes will result in 2 x the lcd of the unsigned as an int, maxing with `int64`, to avoid precision issues (GH3733)
- `na_values` in a list provided to `read_csv/read_excel` will match string and numeric versions e.g. `na_values=['99']` will match 99 whether the column ends up being int, float, or string (GH3611)
- `read_html` now defaults to `None` when reading, and falls back on `bs4` + `html5lib` when lxml fails to parse. a list of parsers to try until success is also valid
- more consistency in the to_datetime return types (give string/array of string inputs) (GH3888)
- The internal `pandas` class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)

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

Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. (GH2301)

Bug Fixes

- Fixed an esoteric excel reading bug, xlrd>= 0.9.0 now required for excel support. Should provide python3 support (for reading) which has been lacking. (GH3164)
- Disallow Series constructor called with MultiIndex which caused segfault (GH4187)
- Allow unioning of date ranges sharing a timezone (GH3491)
- Fix to_csv issue when having a large number of rows and `NaT` in some columns (GH3437)
- `.loc` was not raising when passed an integer list (GH3449)
- Unordered time series selection was misbehaving when using label slicing (GH3448)
- Fix sorting in a frame with a list of columns which contains datetime64[ns] dtypes (GH3461)
- DataFrames fetched via FRED now handle '.' as a NaN. (GH3469)
- Fix regression in a DataFrame apply with axis=1, objects were not being converted back to base dtypes correctly (GH3480)
- Fix issue when storing uint dtypes in an HDFStore. (GH3493)
- Non-unique index support clarified (GH3468)
  - Addressed handling of dupe columns in df.to_csv new and old (GH3454, GH3457)
  - Fix assigning a new index to a duplicate index in a DataFrame would fail (GH3468)
  - Fix construction of a DataFrame with a duplicate index
  - ref_locs support to allow duplicative indices across dtypes, allows iget support to always find the index (even across dtypes) (GH2194)
  - applymap on a DataFrame with a non-unique index now works (removed warning) (GH2786), and fix (GH3230)
  - Fix to_csv to handle non-unique columns (GH3495)
  - Duplicate indexes with getitem will return items in the correct order (GH3455, GH3457) and handle missing elements like unique indices (GH3561)
  - Duplicate indexes with and empty DataFrame.from_records will return a correct frame (GH3562)
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed (GH3602)
  - Non-unique indexing with a slice via `loc` and friends fixed (GH3659)
  - Allow insert/delete to non-unique columns (GH3679)
  - Extend `reindex` to correctly deal with non-unique indices (GH3679)
  - `DataFrame.itertuples()` now works with frames with duplicate column names (GH3873)
  - Bug in non-unique indexing via `iloc` (GH4017); added `takeable` argument to `reindex` for location-based taking
  - Allow non-unique indexing in series via `.ix/.loc` and `__getitem__` (GH4246)

  - Fixed non-unique indexing memory allocation issue with `.ix/.loc` (GH4280)
- Fixed bug in groupby with empty series referencing a variable before assignment. (GH3510)
- Allow index name to be used in groupby for non MultiIndex (GH4014)
- Fixed bug in mixed-frame assignment with aligned series (GH3492)
- Fixed bug in selecting month/quarter/year from a series would not select the time element on the last day (GH3546)
- Fixed a couple of MultiIndex rendering bugs in df.to_html() (GH3547, GH3553)
- Properly convert np.datetime64 objects in a Series (GH3416)
- Raise a `TypeError` on invalid datetime/timedelta operations e.g. add datetimes, multiple timedelta x datetime
- Fix `.diff` on datelike and timedelta operations (GH3100)
- `combine_first` not returning the same dtype in cases where it can (GH3552)
- Fixed bug with `Panel.transpose` argument aliases (GH3556)
- Fixed platform bug in `PeriodIndex.take` (GH3579)
- Fixed bud in incorrect conversion of datetime64[ns] in `combine_first` (GH3593)
- Fixed bug in reset_index with `NaN` in a multi-index (GH3586)
- `fillna` methods now raise a `TypeError` when the `value` parameter is a `list` or `tuple`.
- Fixed bug where a time-series was being selected in preference to an actual column name in a frame (GH3594)
- Make secondary_y work properly for bar plots (GH3598)
- Fix modulo and integer division on Series,DataFrames to act similary to `float` dtypes to return `np.nan` or `np.inf` as appropriate (GH3590)
- Fix incorrect dtype on groupby with `as_index=False` (GH3610)
- Fix `read_csv/read_excel` to correctly encode identical na_values, e.g. `na_values=[-999.0,-999]` was failing (GH3611)
- Disable HTML output in qtconsole again. (GH3657)
- Reworked the new repr display logic, which users found confusing. (GH3663)
- Fix indexing issue in ndim >= 3 with `iloc` (GH3617)
- Correctly parse date columns with embedded (nan/NaT) into datetime64[ns] dtype in `read_csv` when `parse_dates` is specified (GH3062)
- Fix not consolidating before to_csv (GH3624)
- Fix alignment issue when setitem in a DataFrame with a piece of a DataFrame (GH3626) or a mixed DataFrame and a Series (GH3668)
- Fix plotting of unordered DatetimeIndex (GH3601)
- `sql.write_frame` failing when writing a single column to sqlite (GH3628), thanks to @stonebig
- Fix pivoting with `nan` in the index (GH3558)
- Fix running of bs4 tests when it is not installed (GH3605)
- Fix parsing of html table (GH3606)
- `read_html()` now only allows a single backend: `html5lib` (GH3616)

- `convert_objects` with `convert_dates='coerce'` was parsing some single-letter strings into today's date
- `DataFrame.from_records` did not accept empty recarrays (GH3682)
- `DataFrame.to_csv` will succeed with the deprecated option `nanRep`, @tdsmith
- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Fix file tokenization error with r delimiter and quoted fields (GH3453)
- Groupby transform with item-by-item not upcasting correctly (GH3740)
- Incorrectly read a HDFStore multi-index Frame witha column specification (GH3748)
- `read_html` now correctly skips tests (GH3741)
- PandasObjects raise TypeError when trying to hash (GH3882)
- Fix incorrect arguments passed to concat that are not list-like (e.g. concat(df1,df2)) (GH3481)
- Correctly parse when passed the `dtype=str` (or other variable-len string dtypes) in `read_csv` (GH3795)
- Fix index name not propogating when using `loc/ix` (GH3880)
- Fix groupby when applying a custom function resulting in a returned DataFrame was not converting dtypes (GH3911)
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn't working (GH3907)
- Fixed `__truediv__` in Python 2.7 with `numexpr` installed to actually do true division when dividing two integer arrays with at least 10000 cells total (GH3764)
- Indexing with a string with seconds resolution not selecting from a time index (GH3925)
- csv parsers would loop infinitely if `iterator=True` but no `chunksize` was specified (GH3967), python parser failing with `chunksize=1`
- Fix index name not propogating when using `shift`
- Fixed dropna=False being ignored with multi-index stack (GH3997)
- Fixed flattening of columns when renaming MultiIndex columns DataFrame (GH4004)
- Fix `Series.clip` for datetime series. NA/NaN threshold values will now throw ValueError (GH3996)
- Fixed insertion issue into DataFrame, after rename (GH4032)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fix bug where `HDFStore` will fail to append because of a different block ordering on-disk (GH4096)
- Better error messages on inserting incompatible columns to a frame (GH4107)

- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when regex=False (GH4115)
- Fixed bug in `convert_objects(convert_numeric=True)` where a mixed numeric and object Series/Frame was not converting properly (GH4119)
- Fixed bugs in multi-index selection with column multi-index and duplicates (GH4145, GH4146)
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` (GH4152)
- Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
- Fixed bug in `Series.where` where broadcasting a single element input vector to the length of the series resulted in multiplying the value inside the input (GH4192)
- Fixed bug in plotting that wasn't raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
- Fixed bug where Index slices weren't carrying the name attribute (GH4226)
- Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
- Fixed bug where html5lib wasn't being properly skipped (GH4265)
- Fixed bug where get_data_famafrench wasn't using the correct file edges (GH4281)

26.2.3 pandas 0.11.0

Release date: 2013-04-22

New features

- New documentation section, `10 Minutes to Pandas`
- New documentation section, `Cookbook`
- Allow mixed dtypes (e.g `float32/float64/int32/int16/int8`) to coexist in DataFrames and propogate in operations
- Add function to pandas.io.data for retrieving stock index components from Yahoo! finance (GH2795)
- Support slicing with time objects (GH2681)
- Added `.iloc` attribute, to support strict integer based indexing, analogous to `.ix` (GH2922)
- Added `.loc` attribute, to support strict label based indexing, analagous to `.ix` (GH3053)
- Added `.iat` attribute, to support fast scalar access via integers (replaces `iget_value/iset_value`)
- Added `.at` attribute, to support fast scalar access via labels (replaces `get_value/set_value`)
- Moved functionaility from `irow,icol,iget_value/iset_value` to `.iloc` indexer (via `_ixs` methods in each object)
- Added support for expression evaluation using the `numexpr` library
- Added `convert=boolean` to `take` routines to translate negative indices to positive, defaults to True
- Added to_series() method to indices, to facilitate the creation of indexeres (GH3275)

Improvements to existing features

- Improved performance of df.to_csv() by up to 10x in some cases. (GH3059)
- added `blocks` attribute to DataFrames, to return a dict of dtypes to homogeneously dtyped DataFrames

- added keyword `convert_numeric` to `convert_objects()` to try to convert object dtypes to numeric types (default is False)
- `convert_dates` in `convert_objects` can now be `coerce` which will return a datetime64[ns] dtype with non-convertibles set as `NaT`; will preserve an all-nan object (e.g. strings), default is True (to perform soft-conversion
- Series print output now includes the dtype by default
- Optimize internal reindexing routines (GH2819, GH2867)
- `describe_option()` now reports the default and current value of options.
- Add `format` option to `pandas.to_datetime` with faster conversion of strings that can be parsed with datetime.strptime
- Add `axes` property to `Series` for compatibility
- Add `xs` function to `Series` for compatibility
- Allow setitem in a frame where only mixed numerics are present (e.g. int and float), (GH3037)
- `HDFStore`
  - Provide dotted attribute access to `get` from stores (e.g. store.df == store['df'])
  - New keywords `iterator=boolean`, and `chunksize=number_in_a_chunk` are provided to support iteration on `select` and `select_as_multiple` (GH3076)
  - support `read_hdf/to_hdf` API similar to `read_csv/to_csv` (GH3222)
- Add `squeeze` method to possibly remove length 1 dimensions from an object.

```
In [1]: p = Panel(randn(3,4,4),items=['ItemA','ItemB','ItemC'],
   ...:                    major_axis=date_range('20010102',periods=4),
   ...:                    minor_axis=['A','B','C','D'])
   ...:

In [2]: p

<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D

In [3]: p.reindex(items=['ItemA']).squeeze()

                   A         B         C         D
2001-01-02  0.408204 -1.048089 -0.025747 -0.988387
2001-01-03  0.094055  1.262731  1.289997  0.082423
2001-01-04 -0.055758  0.536580 -0.489682  0.369374
2001-01-05 -0.034571 -2.484478 -0.281461  0.030711

In [4]: p.reindex(items=['ItemA'],minor=['B']).squeeze()

2001-01-02   -1.048089
2001-01-03    1.262731
2001-01-04    0.536580
2001-01-05   -2.484478
Freq: D, Name: B, dtype: float64
```

- Improvement to Yahoo API access in `pd.io.data.Options` (GH2758)

- added option *display.max_seq_items* to control the number of elements printed per sequence pprinting it. (GH2979)
- added option *display.chop_threshold* to control display of small numerical values. (GH2739)
- added option *display.max_info_rows* to prevent verbose_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- value_counts() now accepts a "normalize" argument, for normalized histograms. (GH2710).
- DataFrame.from_records now accepts not only dicts but any instance of the collections.Mapping ABC.
- Allow selection semantics via a string with a datelike index to work in both Series and DataFrames (GH3070)

```
In [5]: idx = date_range("2001-10-1", periods=5, freq='M')

In [6]: ts = Series(np.random.rand(len(idx)),index=idx)

In [7]: ts['2001']

2001-10-31    0.751953
2001-11-30    0.561512
2001-12-31    0.572214
Freq: M, dtype: float64

In [8]: df = DataFrame(dict(A = ts))

In [9]: df['2001']

                   A
2001-10-31  0.751953
2001-11-30  0.561512
2001-12-31  0.572214
```

- added option *display.mpl_style* providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).
- Improved performance across several core functions by taking memory ordering of arrays into account. Courtesy of @stephenwlin (GH3130)
- Improved performance of groupby transform method (GH2121)
- Handle "ragged" CSV files missing trailing delimiters in rows with missing fields when also providing explicit list of column names (so the parser knows how many columns to expect in the result) (GH2981)
- On a mixed DataFrame, allow setting with indexers with ndarray/DataFrame on rhs (GH3216)
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- Add `time` method to DatetimeIndex (GH3180)
- Return NA when using Series.str[...] for values that are not long enough (GH3223)
- Display cursor coordinate information in time-series plots (GH1670)
- to_html() now accepts an optional "escape" argument to control reserved HTML character escaping (enabled by default) and escapes `&`, in addition to < and >. (GH2919)

API Changes

- Do not automatically upcast numeric specified dtypes to `int64` or `float64` (GH622 and GH797)
- DataFrame construction of lists and scalars, with no dtype present, will result in casting to `int64` or `float64`, regardless of platform. This is not an apparent change in the API, but noting it.

- Guarantee that `convert_objects()` for Series/DataFrame always returns a copy
- groupby operations will respect dtypes for numeric float operations (float32/float64); other types will be operated on, and will try to cast back to the input dtype (e.g. if an int is passed, as long as the output doesn't have nans, then an int will be returned)
- backfill/pad/take/diff/ohlc will now support `float32/int16/int8` operations
- Block types will upcast as needed in where/masking operations (GH2793)
- Series now automatically will try to set the correct dtype based on passed datetimelike objects (datetime/Timestamp)
  - timedelta64 are returned in appropriate cases (e.g. Series - Series, when both are datetime64)
  - mixed datetimes and objects (GH2751) in a constructor will be cast correctly
  - astype on datetimes to object are now handled (as well as NaT conversions to np.nan)
  - all timedelta like objects will be correctly assigned to `timedelta64` with mixed `NaN` and/or `NaT` allowed
- arguments to DataFrame.clip were inconsistent to numpy and Series clipping (GH2747)
- util.testing.assert_frame_equal now checks the column and index names (GH2964)
- Constructors will now return a more informative ValueError on failures when invalid shapes are passed
- Don't suppress TypeError in GroupBy.agg (GH3238)
- Methods return None when inplace=True (GH1893)
- `HDFStore`
  - added the method `select_column` to select a single column from a table as a Series.
  - deprecated the `unique` method, can be replicated by `select_column(key,column).unique()`
  - `min_itemsize` parameter will now automatically create data_columns for passed keys
- Downcast on pivot if possible (GH3283), adds argument `downcast` to `fillna`
- Introduced options *display.height/width* for explicitly specifying terminal height/width in characters. Deprecated display.line_width, now replaced by display.width. These defaults are in effect for scripts as well, so unless disabled, previously very wide output will now be output as "expand_repr" style wrapped output.
- Various defaults for options (including display.max_rows) have been revised, after a brief survey concluded they were wrong for everyone. Now at w=80,h=60.
- HTML repr output in IPython qtconsole is once again controlled by the option *display.notebook_repr_html*, and on by default.

Bug Fixes

- Fix seg fault on empty data frame when fillna with `pad` or `backfill` (GH2778)
- Single element ndarrays of datetimelike objects are handled (e.g. np.array(datetime(2001,1,1,0,0))), w/o dtype being passed
- 0-dim ndarrays with a passed dtype are handled correctly (e.g. np.array(0.,dtype='float32'))
- Fix some boolean indexing inconsistencies in Series.__getitem__/__setitem__ (GH2776)
- Fix issues with DataFrame and Series constructor with integers that overflow `int64` and some mixed typed type lists (GH2845)
- `HDFStore`

  - Fix weird PyTables error when using too many selectors in a where also correctly filter on any number of values in a Term expression (so not using numexpr filtering, but isin filtering)
  - Internally, change all variables to be private-like (now have leading underscore)
  - Fixes for query parsing to correctly interpret boolean and != (GH2849, GH2973)
  - Fixes for pathological case on SparseSeries with 0-len array and compression (GH2931)
  - Fixes bug with writing rows if part of a block was all-nan (GH3012)
  - Exceptions are now ValueError or TypeError as needed
  - A table will now raise if min_itemsize contains fields which are not queryables
- Bug showing up in applymap where some object type columns are converted (GH2909) had an incorrect default in convert_objects
- TimeDeltas
  - Series ops with a Timestamp on the rhs was throwing an exception (GH2898) added tests for Series ops with datetimes,timedeltas,Timestamps, and datelike Series on both lhs and rhs
  - Fixed subtle timedelta64 inference issue on py3 & numpy 1.7.0 (GH3094)
  - Fixed some formatting issues on timedelta when negative
  - Support null checking on timedelta64, representing (and formatting) with NaT
  - Support setitem with np.nan value, converts to NaT
  - Support min/max ops in a Dataframe (abs not working, nor do we error on non-supported ops)
  - Support idxmin/idxmax/abs/max/min in a Series (GH2989, GH2982)
- Bug on in-place putmasking on an `integer` series that needs to be converted to `float` (GH2746)
- Bug in argsort of `datetime64[ns]` Series with `NaT` (GH2967)
- Bug in value_counts of `datetime64[ns]` Series (GH3002)
- Fixed printing of `NaT` in an index
- Bug in idxmin/idxmax of `datetime64[ns]` Series with `NaT` (GH2982)
- Bug in `icol, take` with negative indicies was producing incorrect return values (see GH2922, GH2892), also check for out-of-bounds indices (GH3029)
- Bug in DataFrame column insertion when the column creation fails, existing frame is left in an irrecoverable state (GH3010)
- Bug in DataFrame update, combine_first where non-specified values could cause dtype changes (GH3016, GH3041)
- Bug in groupby with first/last where dtypes could change (GH3041, GH2763)
- Formatting of an index that has `nan` was inconsistent or wrong (would fill from other values), (GH2850)
- Unstack of a frame with no nans would always cause dtype upcasting (GH2929)
- Fix scalar datetime.datetime parsing bug in read_csv (GH3071)
- Fixed slow printing of large Dataframes, due to inefficient dtype reporting (GH2807)
- Fixed a segfault when using a function as grouper in groupby (GH3035)
- Fix pretty-printing of infinite data structures (closes GH2978)
- Fixed exception when plotting timeseries bearing a timezone (closes GH2877)

- str.contains ignored na argument (GH2806)
- Substitute warning for segfault when grouping with categorical grouper of mismatched length (GH3011)
- Fix exception in SparseSeries.density (GH2083)
- Fix upsampling bug with closed='left' and daily to daily data (GH3020)
- Fixed missing tick bars on scatter_matrix plot (GH3063)
- Fixed bug in Timestamp(d,tz=foo) when d is date() rather then datetime() (GH2993)
- series.plot(kind='bar') now respects pylab color schem (GH3115)
- Fixed bug in reshape if not passed correct input, now raises TypeError (GH2719)
- Fixed a bug where Series ctor did not respect ordering if OrderedDict passed in (GH3282)
- Fix NameError issue on RESO_US (GH2787)
- Allow selection in an *unordered* timeseries to work similary to an *ordered* timeseries (GH2437).
- Fix implemented `.xs` when called with `axes=1` and a level parameter (GH2903)
- Timestamp now supports the class method fromordinal similar to datetimes (GH3042)
- Fix issue with indexing a series with a boolean key and specifiying a 1-len list on the rhs (GH2745) or a list on the rhs (GH3235)
- Fixed bug in groupby apply when kernel generate list of arrays having unequal len (GH1738)
- fixed handling of rolling_corr with center=True which could produce corr>1 (GH3155)
- Fixed issues where indices can be passed as 'index/column' in addition to 0/1 for the axis parameter
- PeriodIndex.tolist now boxes to Period (GH3178)
- PeriodIndex.get_loc KeyError now reports Period instead of ordinal (GH3179)
- df.to_records bug when handling MultiIndex (GH3189)
- Fix Series.__getitem__ segfault when index less than -length (GH3168)
- Fix bug when using Timestamp as a date parser (GH2932)
- Fix bug creating date range from Timestamp with time zone and passing same time zone (GH2926)
- Add comparison operators to Period object (GH2781)
- Fix bug when concatenating two Series into a DataFrame when they have the same name (GH2797)
- Fix automatic color cycling when plotting consecutive timeseries without color arguments (GH2816)
- fixed bug in the pickling of PeriodIndex (GH2891)
- Upcast/split blocks when needed in a mixed DataFrame when setitem with an indexer (GH3216)
- Invoking df.applymap on a dataframe with dupe cols now raises a ValueError (GH2786)
- Apply with invalid returned indices raise correct Exception (GH2808)
- Fixed a bug in plotting log-scale bar plots (GH3247)
- df.plot() grid on/off now obeys the mpl default style, just like series.plot(). (GH3233)
- Fixed a bug in the legend of plotting.andrews_curves() (GH3278)
- Produce a series on apply if we only generate a singular series and have a simple index (GH2893)
- Fix Python ascii file parsing when integer falls outside of floating point spacing (GH3258)

- fixed pretty priniting of sets (GH3294)
- Panel() and Panel.from_dict() now respects ordering when give OrderedDict (GH3303)
- DataFrame where with a datetimelike incorrectly selecting (GH3311)
- Ensure index casts work even in Int64Index
- Fix set_index segfault when passing MultiIndex (GH3308)
- Ensure pickles created in py2 can be read in py3
- Insert ellipsis in MultiIndex summary repr (GH3348)
- Groupby will handle mutation among an input groups columns (and fallback to non-fast apply) (GH3380)
- Eliminated unicode errors on FreeBSD when using MPL GTK backend (GH3360)
- Period.strftime should return unicode strings always (GH3363)
- Respect passed read_* chunksize in get_chunk function (GH3406)

26.2.4 pandas 0.10.1

Release date: 2013-01-22

New features

- Add data inferface to World Bank WDI pandas.io.wb (GH2592)

API Changes

- Restored inplace=True behavior returning self (same object) with deprecation warning until 0.11 (GH1893)
- `HDFStore`
  - refactored HFDStore to deal with non-table stores as objects, will allow future enhancements
  - removed keyword `compression` from `put` (replaced by keyword `complib` to be consistent across library)
  - warn *PerformanceWarning* if you are attempting to store types that will be pickled by PyTables

Improvements to existing features

- `HDFStore`
  - enables storing of multi-index dataframes (closes GH1277)
  - support data column indexing and selection, via `data_columns` keyword in append
  - support write chunking to reduce memory footprint, via `chunksize` keyword to append
  - support automagic indexing via `index` keyword to append
  - support `expectedrows` keyword in append to inform `PyTables` about the expected tablesize
  - support `start` and `stop` keywords in select to limit the row selection space
  - added `get_store` context manager to automatically import with pandas
  - added column filtering via `columns` keyword in select
  - added methods append_to_multiple/select_as_multiple/select_as_coordinates to do multiple-table append/selection
  - added support for datetime64 in columns
  - added method `unique` to select the unique values in an indexable or data column

  - added method `copy` to copy an existing store (and possibly upgrade)
  - show the shape of the data on disk for non-table stores when printing the store
  - added ability to read PyTables flavor tables (allows compatiblity to other HDF5 systems)
- Add `logx` option to DataFrame/Series.plot (GH2327, GH2565)
- Support reading gzipped data from file-like object
- `pivot_table` aggfunc can be anything used in GroupBy.aggregate (GH2643)
- Implement DataFrame merges in case where set cardinalities might overflow 64-bit integer (GH2690)
- Raise exception in C file parser if integer dtype specified and have NA values. (GH2631)
- Attempt to parse ISO8601 format dates when parse_dates=True in read_csv for major performance boost in such cases (GH2698)
- Add methods `neg` and `inv` to Series
- Implement `kind` option in `ExcelFile` to indicate whether it's an XLS or XLSX file (GH2613)

Bug fixes

- Fix read_csv/read_table multithreading issues (GH2608)
- `HDFStore`
  - correctly handle `nan` elements in string columns; serialize via the `nan_rep` keyword to append
  - raise correctly on non-implemented column types (unicode/date)
  - handle correctly `Term` passed types (e.g. `index<1000`, when index is `Int64`), (closes GH512)
  - handle Timestamp correctly in data_columns (closes GH2637)
  - contains correctly matches on non-natural names
  - correctly store `float32` dtypes in tables (if not other float types in the same table)
- Fix DataFrame.info bug with UTF8-encoded columns. (GH2576)
- Fix DatetimeIndex handling of FixedOffset tz (GH2604)
- More robust detection of being in IPython session for wide DataFrame console formatting (GH2585)
- Fix platform issues with `file:///` in unit test (GH2564)
- Fix bug and possible segfault when grouping by hierarchical level that contains NA values (GH2616)
- Ensure that MultiIndex tuples can be constructed with NAs (GH2616)
- Fix int64 overflow issue when unstacking MultiIndex with many levels (GH2616)
- Exclude non-numeric data from DataFrame.quantile by default (GH2625)
- Fix a Cython C int64 boxing issue causing read_csv to return incorrect results (GH2599)
- Fix groupby summing performance issue on boolean data (GH2692)
- Don't bork Series containing datetime64 values with to_datetime (GH2699)
- Fix DataFrame.from_records corner case when passed columns, index column, but empty record list (GH2633)
- Fix C parser-tokenizer bug with trailing fields. (GH2668)
- Don't exclude non-numeric data from GroupBy.max/min (GH2700)
- Don't lose time zone when calling DatetimeIndex.drop (GH2621)

- Fix setitem on a Series with a boolean key and a non-scalar as value (GH2686)
- Box datetime64 values in Series.apply/map (GH2627, GH2689)
- Upconvert datetime + datetime64 values when concatenating frames (GH2624)
- Raise a more helpful error message in merge operations when one DataFrame has duplicate columns (GH2649)
- Fix partial date parsing issue occuring only when code is run at EOM (GH2618)
- Prevent MemoryError when using counting sort in sortlevel with high-cardinality MultiIndex objects (GH2684)
- Fix Period resampling bug when all values fall into a single bin (GH2070)
- Fix buggy interaction with usecols argument in read_csv when there is an implicit first index column (GH2654)

26.2.5 pandas 0.10.0

Release date: 2012-12-17

New features

- Brand new high-performance delimited file parsing engine written in C and Cython. 50% or better performance in many standard use cases with a fraction as much memory usage. (GH407, GH821)
- Many new file parser (read_csv, read_table) features:
  - Support for on-the-fly gzip or bz2 decompression (*compression* option)
  - Ability to get back numpy.recarray instead of DataFrame (*as_recarray=True*)
  - *dtype* option: explicit column dtypes
  - *usecols* option: specify list of columns to be read from a file. Good for reading very wide files with many irrelevant columns (GH1216 GH926, GH2465)
  - Enhanced unicode decoding support via *encoding* option
  - *skipinitialspace* dialect option
  - Can specify strings to be recognized as True (*true_values*) or False (*false_values*)
  - High-performance *delim_whitespace* option for whitespace-delimited files; a preferred alternative to the '\s+' regular expression delimiter
  - Option to skip "bad" lines (wrong number of fields) that would otherwise have caused an error in the past (*error_bad_lines* and *warn_bad_lines* options)
  - Substantially improved performance in the parsing of integers with thousands markers and lines with comments
  - Easy of European (and other) decimal formats (*decimal* option) (GH584, GH2466)
  - Custom line terminators (e.g. lineterminator='~') (GH2457)
  - Handling of no trailing commas in CSV files (GH2333)
  - Ability to handle fractional seconds in date_converters (GH2209)
  - read_csv allow scalar arg to na_values (GH1944)
  - Explicit column dtype specification in read_* functions (GH1858)
  - Easier CSV dialect specification (GH1743)
  - Improve parser performance when handling special characters (GH1204)
- Google Analytics API integration with easy oauth2 workflow (GH2283)

- Add error handling to Series.str.encode/decode (GH2276)
- Add `where` and `mask` to Series (GH2337)
- Grouped histogram via *by* keyword in Series/DataFrame.hist (GH2186)
- Support optional `min_periods` keyword in `corr` and `cov` for both Series and DataFrame (GH2002)
- Add `duplicated` and `drop_duplicates` functions to Series (GH1923)
- Add docs for `HDFStore table` format
- 'density' property in *SparseSeries* (GH2384)
- Add `ffill` and `bfill` convenience functions for forward- and backfilling time series data (GH2284)
- New option configuration system and functions *set_option*, *get_option*, *describe_option*, and *reset_option*. Deprecate *set_printoptions* and *reset_printoptions* (GH2393). You can also access options as attributes via `pandas.options.X`
- Wide DataFrames can be viewed more easily in the console with new *expand_frame_repr* and *line_width* configuration options. This is on by default now (GH2436)
- Scikits.timeseries-like moving window functions via `rolling_window` (GH1270)

Experimental Features

- Add support for Panel4D, a named 4 Dimensional stucture
- Add support for ndpanel factory functions, to create custom, domain-specific N-Dimensional containers

API Changes

- The default binning/labeling behavior for `resample` has been changed to *closed='left', label='left'* for daily and lower frequencies. This had been a large source of confusion for users. See "what's new" page for more on this. (GH2410)
- Methods with `inplace` option now return None instead of the calling (modified) object (GH1893)
- The special case DataFrame - TimeSeries doing column-by-column broadcasting has been deprecated. Users should explicitly do e.g. df.sub(ts, axis=0) instead. This is a legacy hack and can lead to subtle bugs.
- inf/-inf are no longer considered as NA by isnull/notnull. To be clear, this is legacy cruft from early pandas. This behavior can be globally re-enabled using the new option `mode.use_inf_as_null` (GH2050, GH1919)
- `pandas.merge` will now default to `sort=False`. For many use cases sorting the join keys is not necessary, and doing it by default is wasteful
- Specify `header=0` explicitly to replace existing column names in file in read_* functions.
- Default column names for header-less parsed files (yielded by read_csv, etc.) are now the integers 0, 1, ... A new argument *prefix* has been added; to get the v0.9.x behavior specify `prefix='X'` (GH2034). This API change was made to make the default column names more consistent with the DataFrame constructor's default column names when none are specified.
- DataFrame selection using a boolean frame now preserves input shape
- If function passed to Series.apply yields a Series, result will be a DataFrame (GH2316)
- Values like YES/NO/yes/no will not be considered as boolean by default any longer in the file parsers. This can be customized using the new `true_values` and `false_values` options (GH2360)
- *obj.fillna()* is no longer valid; make *method='pad'* no longer the default option, to be more explicit about what kind of filling to perform. Add *ffill/bfill* convenience functions per above (GH2284)
- *HDFStore.keys()* now returns an absolute path-name for each key

- *to_string()* now always returns a unicode string. (GH2224)
- File parsers will not handle NA sentinel values arising from passed converter functions

Improvements to existing features

- Add `nrows` option to DataFrame.from_records for iterators (GH1794)
- Unstack/reshape algorithm rewrite to avoid high memory use in cases where the number of observed key-tuples is much smaller than the total possible number that could occur (GH2278). Also improves performance in most cases.
- Support duplicate columns in DataFrame.from_records (GH2179)
- Add `normalize` option to Series/DataFrame.asfreq (GH2137)
- SparseSeries and SparseDataFrame construction from empty and scalar values now no longer create dense ndarrays unnecessarily (GH2322)
- `HDFStore` now supports hierarchial keys (GH2397)
- Support multiple query selection formats for `HDFStore tables` (GH1996)
- Support `del store['df']` syntax to delete HDFStores
- Add multi-dtype support for `HDFStore tables`
- `min_itemsize` parameter can be specified in `HDFStore table` creation
- Indexing support in `HDFStore tables` (GH698)
- Add *line_terminator* option to DataFrame.to_csv (GH2383)
- added implementation of str(x)/unicode(x)/bytes(x) to major pandas data structures, which should do the right thing on both py2.x and py3.x. (GH2224)
- Reduce groupby.apply overhead substantially by low-level manipulation of internal NumPy arrays in DataFrames (GH535)
- Implement `value_vars` in `melt` and add `melt` to pandas namespace (GH2412)
- Added boolean comparison operators to Panel
- Enable `Series.str.strip/lstrip/rstrip` methods to take an argument (GH2411)
- The DataFrame ctor now respects column ordering when given an OrderedDict (GH2455)
- Assigning DatetimeIndex to Series changes the class to TimeSeries (GH2139)
- Improve performance of .value_counts method on non-integer data (GH2480)
- `get_level_values` method for MultiIndex return Index instead of ndarray (GH2449)
- `convert_to_r_dataframe` conversion for datetime values (GH2351)
- Allow `DataFrame.to_csv` to represent inf and nan differently (GH2026)
- Add `min_i` argument to `nancorr` to specify minimum required observations (GH2002)
- Add `inplace` option to `sortlevel` / `sort` functions on DataFrame (GH1873)
- Enable DataFrame to accept scalar constructor values like Series (GH1856)
- DataFrame.from_records now takes optional `size` parameter (GH1794)
- include iris dataset (GH1709)
- No datetime64 DataFrame column conversion of datetime.datetime with tzinfo (GH1581)
- Micro-optimizations in DataFrame for tracking state of internal consolidation (GH217)

- Format parameter in DataFrame.to_csv (GH1525)
- Partial string slicing for `DatetimeIndex` for daily and higher frequencies (GH2306)
- Implement `col_space` parameter in `to_html` and `to_string` in DataFrame (GH1000)
- Override `Series.tolist` and box datetime64 types (GH2447)
- Optimize `unstack` memory usage by compressing indices (GH2278)
- Fix HTML repr in IPython qtconsole if opening window is small (GH2275)
- Escape more special characters in console output (GH2492)
- df.select now invokes bool on the result of crit(x) (GH2487)

Bug fixes

- Fix major performance regression in DataFrame.iteritems (GH2273)
- Fixes bug when negative period passed to Series/DataFrame.diff (GH2266)
- Escape tabs in console output to avoid alignment issues (GH2038)
- Properly box datetime64 values when retrieving cross-section from mixed-dtype DataFrame (GH2272)
- Fix concatenation bug leading to GH2057, GH2257
- Fix regression in Index console formatting (GH2319)
- Box Period data when assigning PeriodIndex to frame column (GH2243, GH2281)
- Raise exception on calling reset_index on Series with inplace=True (GH2277)
- Enable setting multiple columns in DataFrame with hierarchical columns (GH2295)
- Respect dtype=object in DataFrame constructor (GH2291)
- Fix DatetimeIndex.join bug with tz-aware indexes and how='outer' (GH2317)
- pop(...) and del works with DataFrame with duplicate columns (GH2349)
- Treat empty strings as NA in date parsing (rather than let dateutil do something weird) (GH2263)
- Prevent uint64 -> int64 overflows (GH2355)
- Enable joins between MultiIndex and regular Index (GH2024)
- Fix time zone metadata issue when unioning non-overlapping DatetimeIndex objects (GH2367)
- Raise/handle int64 overflows in parsers (GH2247)
- Deleting of consecutive rows in `HDFStore tables` is much faster than before
- Appending on a HDFStore would fail if the table was not first created via `put`
- Use *col_space* argument as minimum column width in DataFrame.to_html (GH2328)
- Fix tz-aware DatetimeIndex.to_period (GH2232)
- Fix DataFrame row indexing case with MultiIndex (GH2314)
- Fix to_excel exporting issues with Timestamp objects in index (GH2294)
- Fixes assigning scalars and array to hierarchical column chunk (GH1803)
- Fixed a UnicdeDecodeError with series tidy_repr (GH2225)
- Fixed issued with duplicate keys in an index (GH2347, GH2380)
- Fixed issues re: Hash randomization, default on starting w/ py3.3 (GH2331)

- Fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- Fix Timestamp formatting with tzoffset time zone in dateutil 2.1 (GH2443)
- Fix GroupBy.apply issue when using BinGrouper to do ts binning (GH2300)
- Fix issues resulting from datetime.datetime columns being converted to datetime64 when calling DataFrame.apply. (GH2374)
- Raise exception when calling to_panel on non uniquely-indexed frame (GH2441)
- Improved detection of console encoding on IPython zmq frontends (GH2458)
- Preserve time zone when .append-ing two time series (GH2260)
- Box timestamps when calling reset_index on time-zone-aware index rather than creating a tz-less datetime64 column (GH2262)
- Enable searching non-string columns in DataFrame.filter(like=...) (GH2467)
- Fixed issue with losing nanosecond precision upon conversion to DatetimeIndex(GH2252)
- Handle timezones in Datetime.normalize (GH2338)
- Fix test case where dtype specification with endianness causes failures on big endian machines (GH2318)
- Fix plotting bug where upsampling causes data to appear shifted in time (GH2448)
- Fix `read_csv` failure for UTF-16 with BOM and skiprows(GH2298)
- read_csv with names arg not implicitly setting header=None(GH2459)
- Unrecognized compression mode causes segfault in read_csv(GH2474)
- In read_csv, header=0 and passed names should discard first row(GH2269)
- Correctly route to stdout/stderr in read_table (GH2071)
- Fix exception when Timestamp.to_datetime is called on a Timestamp with tzoffset (GH2471)
- Fixed unintentional conversion of datetime64 to long in groupby.first() (GH2133)
- Union of empty DataFrames now return empty with concatenated index (GH2307)
- DataFrame.sort_index raises more helpful exception if sorting by column with duplicates (GH2488)
- DataFrame.to_string formatters can be list, too (GH2520)
- DataFrame.combine_first will always result in the union of the index and columns, even if one DataFrame is length-zero (GH2525)
- Fix several DataFrame.icol/irow with duplicate indices issues (GH2228, GH2259)
- Use Series names for column names when using concat with axis=1 (GH2489)
- Raise Exception if start, end, periods all passed to date_range (GH2538)
- Fix Panel resampling issue (GH2537)

26.2.6 pandas 0.9.1

Release date: 2012-11-14

New features

- Can specify multiple sort orders in DataFrame/Series.sort/sort_index (GH928)
- New *top* and *bottom* options for handling NAs in rank (GH1508, GH2159)

- Add *where* and *mask* functions to DataFrame (GH2109, GH2151)
- Add *at_time* and *between_time* functions to DataFrame (GH2149)
- Add flexible *pow* and *rpow* methods to DataFrame (GH2190)

API Changes

- Upsampling period index "spans" intervals. Example: annual periods upsampled to monthly will span all months in each year
- Period.end_time will yield timestamp at last nanosecond in the interval (GH2124, GH2125, GH1764)
- File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

Improvements to existing features

- Time rule inference for week-of-month (e.g. WOM-2FRI) rules (GH2140)
- Improve performance of datetime + business day offset with large number of offset periods
- Improve HTML display of DataFrame objects with hierarchical columns
- Enable referencing of Excel columns by their column names (GH1936)
- DataFrame.dot can accept ndarrays (GH2042)
- Support negative periods in Panel.shift (GH2164)
- Make .drop(...) work with non-unique indexes (GH2101)
- Improve performance of Series/DataFrame.diff (re: GH2087)
- Support unary ~ (__invert__) in DataFrame (GH2110)
- Turn off pandas-style tick locators and formatters (GH2205)
- DataFrame[DataFrame] uses DataFrame.where to compute masked frame (GH2230)

Bug fixes

- Fix some duplicate-column DataFrame constructor issues (GH2079)
- Fix bar plot color cycle issues (GH2082)
- Fix off-center grid for stacked bar plots (GH2157)
- Fix plotting bug if inferred frequency is offset with N > 1 (GH2126)
- Implement comparisons on date offsets with fixed delta (GH2078)
- Handle inf/-inf correctly in read_* parser functions (GH2041)
- Fix matplotlib unicode interaction bug
- Make WLS r-squared match statsmodels 0.5.0 fixed value
- Fix zero-trimming DataFrame formatting bug
- Correctly compute/box datetime64 min/max values from Series.min/max (GH2083)
- Fix unstacking edge case with unrepresented groups (GH2100)
- Fix Series.str failures when using pipe pattern '|' (GH2119)
- Fix pretty-printing of dict entries in Series, DataFrame (GH2144)
- Cast other datetime64 values to nanoseconds in DataFrame ctor (GH2095)
- Alias Timestamp.astimezone to tz_convert, so will yield Timestamp (GH2060)

- Fix timedelta64 formatting from Series (GH2165, GH2146)
- Handle None values gracefully in dict passed to Panel constructor (GH2075)
- Box datetime64 values as Timestamp objects in Series/DataFrame.iget (GH2148)
- Fix Timestamp indexing bug in DatetimeIndex.insert (GH2155)
- Use index name(s) (if any) in DataFrame.to_records (GH2161)
- Don't lose index names in Panel.to_frame/DataFrame.to_panel (GH2163)
- Work around length-0 boolean indexing NumPy bug (GH2096)
- Fix partial integer indexing bug in DataFrame.xs (GH2107)
- Fix variety of cut/qcut string-bin formatting bugs (GH1978, GH1979)
- Raise Exception when xs view not possible of MultiIndex'd DataFrame (GH2117)
- Fix groupby(...).first() issue with datetime64 (GH2133)
- Better floating point error robustness in some rolling_* functions (GH2114, GH2527)
- Fix ewma NA handling in the middle of Series (GH2128)
- Fix numerical precision issues in diff with integer data (GH2087)
- Fix bug in MultiIndex.__getitem__ with NA values (GH2008)
- Fix DataFrame.from_records dict-arg bug when passing columns (GH2179)
- Fix Series and DataFrame.diff for integer dtypes (GH2087, GH2174)
- Fix bug when taking intersection of DatetimeIndex with empty index (GH2129)
- Pass through timezone information when calling DataFrame.align (GH2127)
- Properly sort when joining on datetime64 values (GH2196)
- Fix indexing bug in which False/True were being coerced to 0/1 (GH2199)
- Many unicode formatting fixes (GH2201)
- Fix improper MultiIndex conversion issue when assigning e.g. DataFrame.index (GH2200)
- Fix conversion of mixed-type DataFrame to ndarray with dup columns (GH2236)
- Fix duplicate columns issue (GH2218, GH2219)
- Fix SparseSeries.__pow__ issue with NA input (GH2220)
- Fix icol with integer sequence failure (GH2228)
- Fixed resampling tz-aware time series issue (GH2245)
- SparseDataFrame.icol was not returning SparseSeries (GH2227, GH2229)
- Enable ExcelWriter to handle PeriodIndex (GH2240)
- Fix issue constructing DataFrame from empty Series with name (GH2234)
- Use console-width detection in interactive sessions only (GH1610)
- Fix parallel_coordinates legend bug with mpl 1.2.0 (GH2237)
- Make tz_localize work in corner case of empty Series (GH2248)

26.2.7 pandas 0.9.0

Release date: 10/7/2012

New features

- Add `str.encode` and `str.decode` to Series (GH1706)
- Add *to_latex* method to DataFrame (GH1735)
- Add convenient expanding window equivalents of all rolling_* ops (GH1785)
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance (GH1748, GH1739)
- Recognize and convert more boolean values in file parsing (Yes, No, TRUE, FALSE, variants thereof) (GH1691, GH1295)
- Add Panel.update method, analogous to DataFrame.update (GH1999, GH1988)

Improvements to existing features

- Proper handling of NA values in merge operations (GH1990)
- Add `flags` option for `re.compile` in some Series.str methods (GH1659)
- Parsing of UTC date strings in read_* functions (GH1693)
- Handle generator input to Series (GH1679)
- Add *na_action='ignore'* to Series.map to quietly propagate NAs (GH1661)
- Add args/kwds options to Series.apply (GH1829)
- Add inplace option to Series/DataFrame.reset_index (GH1797)
- Add `level` parameter to `Series.reset_index`
- Add quoting option for DataFrame.to_csv (GH1902)
- Indicate long column value truncation in DataFrame output with ... (GH1854)
- DataFrame.dot will not do data alignment, and also work with Series (GH1915)
- Add `na` option for missing data handling in some vectorized string methods (GH1689)
- If index_label=False in DataFrame.to_csv, do not print fields/commas in the text output. Results in easier importing into R (GH1583)
- Can pass tuple/list of axes to DataFrame.dropna to simplify repeated calls (dropping both columns and rows) (GH924)
- Improve DataFrame.to_html output for hierarchically-indexed rows (do not repeat levels) (GH1929)
- TimeSeries.between_time can now select times across midnight (GH1871)
- Enable *skip_footer* parameter in *ExcelFile.parse* (GH1843)

API Changes

- Change default header names in read_* functions to more Pythonic X0, X1, etc. instead of X.1, X.2. (GH2000)
- Deprecated `day_of_year` API removed from PeriodIndex, use `dayofyear` (GH1723)
- Don't modify NumPy suppress printoption at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by HDFStore (GH1834, GH1824)
- Legacy cruft removed: pandas.stats.misc.quantileTS

- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)
- Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)
- Setting parts of DataFrame/Panel using ix now aligns input Series/DataFrame (GH1630)
- *first* and *last* methods in *GroupBy* no longer drop non-numeric columns (GH1809)
- Resolved inconsistencies in specifying custom NA values in text parser. *na_values* of type dict no longer override default NAs unless *keep_default_na* is set to false explicitly (GH1657)
- Enable *skipfooter* parameter in text parsers as an alias for *skip_footer*

Bug fixes

- Perform arithmetic column-by-column in mixed-type DataFrame to avoid type upcasting issues. Caused downstream DataFrame.diff bug (GH1896)
- Fix matplotlib auto-color assignment when no custom spectrum passed. Also respect passed color keyword argument (GH1711)
- Fix resampling logical error with closed='left' (GH1726)
- Fix critical DatetimeIndex.union bugs (GH1730, GH1719, GH1745, GH1702, GH1753)
- Fix critical DatetimeIndex.intersection bug with unanchored offsets (GH1708)
- Fix MM-YYYY time series indexing case (GH1672)
- Fix case where Categorical group key was not being passed into index in GroupBy result (GH1701)
- Handle Ellipsis in Series.__getitem__/__setitem__ (GH1721)
- Fix some bugs with handling datetime64 scalars of other units in NumPy 1.6 and 1.7 (GH1717)
- Fix performance issue in MultiIndex.format (GH1746)
- Fixed GroupBy bugs interacting with DatetimeIndex asof / map methods (GH1677)
- Handle factors with NAs in pandas.rpy (GH1615)
- Fix statsmodels import in pandas.stats.var (GH1734)
- Fix DataFrame repr/info summary with non-unique columns (GH1700)
- Fix Series.iget_value for non-unique indexes (GH1694)
- Don't lose tzinfo when passing DatetimeIndex as DataFrame column (GH1682)
- Fix tz conversion with time zones that haven't had any DST transitions since first date in the array (GH1673)
- Fix field access with UTC->local conversion on unsorted arrays (GH1756)
- Fix isnull handling of array-like (list) inputs (GH1755)
- Fix regression in handling of Series in Series constructor (GH1671)
- Fix comparison of Int64Index with DatetimeIndex (GH1681)
- Fix min_periods handling in new rolling_max/min at array start (GH1695)
- Fix errors with how='median' and generic NumPy resampling in some cases caused by SeriesBinGrouper (GH1648, GH1688)
- When grouping by level, exclude unobserved levels (GH1697)
- Don't lose tzinfo in DatetimeIndex when shifting by different offset (GH1683)

- Hack to support storing data with a zero-length axis in HDFStore (GH1707)
- Fix DatetimeIndex tz-aware range generation issue (GH1674)
- Fix method='time' interpolation with intraday data (GH1698)
- Don't plot all-NA DataFrame columns as zeros (GH1696)
- Fix bug in scatter_plot with by option (GH1716)
- Fix performance problem in infer_freq with lots of non-unique stamps (GH1686)
- Fix handling of PeriodIndex as argument to create MultiIndex (GH1705)
- Fix re: unicode MultiIndex level names in Series/DataFrame repr (GH1736)
- Handle PeriodIndex in to_datetime instance method (GH1703)
- Support StaticTzInfo in DatetimeIndex infrastructure (GH1692)
- Allow MultiIndex setops with length-0 other type indexes (GH1727)
- Fix handling of DatetimeIndex in DataFrame.to_records (GH1720)
- Fix handling of general objects in isnull on which bool(...) fails (GH1749)
- Fix .ix indexing with MultiIndex ambiguity (GH1678)
- Fix .ix setting logic error with non-unique MultiIndex (GH1750)
- Basic indexing now works on MultiIndex with > 1000000 elements, regression from earlier version of pandas (GH1757)
- Handle non-float64 dtypes in fast DataFrame.corr/cov code paths (GH1761)
- Fix DatetimeIndex.isin to function properly (GH1763)
- Fix conversion of array of tz-aware datetime.datetime to DatetimeIndex with right time zone (GH1777)
- Fix DST issues with generating ancxhored date ranges (GH1778)
- Fix issue calling sort on result of Series.unique (GH1807)
- Fix numerical issue leading to square root of negative number in rolling_std (GH1840)
- Let Series.str.split accept no arguments (like str.split) (GH1859)
- Allow user to have dateutil 2.1 installed on a Python 2 system (GH1851)
- Catch ImportError less aggressively in pandas/__init__.py (GH1845)
- Fix pip source installation bug when installing from GitHub (GH1805)
- Fix error when window size > array size in rolling_apply (GH1850)
- Fix pip source installation issues via SSH from GitHub
- Fix OLS.summary when column is a tuple (GH1837)
- Fix bug in __doc__ patching when -OO passed to interpreter (GH1792 GH1741 GH1774)
- Fix unicode console encoding issue in IPython notebook (GH1782, GH1768)
- Fix unicode formatting issue with Series.name (GH1782)
- Fix bug in DataFrame.duplicated with datetime64 columns (GH1833)
- Fix bug in Panel internals resulting in error when doing fillna after truncate not changing size of panel (GH1823)
- Prevent segfault due to MultiIndex not being supported in HDFStore table format (GH1848)

- Fix UnboundLocalError in Panel.__setitem__ and add better error (GH1826)
- Fix to_csv issues with list of string entries. Isnull works on list of strings now too (GH1791)
- Fix Timestamp comparisons with datetime values outside the nanosecond range (1677-2262)
- Revert to prior behavior of normalize_date with datetime.date objects (return datetime)
- Fix broken interaction between np.nansum and Series.any/all
- Fix bug with multiple column date parsers (GH1866)
- DatetimeIndex.union(Int64Index) was broken
- Make plot x vs y interface consistent with integer indexing (GH1842)
- set_index inplace modified data even if unique check fails (GH1831)
- Only use Q-OCT/NOV/DEC in quarterly frequency inference (GH1789)
- Upcast to dtype=object when unstacking boolean DataFrame (GH1820)
- Fix float64/float32 merging bug (GH1849)
- Fixes to Period.start_time for non-daily frequencies (GH1857)
- Fix failure when converter used on index_col in read_csv (GH1835)
- Implement PeriodIndex.append so that pandas.concat works correctly (GH1815)
- Avoid Cython out-of-bounds access causing segfault sometimes in pad_2d, backfill_2d
- Fix resampling error with intraday times and anchored target time (like AS-DEC) (GH1772)
- Fix .ix indexing bugs with mixed-integer indexes (GH1799)
- Respect passed color keyword argument in Series.plot (GH1890)
- Fix rolling_min/max when the window is larger than the size of the input array. Check other malformed inputs (GH1899, GH1897)
- Rolling variance / standard deviation with only a single observation in window (GH1884)
- Fix unicode sheet name failure in to_excel (GH1828)
- Override DatetimeIndex.min/max to return Timestamp objects (GH1895)
- Fix column name formatting issue in length-truncated column (GH1906)
- Fix broken handling of copying Index metadata to new instances created by view(...) calls inside the NumPy infrastructure
- Support datetime.date again in DateOffset.rollback/rollforward
- Raise Exception if set passed to Series constructor (GH1913)
- Add TypeError when appending HDFStore table w/ wrong index type (GH1881)
- Don't raise exception on empty inputs in EW functions (e.g. ewma) (GH1900)
- Make asof work correctly with PeriodIndex (GH1883)
- Fix extlinks in doc build
- Fill boolean DataFrame with NaN when calling shift (GH1814)
- Fix setuptools bug causing pip not to Cythonize .pyx files sometimes
- Fix negative integer indexing regression in .ix from 0.7.x (GH1888)

- Fix error while retrieving timezone and utc offset from subclasses of datetime.tzinfo without .zone and ._utcoffset attributes (GH1922)
- Fix DataFrame formatting of small, non-zero FP numbers (GH1911)
- Various fixes by upcasting of date -> datetime (GH1395)
- Raise better exception when passing multiple functions with the same name, such as lambdas, to GroupBy.aggregate
- Fix DataFrame.apply with axis=1 on a non-unique index (GH1878)
- Proper handling of Index subclasses in pandas.unique (GH1759)
- Set index names in DataFrame.from_records (GH1744)
- Fix time series indexing error with duplicates, under and over hash table size cutoff (GH1821)
- Handle list keys in addition to tuples in DataFrame.xs when partial-indexing a hierarchically-indexed DataFrame (GH1796)
- Support multiple column selection in DataFrame.__getitem__ with duplicate columns (GH1943)
- Fix time zone localization bug causing improper fields (e.g. hours) in time zones that have not had a UTC transition in a long time (GH1946)
- Fix errors when parsing and working with with fixed offset timezones (GH1922, GH1928)
- Fix text parser bug when handling UTC datetime objects generated by dateutil (GH1693)
- Fix plotting bug when 'B' is the inferred frequency but index actually contains weekends (GH1668, GH1669)
- Fix plot styling bugs (GH1666, GH1665, GH1658)
- Fix plotting bug with index/columns with unicode (GH1685)
- Fix DataFrame constructor bug when passed Series with datetime64 dtype in a dict (GH1680)
- Fixed regression in generating DatetimeIndex using timezone aware datetime.datetime (GH1676)
- Fix DataFrame bug when printing concatenated DataFrames with duplicated columns (GH1675)
- Fixed bug when plotting time series with multiple intraday frequencies (GH1732)
- Fix bug in DataFrame.duplicated to enable iterables other than list-types as input argument (GH1773)
- Fix resample bug when passed list of lambdas as *how* argument (GH1808)
- Repr fix for MultiIndex level with all NAs (GH1971)
- Fix PeriodIndex slicing bug when slice start/end are out-of-bounds (GH1977)
- Fix read_table bug when parsing unicode (GH1975)
- Fix BlockManager.iget bug when dealing with non-unique MultiIndex as columns (GH1970)
- Fix reset_index bug if both drop and level are specified (GH1957)
- Work around unsafe NumPy object->int casting with Cython function (GH1987)
- Fix datetime64 formatting bug in DataFrame.to_csv (GH1993)
- Default start date in pandas.io.data to 1/1/2000 as the docs say (GH2011)

26.2.8 pandas 0.8.1

Release date: July 22, 2012

New features

- Add vectorized, NA-friendly string methods to Series (GH1621, GH620)
- Can pass dict of per-column line styles to DataFrame.plot (GH1559)
- Selective plotting to secondary y-axis on same subplot (GH1640)
- Add new `bootstrap_plot` plot function
- Add new `parallel_coordinates` plot function (GH1488)
- Add `radviz` plot function (GH1566)
- Add `multi_sparse` option to `set_printoptions` to modify display of hierarchical indexes (GH1538)
- Add `dropna` method to Panel (GH171)

Improvements to existing features

- Use moving min/max algorithms from Bottleneck in rolling_min/rolling_max for > 100x speedup. (GH1504, GH50)
- Add Cython group median method for >15x speedup (GH1358)
- Drastically improve `to_datetime` performance on ISO8601 datetime strings (with no time zones) (GH1571)
- Improve single-key groupby performance on large data sets, accelerate use of groupby with a Categorical variable
- Add ability to append hierarchical index levels with `set_index` and to drop single levels with `reset_index` (GH1569, GH1577)
- Always apply passed functions in `resample`, even if upsampling (GH1596)
- Avoid unnecessary copies in DataFrame constructor with explicit dtype (GH1572)
- Cleaner DatetimeIndex string representation with 1 or 2 elements (GH1611)
- Improve performance of array-of-Period to PeriodIndex, convert such arrays to PeriodIndex inside Index (GH1215)
- More informative string representation for weekly Period objects (GH1503)
- Accelerate 3-axis multi data selection from homogeneous Panel (GH979)
- Add `adjust` option to ewma to disable adjustment factor (GH1584)
- Add new matplotlib converters for high frequency time series plotting (GH1599)
- Handling of tz-aware datetime.datetime objects in to_datetime; raise Exception unless utc=True given (GH1581)

Bug fixes

- Fix NA handling in DataFrame.to_panel (GH1582)
- Handle TypeError issues inside PyObject_RichCompareBool calls in khash (GH1318)
- Fix resampling bug to lower case daily frequency (GH1588)
- Fix kendall/spearman DataFrame.corr bug with no overlap (GH1595)
- Fix bug in DataFrame.set_index (GH1592)
- Don't ignore axes in boxplot if by specified (GH1565)

- Fix Panel .ix indexing with integers bug (GH1603)
- Fix Partial indexing bugs (years, months, ...) with PeriodIndex (GH1601)
- Fix MultiIndex console formatting issue (GH1606)
- Unordered index with duplicates doesn't yield scalar location for single entry (GH1586)
- Fix resampling of tz-aware time series with "anchored" freq (GH1591)
- Fix DataFrame.rank error on integer data (GH1589)
- Selection of multiple SparseDataFrame columns by list in __getitem__ (GH1585)
- Override Index.tolist for compatibility with MultiIndex (GH1576)
- Fix hierarchical summing bug with MultiIndex of length 1 (GH1568)
- Work around numpy.concatenate use/bug in Series.set_value (GH1561)
- Ensure Series/DataFrame are sorted before resampling (GH1580)
- Fix unhandled IndexError when indexing very large time series (GH1562)
- Fix DatetimeIndex intersection logic error with irregular indexes (GH1551)
- Fix unit test errors on Python 3 (GH1550)
- Fix .ix indexing bugs in duplicate DataFrame index (GH1201)
- Better handle errors with non-existing objects in HDFStore (GH1254)
- Don't copy int64 array data in DatetimeIndex when copy=False (GH1624)
- Fix resampling of conforming periods quarterly to annual (GH1622)
- Don't lose index name on resampling (GH1631)
- Support python-dateutil version 2.1 (GH1637)
- Fix broken scatter_matrix axis labeling, esp. with time series (GH1625)
- Fix cases where extra keywords weren't being passed on to matplotlib from Series.plot (GH1636)
- Fix BusinessMonthBegin logic for dates before 1st bday of month (GH1645)
- Ensure string alias converted (valid in DatetimeIndex.get_loc) in DataFrame.xs / __getitem__ (GH1644)
- Fix use of string alias timestamps with tz-aware time series (GH1647)
- Fix Series.max/min and Series.describe on len-0 series (GH1650)
- Handle None values in dict passed to concat (GH1649)
- Fix Series.interpolate with method='values' and DatetimeIndex (GH1646)
- Fix IndexError in left merges on a DataFrame with 0-length (GH1628)
- Fix DataFrame column width display with UTF-8 encoded characters (GH1620)
- Handle case in pandas.io.data.get_data_yahoo where Yahoo! returns duplicate dates for most recent business day
- Avoid downsampling when plotting mixed frequencies on the same subplot (GH1619)
- Fix read_csv bug when reading a single line (GH1553)
- Fix bug in C code causing monthly periods prior to December 1969 to be off (GH1570)

26.2.9 pandas 0.8.0

Release date: 6/29/2012

New features

- New unified DatetimeIndex class for nanosecond-level timestamp data
- New Timestamp datetime.datetime subclass with easy time zone conversions, and support for nanoseconds
- New PeriodIndex class for timespans, calendar logic, and Period scalar object
- High performance resampling of timestamp and period data. New *resample* method of all pandas data structures
- New frequency names plus shortcut string aliases like '15h', '1h30min'
- Time series string indexing shorthand (GH222)
- Add week, dayofyear array and other timestamp array-valued field accessor functions to DatetimeIndex
- Add GroupBy.prod optimized aggregation function and 'prod' fast time series conversion method (GH1018)
- Implement robust frequency inference function and *inferred_freq* attribute on DatetimeIndex (GH391)
- New `tz_convert` and `tz_localize` methods in Series / DataFrame
- Convert DatetimeIndexes to UTC if time zones are different in join/setops (GH864)
- Add limit argument for forward/backward filling to reindex, fillna, etc. (GH825 and others)
- Add support for indexes (dates or otherwise) with duplicates and common sense indexing/selection functionality
- Series/DataFrame.update methods, in-place variant of combine_first (GH961)
- Add `match` function to API (GH502)
- Add Cython-optimized first, last, min, max, prod functions to GroupBy (GH994, GH1043)
- Dates can be split across multiple columns (GH1227, GH1186)
- Add experimental support for converting pandas DataFrame to R data.frame via rpy2 (GH350, GH1212)
- Can pass list of (name, function) to GroupBy.aggregate to get aggregates in a particular order (GH610)
- Can pass dicts with lists of functions or dicts to GroupBy aggregate to do much more flexible multiple function aggregation (GH642, GH610)
- New ordered_merge functions for merging DataFrames with ordered data. Also supports group-wise merging for panel data (GH813)
- Add keys() method to DataFrame
- Add flexible replace method for replacing potentially values to Series and DataFrame (GH929, GH1241)
- Add 'kde' plot kind for Series/DataFrame.plot (GH1059)
- More flexible multiple function aggregation with GroupBy
- Add pct_change function to Series/DataFrame
- Add option to interpolate by Index values in Series.interpolate (GH1206)
- Add `max_colwidth` option for DataFrame, defaulting to 50
- Conversion of DataFrame through rpy2 to R data.frame (GH1282,)
- Add keys() method on DataFrame (GH1240)
- Add new `match` function to API (similar to R) (GH502)

- Add dayfirst option to parsers (GH854)
- Add `method` argument to `align` method for forward/backward fillin (GH216)
- Add Panel.transpose method for rearranging axes (GH695)
- Add new `cut` function (patterned after R) for discretizing data into equal range-length bins or arbitrary breaks of your choosing (GH415)
- Add new `qcut` for cutting with quantiles (GH1378)
- Add `value_counts` top level array method (GH1392)
- Added Andrews curves plot tupe (GH1325)
- Add lag plot (GH1440)
- Add autocorrelation_plot (GH1425)
- Add support for tox and Travis CI (GH1382)
- Add support for Categorical use in GroupBy (GH292)
- Add `any` and `all` methods to DataFrame (GH1416)
- Add `secondary_y` option to Series.plot
- Add experimental `lreshape` function for reshaping wide to long

Improvements to existing features

- Switch to klib/khash-based hash tables in Index classes for better performance in many cases and lower memory footprint
- Shipping some functions from scipy.stats to reduce dependency, e.g. Series.describe and DataFrame.describe (GH1092)
- Can create MultiIndex by passing list of lists or list of arrays to Series, DataFrame constructor, etc. (GH831)
- Can pass arrays in addition to column names to DataFrame.set_index (GH402)
- Improve the speed of "square" reindexing of homogeneous DataFrame objects by significant margin (GH836)
- Handle more dtypes when passed MaskedArrays in DataFrame constructor (GH406)
- Improved performance of join operations on integer keys (GH682)
- Can pass multiple columns to GroupBy object, e.g. grouped[[col1, col2]] to only aggregate a subset of the value columns (GH383)
- Add histogram / kde plot options for scatter_matrix diagonals (GH1237)
- Add inplace option to Series/DataFrame.rename and sort_index, DataFrame.drop_duplicates (GH805, GH207)
- More helpful error message when nothing passed to Series.reindex (GH1267)
- Can mix array and scalars as dict-value inputs to DataFrame ctor (GH1329)
- Use DataFrame columns' name for legend title in plots
- Preserve frequency in DatetimeIndex when possible in boolean indexing operations
- Promote datetime.date values in data alignment operations (GH867)
- Add `order` method to Index classes (GH1028)
- Avoid hash table creation in large monotonic hash table indexes (GH1160)
- Store time zones in HDFStore (GH1232)

- Enable storage of sparse data structures in HDFStore (GH85)
- Enable Series.asof to work with arrays of timestamp inputs
- Cython implementation of DataFrame.corr speeds up by > 100x (GH1349, GH1354)
- Exclude "nuisance" columns automatically in GroupBy.transform (GH1364)
- Support functions-as-strings in GroupBy.transform (GH1362)
- Use index name as xlabel/ylabel in plots (GH1415)
- Add `convert_dtype` option to Series.apply to be able to leave data as dtype=object (GH1414)
- Can specify all index level names in concat (GH1419)
- Add `dialect` keyword to parsers for quoting conventions (GH1363)
- Enable DataFrame[bool_DataFrame] += value (GH1366)
- Add `retries` argument to `get_data_yahoo` to try to prevent Yahoo! API 404s (GH826)
- Improve performance of reshaping by using O(N) categorical sorting
- Series names will be used for index of DataFrame if no index passed (GH1494)
- Header argument in DataFrame.to_csv can accept a list of column names to use instead of the object's columns (GH921)
- Add `raise_conflict` argument to DataFrame.update (GH1526)
- Support file-like objects in ExcelFile (GH1529)

API Changes

- Rename *pandas._tseries* to *pandas.lib*
- Rename Factor to Categorical and add improvements. Numerous Categorical bug fixes
- Frequency name overhaul, WEEKDAY/EOM and rules with @ deprecated. get_legacy_offset_name backwards compatibility function added
- Raise ValueError in DataFrame.__nonzero__, so "if df" no longer works (GH1073)
- Change BDay (business day) to not normalize dates by default (GH506)
- Remove deprecated DataMatrix name
- Default merge suffixes for overlap now have underscores instead of periods to facilitate tab completion, etc. (GH1239)
- Deprecation of offset, time_rule timeRule parameters throughout codebase
- Series.append and DataFrame.append no longer check for duplicate indexes by default, add verify_integrity parameter (GH1394)
- Refactor Factor class, old constructor moved to Factor.from_array
- Modified internals of MultiIndex to use less memory (no longer represented as array of tuples) internally, speed up construction time and many methods which construct intermediate hierarchical indexes (GH1467)

Bug fixes

- Fix OverflowError from storing pre-1970 dates in HDFStore by switching to datetime64 (GH179)
- Fix logical error with February leap year end in YearEnd offset
- Series([False, nan]) was getting casted to float64 (GH1074)
- Fix binary operations between boolean Series and object Series with booleans and NAs (GH1074, GH1079)

- Couldn't assign whole array to column in mixed-type DataFrame via .ix (GH1142)
- Fix label slicing issues with float index values (GH1167)
- Fix segfault caused by empty groups passed to groupby (GH1048)
- Fix occasionally misbehaved reindexing in the presence of NaN labels (GH522)
- Fix imprecise logic causing weird Series results from .apply (GH1183)
- Unstack multiple levels in one shot, avoiding empty columns in some cases. Fix pivot table bug (GH1181)
- Fix formatting of MultiIndex on Series/DataFrame when index name coincides with label (GH1217)
- Handle Excel 2003 #N/A as NaN from xlrd (GH1213, GH1225)
- Fix timestamp locale-related deserialization issues with HDFStore by moving to datetime64 representation (GH1081, GH809)
- Fix DataFrame.duplicated/drop_duplicates NA value handling (GH557)
- Actually raise exceptions in fast reducer (GH1243)
- Fix various timezone-handling bugs from 0.7.3 (GH969)
- GroupBy on level=0 discarded index name (GH1313)
- Better error message with unmergeable DataFrames (GH1307)
- Series.__repr__ alignment fix with unicode index values (GH1279)
- Better error message if nothing passed to reindex (GH1267)
- More robust NA handling in DataFrame.drop_duplicates (GH557)
- Resolve locale-based and pre-epoch HDF5 timestamp deserialization issues (GH973, GH1081, GH179)
- Implement Series.repeat (GH1229)
- Fix indexing with namedtuple and other tuple subclasses (GH1026)
- Fix float64 slicing bug (GH1167)
- Parsing integers with commas (GH796)
- Fix groupby improper data type when group consists of one value (GH1065)
- Fix negative variance possibility in nanvar resulting from floating point error (GH1090)
- Consistently set name on groupby pieces (GH184)
- Treat dict return values as Series in GroupBy.apply (GH823)
- Respect column selection for DataFrame in in GroupBy.transform (GH1365)
- Fix MultiIndex partial indexing bug (GH1352)
- Enable assignment of rows in mixed-type DataFrame via .ix (GH1432)
- Reset index mapping when grouping Series in Cython (GH1423)
- Fix outer/inner DataFrame.join with non-unique indexes (GH1421)
- Fix MultiIndex groupby bugs with empty lower levels (GH1401)
- Calling fillna with a Series will have same behavior as with dict (GH1486)
- SparseSeries reduction bug (GH1375)
- Fix unicode serialization issue in HDFStore (GH1361)

- Pass keywords to pyplot.boxplot in DataFrame.boxplot (GH1493)
- Bug fixes in MonthBegin (GH1483)
- Preserve MultiIndex names in drop (GH1513)
- Fix Panel DataFrame slice-assignment bug (GH1533)
- Don't use locals() in read_* functions (GH1547)

26.2.10 pandas 0.7.3

Release date: April 12, 2012

New features / modules

- Support for non-unique indexes: indexing and selection, many-to-one and many-to-many joins (GH1306)
- Added fixed-width file reader, read_fwf (GH952)
- Add group_keys argument to groupby to not add group names to MultiIndex in result of apply (GH938)
- DataFrame can now accept non-integer label slicing (GH946). Previously only DataFrame.ix was able to do so.
- DataFrame.apply now retains name attributes on Series objects (GH983)
- Numeric DataFrame comparisons with non-numeric values now raises proper TypeError (GH943). Previously raise "PandasError: DataFrame constructor not properly called!"
- Add `kurt` methods to Series and DataFrame (GH964)
- Can pass dict of column -> list/set NA values for text parsers (GH754)
- Allows users specified NA values in text parsers (GH754)
- Parsers checks for openpyxl dependency and raises ImportError if not found (GH1007)
- New factory function to create HDFStore objects that can be used in a with statement so users do not have to explicitly call HDFStore.close (GH1005)
- pivot_table is now more flexible with same parameters as groupby (GH941)
- Added stacked bar plots (GH987)
- scatter_matrix method in pandas/tools/plotting.py (GH935)
- DataFrame.boxplot returns plot results for ex-post styling (GH985)
- Short version number accessible as pandas.version.short_version (GH930)
- Additional documentation in panel.to_frame (GH942)
- More informative Series.apply docstring regarding element-wise apply (GH977)
- Notes on rpy2 installation (GH1006)
- Add rotation and font size options to hist method (GH1012)
- Use exogenous / X variable index in result of OLS.y_predict. Add OLS.predict method (GH1027, GH1008)

API Changes

- Calling apply on grouped Series, e.g. describe(), will no longer yield DataFrame by default. Will have to call unstack() to get prior behavior
- NA handling in non-numeric comparisons has been tightened up (GH933, GH953)
- No longer assign dummy names key_0, key_1, etc. to groupby index (GH1291)

Bug fixes

- Fix logic error when selecting part of a row in a DataFrame with a MultiIndex index (GH1013)
- Series comparison with Series of differing length causes crash (GH1016).
- Fix bug in indexing when selecting section of hierarchically-indexed row (GH1013)
- DataFrame.plot(logy=True) has no effect (GH1011).
- Broken arithmetic operations between SparsePanel-Panel (GH1015)
- Unicode repr issues in MultiIndex with non-ascii characters (GH1010)
- DataFrame.lookup() returns inconsistent results if exact match not present (GH1001)
- DataFrame arithmetic operations not treating None as NA (GH992)
- DataFrameGroupBy.apply returns incorrect result (GH991)
- Series.reshape returns incorrect result for multiple dimensions (GH989)
- Series.std and Series.var ignores ddof parameter (GH934)
- DataFrame.append loses index names (GH980)
- DataFrame.plot(kind='bar') ignores color argument (GH958)
- Inconsistent Index comparison results (GH948)
- Improper int dtype DataFrame construction from data with NaN (GH846)
- Removes default 'result' name in grouby results (GH995)
- DataFrame.from_records no longer mutate input columns (GH975)
- Use Index name when grouping by it (GH1313)

26.2.11 pandas 0.7.2

Release date: March 16, 2012

New features / modules

- Add additional tie-breaking methods in DataFrame.rank (GH874)
- Add ascending parameter to rank in Series, DataFrame (GH875)
- Add sort_columns parameter to allow unsorted plots (GH918)
- IPython tab completion on GroupBy objects

API Changes

- Series.sum returns 0 instead of NA when called on an empty series. Analogously for a DataFrame whose rows or columns are length 0 (GH844)

Improvements to existing features

- Don't use groups dict in Grouper.size (GH860)
- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Enable column access via attributes on GroupBy (GH882)
- Enable setting existing columns (only) via attributes on DataFrame, Panel (GH883)
- Intercept __builtin__.sum in groupby (GH885)

- Can pass dict to DataFrame.fillna to use different values per column (GH661)
- Can select multiple hierarchical groups by passing list of values in .ix (GH134)
- Add level keyword to `drop` for dropping values from a level (GH159)
- Add `coerce_float` option on DataFrame.from_records (GH893)
- Raise exception if passed date_parser fails in `read_csv`
- Add `axis` option to DataFrame.fillna (GH174)
- Fixes to Panel to make it easier to subclass (GH888)

Bug fixes

- Fix overflow-related bugs in groupby (GH850, GH851)
- Fix unhelpful error message in parsers (GH856)
- Better err msg for failed boolean slicing of dataframe (GH859)
- Series.count cannot accept a string (level name) in the level argument (GH869)
- Group index platform int check (GH870)
- concat on axis=1 and ignore_index=True raises TypeError (GH871)
- Further unicode handling issues resolved (GH795)
- Fix failure in multiindex-based access in Panel (GH880)
- Fix DataFrame boolean slice assignment failure (GH881)
- Fix combineAdd NotImplementedError for SparseDataFrame (GH887)
- Fix DataFrame.to_html encoding and columns (GH890, GH891, GH909)
- Fix na-filling handling in mixed-type DataFrame (GH910)
- Fix to DataFrame.set_value with non-existant row/col (GH911)
- Fix malformed block in groupby when excluding nuisance columns (GH916)
- Fix inconsistant NA handling in dtype=object arrays (GH925)
- Fix missing center-of-mass computation in ewmcov (GH862)
- Don't raise exception when opening read-only HDF5 file (GH847)
- Fix possible out-of-bounds memory access in 0-length Series (GH917)

26.2.12 pandas 0.7.1

Release date: February 29, 2012

New features / modules

- 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

Improvements to existing features

- Improve performance and memory usage of fillna on DataFrame
- Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH787)

Bug fixes

- Fix memory leak when inserting large number of columns into a single DataFrame (GH790)
- Appending length-0 DataFrame with new columns would not result in those new columns being part of the resulting concatenated DataFrame (GH782)
- Fixed groupby corner case when passing dictionary grouper and as_index is False (GH819)
- Fixed bug whereby bool array sometimes had object dtype (GH820)
- Fix exception thrown on np.diff (GH816)
- Fix to_records where columns are non-strings (GH822)
- Fix Index.intersection where indices have incomparable types (GH811)
- Fix ExcelFile throwing an exception for two-line file (GH837)
- Add clearer error message in csv parser (GH835)
- Fix loss of fractional seconds in HDFStore (GH513)
- Fix DataFrame join where columns have datetimes (GH787)
- Work around numpy performance issue in take (GH817)
- Improve comparison operations for NA-friendliness (GH801)
- Fix indexing operation for floating point values (GH780, GH798)
- Fix groupby case resulting in malformed dataframe (GH814)
- Fix behavior of reindex of Series dropping name (GH812)
- Improve on redudant groupby computation (GH775)
- Catch possible NA assignment to int/bool series with exception (GH839)

26.2.13 pandas 0.7.0

Release date: 2/9/2012

New features / modules

- New `merge` function for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains (GH220, GH249, GH267)
- New `concat` function for concatenating DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of `DataFrame.append` (GH468, GH479, GH273)
- Handle differently-indexed output values in `DataFrame.apply` (GH498)
- Can pass list of dicts (e.g., a list of shallow JSON objects) to DataFrame constructor (GH526)
- Add `reorder_levels` method to Series and DataFrame (GH534)

- Add dict-like `get` function to DataFrame and Panel (GH521)
- `DataFrame.iterrows` method for efficiently iterating through the rows of a DataFrame
- Added `DataFrame.to_panel` with code adapted from `LongPanel.to_long`
- `reindex_axis` method added to DataFrame
- Add `level` option to binary arithmetic functions on `DataFrame` and `Series`
- Add `level` option to the `reindex` and `align` methods on Series and DataFrame for broadcasting values across a level (GH542, GH552, others)
- Add attribute-based item access to `Panel` and add IPython completion (PR GH554)
- Add `logy` option to `Series.plot` for log-scaling on the Y axis
- Add `index`, `header`, and `justify` options to `DataFrame.to_string`. Add option to (GH570, GH571)
- Can pass multiple DataFrames to `DataFrame.join` to join on index (GH115)
- Can pass multiple Panels to `Panel.join` (GH115)
- Can pass multiple DataFrames to *DataFrame.append* to concatenate (stack) and multiple Series to `Series.append` too
- Added `justify` argument to `DataFrame.to_string` to allow different alignment of column headers
- Add `sort` option to GroupBy to allow disabling sorting of the group keys for potential speedups (GH595)
- Can pass MaskedArray to Series constructor (GH563)
- Add Panel item access via attributes and IPython completion (GH554)
- Implement `DataFrame.lookup`, fancy-indexing analogue for retrieving values given a sequence of row and column labels (GH338)
- Add `verbose` option to `read_csv` and `read_table` to show number of NA values inserted in non-numeric columns (GH614)
- Can pass a list of dicts or Series to `DataFrame.append` to concatenate multiple rows (GH464)
- Add `level` argument to `DataFrame.xs` for selecting data from other MultiIndex levels. Can take one or more levels with potentially a tuple of keys for flexible retrieval of data (GH371, GH629)
- New `crosstab` function for easily computing frequency tables (GH170)
- Can pass a list of functions to aggregate with groupby on a DataFrame, yielding an aggregated result with hierarchical columns (GH166)
- Add integer-indexing functions `iget` in Series and `irow` / `iget` in DataFrame (GH628)
- Add new `Series.unique` function, significantly faster than `numpy.unique` (GH658)
- Add new `cummin` and `cummax` instance methods to `Series` and `DataFrame` (GH647)
- Add new `value_range` function to return min/max of a dataframe (GH288)
- Add `drop` parameter to `reset_index` method of `DataFrame` and added method to `Series` as well (GH699)
- Add `isin` method to Index objects, works just like `Series.isin` (GH GH657)
- Implement array interface on Panel so that ufuncs work (re: GH740)
- Add `sort` option to `DataFrame.join` (GH731)
- Improved handling of NAs (propagation) in binary operations with dtype=object arrays (GH737)

- Add `abs` method to Pandas objects
- Added `algorithms` module to start collecting central algos

API Changes

- Label-indexing with integer indexes now raises KeyError if a label is not found instead of falling back on location-based indexing (GH700)
- Label-based slicing via `ix` or `[]` on Series will now only work if exact matches for the labels are found or if the index is monotonic (for range selections)
- Label-based slicing and sequences of labels can be passed to `[]` on a Series for both getting and setting (GH86)
- *[]* operator (`__getitem__` and `__setitem__`) will raise KeyError with integer indexes when an index is not contained in the index. The prior behavior would fall back on position-based indexing if a key was not found in the index which would lead to subtle bugs. This is now consistent with the behavior of `.ix` on DataFrame and friends (GH328)
- Rename `DataFrame.delevel` to `DataFrame.reset_index` and add deprecation warning
- *Series.sort* (an in-place operation) called on a Series which is a view on a larger array (e.g. a column in a DataFrame) will generate an Exception to prevent accidentally modifying the data source (GH316)
- Refactor to remove deprecated `LongPanel` class (GH552)
- Deprecated `Panel.to_long`, renamed to `to_frame`
- Deprecated `colSpace` argument in `DataFrame.to_string`, renamed to `col_space`
- Rename `precision` to `accuracy` in engineering float formatter (GH GH395)
- The default delimiter for `read_csv` is comma rather than letting `csv.Sniffer` infer it
- Rename `col_or_columns` argument in `DataFrame.drop_duplicates` (GH GH734)

Improvements to existing features

- Better error message in DataFrame constructor when passed column labels don't match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don't print length by default in Series.to_string, add *length* option (GH GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of Series.__getitem__ for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in setup.py if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also (GH536)
- Default name assignment when calling `reset_index` on DataFrame with a regular (non-hierarchical) index (GH476)
- Use Cythonized groupers when possible in Series/DataFrame stat ops with `level` parameter passed (GH545)

- Ported skiplist data structure to C to speed up `rolling_median` by about 5-10x in most typical use cases (GH374)
- Some performance enhancements in constructing a Panel from a dict of DataFrame objects
- Made `Index._get_duplicates` a public method by removing the underscore
- Prettier printing of floats, and column spacing fix (GH395, GH571)
- Add `bold_rows` option to DataFrame.to_html (GH586)
- Improve the performance of `DataFrame.sort_index` by up to 5x or more when sorting by multiple columns
- Substantially improve performance of DataFrame and Series constructors when passed a nested dict or dict, respectively (GH540, GH621)
- Modified setup.py so that pip / setuptools will install dependencies (GH GH507, various pull requests)
- Unstack called on DataFrame with non-MultiIndex will return Series (GH GH477)
- Improve DataFrame.to_string and console formatting to be more consistent in the number of displayed digits (GH395)
- Use bottleneck if available for performing NaN-friendly statistical operations that it implemented (GH91)
- Monkey-patch context to traceback in `DataFrame.apply` to indicate which row/column the function application failed on (GH614)
- Improved ability of read_table and read_clipboard to parse console-formatted DataFrames (can read the row of index names, etc.)
- Can pass list of group labels (without having to convert to an ndarray yourself) to `groupby` in some cases (GH659)
- Use `kind` argument to Series.order for selecting different sort kinds (GH668)
- Add option to Series.to_csv to omit the index (GH684)
- Add `delimiter` as an alternative to `sep` in `read_csv` and other parsing functions
- Substantially improved performance of groupby on DataFrames with many columns by aggregating blocks of columns all at once (GH745)
- Can pass a file handle or StringIO to Series/DataFrame.to_csv (GH765)
- Can pass sequence of integers to DataFrame.irow(icol) and Series.iget, (GH GH654)
- Prototypes for some vectorized string functions
- Add float64 hash table to solve the Series.unique problem with NAs (GH714)
- Memoize objects when reading from file to reduce memory footprint
- Can get and set a column of a DataFrame with hierarchical columns containing "empty" ('') lower levels without passing the empty levels (PR GH768)

Bug fixes

- Raise exception in out-of-bounds indexing of Series instead of seg-faulting, regression from earlier releases (GH495)
- Fix error when joining DataFrames of different dtypes within the same typeclass (e.g. float32 and float64) (GH486)
- Fix bug in Series.min/Series.max on objects like datetime.datetime (GH GH487)
- Preserve index names in Index.union (GH501)

- Fix bug in Index joining causing subclass information (like DateRange type) to be lost in some cases (GH500)
- Accept empty list as input to DataFrame constructor, regression from 0.6.0 (GH491)
- Can output DataFrame and Series with ndarray objects in a dtype=object array (GH490)
- Return empty string from Series.to_string when called on empty Series (GH GH488)
- Fix exception passing empty list to DataFrame.from_records
- Fix Index.format bug (excluding name field) with datetimes with time info
- Fix scalar value access in Series to always return NumPy scalars, regression from prior versions (GH510)
- Handle rows skipped at beginning of file in read_* functions (GH505)
- Handle improper dtype casting in `set_value` methods
- Unary '-' / __neg__ operator on DataFrame was returning integer values
- Unbox 0-dim ndarrays from certain operators like all, any in Series
- Fix handling of missing columns (was combine_first-specific) in DataFrame.combine for general case (GH529)
- Fix type inference logic with boolean lists and arrays in DataFrame indexing
- Use centered sum of squares in R-square computation if entity_effects=True in panel regression
- Handle all NA case in Series.{corr, cov}, was raising exception (GH548)
- Aggregating by multiple levels with `level` argument to DataFrame, Series stat method, was broken (GH545)
- Fix Cython buf when converter passed to read_csv produced a numeric array (buffer dtype mismatch when passed to Cython type inference function) (GH GH546)
- Fix exception when setting scalar value using .ix on a DataFrame with a MultiIndex (GH551)
- Fix outer join between two DateRanges with different offsets that returned an invalid DateRange
- Cleanup DataFrame.from_records failure where index argument is an integer
- Fix Data.from_records failure when passed a dictionary
- Fix NA handling in {Series, DataFrame}.rank with non-floating point dtypes
- Fix bug related to integer type-checking in .ix-based indexing
- Handle non-string index name passed to DataFrame.from_records
- DataFrame.insert caused the columns name(s) field to be discarded (GH527)
- Fix erroneous in monotonic many-to-one left joins
- Fix DataFrame.to_string to remove extra column white space (GH571)
- Format floats to default to same number of digits (GH395)
- Added decorator to copy docstring from one function to another (GH449)
- Fix error in monotonic many-to-one left joins
- Fix __eq__ comparison between DateOffsets with different relativedelta keywords passed
- Fix exception caused by parser converter returning strings (GH583)
- Fix MultiIndex formatting bug with integer names (GH601)
- Fix bug in handling of non-numeric aggregates in Series.groupby (GH612)
- Fix TypeError with tuple subclasses (e.g. namedtuple) in DataFrame.from_records (GH611)

- Catch misreported console size when running IPython within Emacs
- Fix minor bug in pivot table margins, loss of index names and length-1 'All' tuple in row labels
- Add support for legacy WidePanel objects to be read from HDFStore
- Fix out-of-bounds segfault in pad_object and backfill_object methods when either source or target array are empty
- Could not create a new column in a DataFrame from a list of tuples
- Fix bugs preventing SparseDataFrame and SparseSeries working with groupby (GH666)
- Use sort kind in Series.sort / argsort (GH668)
- Fix DataFrame operations on non-scalar, non-pandas objects (GH672)
- Don't convert DataFrame column to integer type when passing integer to __setitem__ (GH669)
- Fix downstream bug in pivot_table caused by integer level names in MultiIndex (GH678)
- Fix SparseSeries.combine_first when passed a dense Series (GH687)
- Fix performance regression in HDFStore loading when DataFrame or Panel stored in table format with datetimes
- Raise Exception in DateRange when offset with n=0 is passed (GH683)
- Fix get/set inconsistency with .ix property and integer location but non-integer index (GH707)
- Use right dropna function for SparseSeries. Return dense Series for NA fill value (GH730)
- Fix Index.format bug causing incorrectly string-formatted Series with datetime indexes (GH726, GH758)
- Fix errors caused by object dtype arrays passed to ols (GH759)
- Fix error where column names lost when passing list of labels to DataFrame.__getitem__, (GH662)
- Fix error whereby top-level week iterator overwrote week instance
- Fix circular reference causing memory leak in sparse array / series / frame, (GH663)
- Fix integer-slicing from integers-as-floats (GH670)
- Fix zero division errors in nanops from object dtype arrays in all NA case (GH676)
- Fix csv encoding when using unicode (GH705, GH717, GH738)
- Fix assumption that each object contains every unique block type in concat, (GH708)
- Fix sortedness check of multiindex in to_panel (GH719, 720)
- Fix that None was not treated as NA in PyObjectHashtable
- Fix hashing dtype because of endianness confusion (GH747, GH748)
- Fix SparseSeries.dropna to return dense Series in case of NA fill value (GH GH730)
- Use map_infer instead of np.vectorize. handle NA sentinels if converter yields numeric array, (GH753)
- Fixes and improvements to DataFrame.rank (GH742)
- Fix catching AttributeError instead of NameError for bottleneck
- Try to cast non-MultiIndex to better dtype when calling reset_index (GH726 GH440)
- Fix #1.QNAN0' float bug on 2.6/win64
- Allow subclasses of dicts in DataFrame constructor, with tests
- Fix problem whereby set_index destroys column multiindex (GH764)

- Hack around bug in generating DateRange from naive DateOffset (GH770)
- Fix bug in DateRange.intersection causing incorrect results with some overlapping ranges (GH771)

26.3 Thanks

- Craig Austin
- Chris Billington
- Marius Cobzarenco
- Mario Gamboa-Cavazos
- Hans-Martin Gaudecker
- Arthur Gerigk
- Yaroslav Halchenko
- Jeff Hammerbacher
- Matt Harrison
- Andreas Hilboll
- Luc Kesters
- Adam Klein
- Gregg Lind
- Solomon Negusse
- Wouter Overmeire
- Christian Prinoth
- Jeff Reback
- Sam Reckoner
- Craig Reeson
- Jan Schulz
- Skipper Seabold
- Ted Square
- Graham Taylor
- Aman Thakral
- Chris Uga
- Dieter Vandenbussche
- Texas P.
- Pinxing Ye
- ... and everyone I forgot

26.3.1 pandas 0.6.1

Release date: 12/13/2011

API Changes

- Rename *names* argument in DataFrame.from_records to *columns*. Add deprecation warning
- Boolean get/set operations on Series with boolean Series will reindex instead of requiring that the indexes be exactly equal (GH429)

New features / modules

- Can pass Series to DataFrame.append with ignore_index=True for appending a single row (GH430)
- Add Spearman and Kendall correlation options to Series.corr and DataFrame.corr (GH428)
- Add new *get_value* and *set_value* methods to Series, DataFrame, and Panel to very low-overhead access to scalar elements. df.get_value(row, column) is about 3x faster than df[column][row] by handling fewer cases (GH437, GH438). Add similar methods to sparse data structures for compatibility
- Add Qt table widget to sandbox (GH435)
- DataFrame.align can accept Series arguments, add axis keyword (GH461)
- Implement new SparseList and SparseArray data structures. SparseSeries now derives from SparseArray (GH463)
- max_columns / max_rows options in set_printoptions (GH453)
- Implement Series.rank and DataFrame.rank, fast versions of scipy.stats.rankdata (GH428)
- Implement DataFrame.from_items alternate constructor (GH444)
- DataFrame.convert_objects method for inferring better dtypes for object columns (GH302)
- Add rolling_corr_pairwise function for computing Panel of correlation matrices (GH189)
- Add *margins* option to *pivot_table* for computing subgroup aggregates (GH GH114)
- Add *Series.from_csv* function (GH482)

Improvements to existing features

- Improve memory usage of *DataFrame.describe* (do not copy data unnecessarily) (GH425)
- Use same formatting function for outputting floating point Series to console as in DataFrame (GH420)
- DataFrame.delevel will try to infer better dtype for new columns (GH440)
- Exclude non-numeric types in DataFrame.{corr, cov}
- Override Index.astype to enable dtype casting (GH412)
- Use same float formatting function for Series.__repr__ (GH420)
- Use available console width to output DataFrame columns (GH453)
- Accept ndarrays when setting items in Panel (GH452)
- Infer console width when printing __repr__ of DataFrame to console (PR GH453)
- Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
- Can pass DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH462)
- Fix performance regression in cross-sectional count in DataFrame, affecting DataFrame.dropna speed
- Column deletion in DataFrame copies no data (computes views on blocks) (GH GH158)

- MultiIndex.get_level_values can take the level name
- More helpful error message when DataFrame.plot fails on one of the columns (GH478)
- Improve performance of DataFrame.{index, columns} attribute lookup

Bug fixes

- Fix O(K^2) memory leak caused by inserting many columns without consolidating, had been present since 0.4.0 (GH467)
- *DataFrame.count* should return Series with zero instead of NA with length-0 axis (GH423)
- Fix Yahoo! Finance API usage in pandas.io.data (GH419, GH427)
- Fix upstream bug causing failure in Series.align with empty Series (GH434)
- Function passed to DataFrame.apply can return a list, as long as it's the right length. Regression from 0.4 (GH432)
- Don't "accidentally" upcast scalar values when indexing using .ix (GH431)
- Fix groupby exception raised with as_index=False and single column selected (GH421)
- Implement DateOffset.__ne__ causing downstream bug (GH456)
- Fix __doc__-related issue when converting py -> pyo with py2exe
- Bug fix in left join Cython code with duplicate monotonic labels
- Fix bug when unstacking multiple levels described in GH451
- Exclude NA values in dtype=object arrays, regression from 0.5.0 (GH469)
- Use Cython map_infer function in DataFrame.applymap to properly infer output type, handle tuple return values and other things that were breaking (GH465)
- Handle floating point index values in HDFStore (GH454)
- Fixed stale column reference bug (cached Series object) caused by type change / item deletion in DataFrame (GH473)
- Index.get_loc should always raise Exception when there are duplicates
- Handle differently-indexed Series input to DataFrame constructor (GH475)
- Omit nuisance columns in multi-groupby with Python function
- Buglet in handling of single grouping in general apply
- Handle type inference properly when passing list of lists or tuples to DataFrame constructor (GH484)
- Preserve Index / MultiIndex names in GroupBy.apply concatenation step (GH GH481)

26.4 Thanks

- Ralph Bean
- Luca Beltrame
- Marius Cobzarenco
- Andreas Hilboll
- Jev Kuznetsov
- Adam Lichtenstein

- Wouter Overmeire
- Fernando Perez
- Nathan Pinger
- Christian Prinoth
- Alex Reyfman
- Joon Ro
- Chang She
- Ted Square
- Chris Uga
- Dieter Vandenbussche

26.4.1 pandas 0.6.0

Release date: 11/25/2011

API Changes

- Arithmetic methods like *sum* will attempt to sum dtype=object values by default instead of excluding them (GH382)

New features / modules

- Add *melt* function to *pandas.core.reshape*
- Add *level* parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Add *head* and *tail* methods to Series, analogous to to DataFrame (PR GH296)
- Add *Series.isin* function which checks if each value is contained in a passed sequence (GH289)
- Add *float_format* option to *Series.to_string*
- Add *skip_footer* (GH291) and *converters* (GH343) options to *read_csv* and *read_table*
- Add proper, tested weighted least squares to standard and panel OLS (GH GH303)
- Add *drop_duplicates* and *duplicated* functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- Implement logical (boolean) operators `&`, `|`, `^` on DataFrame (GH347)
- Add *Series.mad*, mean absolute deviation, matching DataFrame
- Add *QuarterEnd* DateOffset (GH321)
- Add matrix multiplication function *dot* to DataFrame (GH65)
- Add *orient* option to *Panel.from_dict* to ease creation of mixed-type Panels (GH359, GH301)
- Add *DataFrame.from_dict* with similar *orient* option
- Can now pass list of tuples or list of lists to *DataFrame.from_records* for fast conversion to DataFrame (GH357)
- Can pass multiple levels to groupby, e.g. *df.groupby(level=[0, 1])* (GH GH103)
- Can sort by multiple columns in *DataFrame.sort_index* (GH92, GH362)
- Add fast *get_value* and *put_value* methods to DataFrame and micro-performance tweaks (GH360)
- Add *cov* instance methods to Series and DataFrame (GH194, GH362)

- Add bar plot option to *DataFrame.plot* (GH348)
- Add *idxmin* and *idxmax* functions to Series and DataFrame for computing index labels achieving maximum and minimum values (GH286)
- Add *read_clipboard* function for parsing DataFrame from OS clipboard, should work across platforms (GH300)
- Add *nunique* function to Series for counting unique elements (GH297)
- DataFrame constructor will use Series name if no columns passed (GH373)
- Support regular expressions and longer delimiters in read_table/read_csv, but does not handle quoted strings yet (GH364)
- Add *DataFrame.to_html* for formatting DataFrame to HTML (GH387)
- MaskedArray can be passed to DataFrame constructor and masked values will be converted to NaN (GH396)
- Add *DataFrame.boxplot* function (GH368, others)
- Can pass extra args, kwds to DataFrame.apply (GH376)

Improvements to existing features

- Raise more helpful exception if date parsing fails in DateRange (GH298)
- Vastly improved performance of GroupBy on axes with a MultiIndex (GH299)
- Print level names in hierarchical index in Series repr (GH305)
- Return DataFrame when performing GroupBy on selected column and as_index=False (GH308)
- Can pass vector to *on* argument in *DataFrame.join* (GH312)
- Don't show Series name if it's None in the repr, also omit length for short Series (GH317)
- Show legend by default in *DataFrame.plot*, add *legend* boolean flag (GH GH324)
- Significantly improved performance of *Series.order*, which also makes np.unique called on a Series faster (GH327)
- Faster cythonized count by level in Series and DataFrame (GH341)
- Raise exception if dateutil 2.0 installed on Python 2.x runtime (GH346)
- Significant GroupBy performance enhancement with multiple keys with many "empty" combinations
- New Cython vectorized function *map_infer* speeds up *Series.apply* and *Series.map* significantly when passed elementwise Python function, motivated by GH355
- Cythonized *cache_readonly*, resulting in substantial micro-performance enhancements throughout the codebase (GH361)
- Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than *np.apply_along_axis* (GH309)
- Add *raw* option to *DataFrame.apply* for getting better performance when the passed function only requires an ndarray (GH309)
- Improve performance of *MultiIndex.from_tuples*
- Can pass multiple levels to *stack* and *unstack* (GH370)
- Can pass multiple values columns to *pivot_table* (GH381)
- Can call *DataFrame.delevel* with standard Index with name set (GH393)
- Use Series name in GroupBy for result index (GH363)

- Refactor Series/DataFrame stat methods to use common set of NaN-friendly function
- Handle NumPy scalar integers at C level in Cython conversion routines

Bug fixes

- Fix bug in *DataFrame.to_csv* when writing a DataFrame with an index name (GH290)
- DataFrame should clear its Series caches on consolidation, was causing "stale" Series to be returned in some corner cases (GH304)
- DataFrame constructor failed if a column had a list of tuples (GH293)
- Ensure that *Series.apply* always returns a Series and implement *Series.round* (GH314)
- Support boolean columns in Cythonized groupby functions (GH315)
- *DataFrame.describe* should not fail if there are no numeric columns, instead return categorical describe (GH323)
- Fixed bug which could cause columns to be printed in wrong order in *DataFrame.to_string* if specific list of columns passed (GH325)
- Fix legend plotting failure if DataFrame columns are integers (GH326)
- Shift start date back by one month for Yahoo! Finance API in pandas.io.data (GH329)
- Fix *DataFrame.join* failure on unconsolidated inputs (GH331)
- DataFrame.min/max will no longer fail on mixed-type DataFrame (GH337)
- Fix *read_csv* / *read_table* failure when passing list to index_col that is not in ascending order (GH349)
- Fix failure passing Int64Index to Index.union when both are monotonic
- Fix error when passing SparseSeries to (dense) DataFrame constructor
- Added missing bang at top of setup.py (GH352)
- Change *is_monotonic* on MultiIndex so it properly compares the tuples
- Fix MultiIndex outer join logic (GH351)
- Set index name attribute with single-key groupby (GH358)
- Bug fix in reflexive binary addition in Series and DataFrame for non-commutative operations (like string concatenation) (GH353)
- setupegg.py will invoke Cython (GH192)
- Fix block consolidation bug after inserting column into MultiIndex (GH366)
- Fix bug in join operations between Index and Int64Index (GH367)
- Handle min_periods=0 case in moving window functions (GH365)
- Fixed corner cases in DataFrame.apply/pivot with empty DataFrame (GH378)
- Fixed repr exception when Series name is a tuple
- Always return DateRange from *asfreq* (GH390)
- Pass level names to *swaplavel* (GH379)
- Don't lose index names in *MultiIndex.droplevel* (GH394)
- Infer more proper return type in *DataFrame.apply* when no columns or rows depending on whether the passed function is a reduction (GH389)
- Always return NA/NaN from Series.min/max and DataFrame.min/max when all of a row/column/values are NA (GH384)

- Enable partial setting with .ix / advanced indexing (GH397)
- Handle mixed-type DataFrames correctly in unstack, do not lose type information (GH403)
- Fix integer name formatting bug in Index.format and in Series.__repr__
- Handle label types other than string passed to groupby (GH405)
- Fix bug in .ix-based indexing with partial retrieval when a label is not contained in a level
- Index name was not being pickled (GH408)
- Level name should be passed to result index in GroupBy.apply (GH416)

26.5 Thanks

- Craig Austin
- Marius Cobzarenco
- Joel Cross
- Jeff Hammerbacher
- Adam Klein
- Thomas Kluyver
- Jev Kuznetsov
- Kieran O'Mahony
- Wouter Overmeire
- Nathan Pinger
- Christian Prinoth
- Skipper Seabold
- Chang She
- Ted Square
- Aman Thakral
- Chris Uga
- Dieter Vandenbussche
- carljv
- rsamson

26.5.1 pandas 0.5.0

Release date: 10/24/2011

This release of pandas includes a number of API changes (see below) and cleanup of deprecated APIs from pre-0.4.0 releases. There are also bug fixes, new features, numerous significant performance enhancements, and includes a new IPython completer hook to enable tab completion of DataFrame columns accesses as attributes (a new feature).

In addition to the changes listed here from 0.4.3 to 0.5.0, the minor releases 0.4.1, 0.4.2, and 0.4.3 brought some significant new functionality and performance improvements that are worth taking a look at.

Thanks to all for bug reports, contributed patches and generally providing feedback on the library.

API Changes

- *read_table*, *read_csv*, and *ExcelFile.parse* default arguments for *index_col* is now None. To use one or more of the columns as the resulting DataFrame's index, these must be explicitly specified now
- Parsing functions like *read_csv* no longer parse dates by default (GH GH225)
- Removed *weights* option in panel regression which was not doing anything principled (GH155)
- Changed *buffer* argument name in *Series.to_string* to *buf*
- *Series.to_string* and *DataFrame.to_string* now return strings by default instead of printing to sys.stdout
- Deprecated *nanRep* argument in various *to_string* and *to_csv* functions in favor of *na_rep*. Will be removed in 0.6 (GH275)
- Renamed *delimiter* to *sep* in *DataFrame.from_csv* for consistency
- Changed order of *Series.clip* arguments to match those of *numpy.clip* and added (unimplemented) *out* argument so *numpy.clip* can be called on a Series (GH272)
- Series functions renamed (and thus deprecated) in 0.4 series have been removed:
  - *asOf*, use *asof*
  - *toDict*, use *to_dict*
  - *toString*, use *to_string*
  - *toCSV*, use *to_csv*
  - *merge*, use *map*
  - *applymap*, use *apply*
  - *combineFirst*, use *combine_first*
  - *_firstTimeWithValue* use *first_valid_index*
  - *_lastTimeWithValue* use *last_valid_index*
- DataFrame functions renamed / deprecated in 0.4 series have been removed:
  - *asMatrix* method, use *as_matrix* or *values* attribute
  - *combineFirst*, use *combine_first*
  - *getXS*, use *xs*
  - *merge*, use *join*
  - *fromRecords*, use *from_records*
  - *fromcsv*, use *from_csv*
  - *toRecords*, use *to_records*
  - *toDict*, use *to_dict*
  - *toString*, use *to_string*
  - *toCSV*, use *to_csv*
  - *_firstTimeWithValue* use *first_valid_index*
  - *_lastTimeWithValue* use *last_valid_index*
  - *toDataMatrix* is no longer needed

  - *rows()* method, use *index* attribute
  - *cols()* method, use *columns* attribute
  - *dropEmptyRows()*, use *dropna(how='all')*
  - *dropIncompleteRows()*, use *dropna()*
  - *tapply(f)*, use *apply(f, axis=1)*
  - *tgroupby(keyfunc, aggfunc)*, use *groupby* with *axis=1*
- Other outstanding deprecations have been removed:
  - *indexField* argument in *DataFrame.from_records*
  - *missingAtEnd* argument in *Series.order*. Use *na_last* instead
  - *Series.fromValue* classmethod, use regular *Series* constructor instead
  - Functions *parseCSV*, *parseText*, and *parseExcel* methods in *pandas.io.parsers* have been removed
  - *Index.asOfDate* function
  - *Panel.getMinorXS* (use *minor_xs*) and *Panel.getMajorXS* (use *major_xs*)
  - *Panel.toWide*, use *Panel.to_wide* instead

New features / modules

- Added *DataFrame.align* method with standard join options
- Added *parse_dates* option to *read_csv* and *read_table* methods to optionally try to parse dates in the index columns
- Add *nrows*, *chunksize*, and *iterator* arguments to *read_csv* and *read_table*. The last two return a new *TextParser* class capable of lazily iterating through chunks of a flat file (GH242)
- Added ability to join on multiple columns in *DataFrame.join* (GH214)
- Added private *_get_duplicates* function to *Index* for identifying duplicate values more easily
- Added column attribute access to DataFrame, e.g. df.A equivalent to df['A'] if 'A' is a column in the DataFrame (GH213)
- Added IPython tab completion hook for DataFrame columns. (GH233, GH230)
- Implement *Series.describe* for Series containing objects (GH241)
- Add inner join option to *DataFrame.join* when joining on key(s) (GH248)
- Can select set of DataFrame columns by passing a list to *__getitem__* (GH GH253)
- Can use & and | to intersection / union Index objects, respectively (GH GH261)
- Added *pivot_table* convenience function to pandas namespace (GH234)
- Implemented *Panel.rename_axis* function (GH243)
- DataFrame will show index level names in console output
- Implemented *Panel.take*
- Add *set_eng_float_format* function for setting alternate DataFrame floating point string formatting
- Add convenience *set_index* function for creating a DataFrame index from its existing columns

Improvements to existing features

- Major performance improvements in file parsing functions *read_csv* and *read_table*

- Added Cython function for converting tuples to ndarray very fast. Speeds up many MultiIndex-related operations
- File parsing functions like *read_csv* and *read_table* will explicitly check if a parsed index has duplicates and raise a more helpful exception rather than deferring the check until later
- Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance (GH211)
- Improved speed of *DataFrame.xs* on mixed-type DataFrame objects by about 5x, regression from 0.3.0 (GH215)
- With new *DataFrame.align* method, speeding up binary operations between differently-indexed DataFrame objects by 10-25%.
- Significantly sped up conversion of nested dict into DataFrame (GH212)
- Can pass hierarchical index level name to *groupby* instead of the level number if desired (GH223)
- Add support for different delimiters in *DataFrame.to_csv* (GH244)
- Add more helpful error message when importing pandas post-installation from the source directory (GH250)
- Significantly speed up DataFrame *__repr__* and *count* on large mixed-type DataFrame objects
- Better handling of pyx file dependencies in Cython module build (GH271)

Bug fixes

- *read_csv* / *read_table* fixes
  - Be less aggressive about converting float->int in cases of floating point representations of integers like 1.0, 2.0, etc.
  - "True"/"False" will not get correctly converted to boolean
  - Index name attribute will get set when specifying an index column
  - Passing column names should force *header=None* (GH257)
  - Don't modify passed column names when *index_col* is not None (GH258)
  - Can sniff CSV separator in zip file (since seek is not supported, was failing before)
- Worked around matplotlib "bug" in which series[:, np.newaxis] fails. Should be reported upstream to matplotlib (GH224)
- DataFrame.iteritems was not returning Series with the name attribute set. Also neither was DataFrame._series
- Can store datetime.date objects in HDFStore (GH231)
- Index and Series names are now stored in HDFStore
- Fixed problem in which data would get upcasted to object dtype in GroupBy.apply operations (GH237)
- Fixed outer join bug with empty DataFrame (GH238)
- Can create empty Panel (GH239)
- Fix join on single key when passing list with 1 entry (GH246)
- Don't raise Exception on plotting DataFrame with an all-NA column (GH251, GH254)
- Bug min/max errors when called on integer DataFrames (GH241)
- *DataFrame.iteritems* and *DataFrame._series* not assigning name attribute
- Panel.__repr__ raised exception on length-0 major/minor axes
- *DataFrame.join* on key with empty DataFrame produced incorrect columns

- Implemented *MultiIndex.diff* (GH260)
- *Int64Index.take* and *MultiIndex.take* lost name field, fix downstream issue GH262
- Can pass list of tuples to *Series* (GH270)
- Can pass level name to *DataFrame.stack*
- Support set operations between MultiIndex and Index
- Fix many corner cases in MultiIndex set operations - Fix MultiIndex-handling bug with GroupBy.apply when returned groups are not indexed the same
- Fix corner case bugs in DataFrame.apply
- Setting DataFrame index did not cause Series cache to get cleared
- Various int32 -> int64 platform-specific issues
- Don't be too aggressive converting to integer when parsing file with MultiIndex (GH285)
- Fix bug when slicing Series with negative indices before beginning

26.6 Thanks

- Thomas Kluyver
- Daniel Fortunov
- Aman Thakral
- Luca Beltrame
- Wouter Overmeire

26.6.1 pandas 0.4.3

26.7 Release notes

Release date: 10/9/2011

This is largely a bugfix release from 0.4.2 but also includes a handful of new and enhanced features. Also, pandas can now be installed and used on Python 3 (thanks Thomas Kluyver!).

New features / modules

- Python 3 support using 2to3 (GH200, Thomas Kluyver)
- Add *name* attribute to *Series* and added relevant logic and tests. Name now prints as part of *Series.__repr__*
- Add *name* attribute to standard Index so that stacking / unstacking does not discard names and so that indexed DataFrame objects can be reliably round-tripped to flat files, pickle, HDF5, etc.
- Add *isnull* and *notnull* as instance methods on Series (GH209, GH203)

Improvements to existing features

- Skip xlrd-related unit tests if not installed
- *Index.append* and *MultiIndex.append* can accept a list of Index objects to concatenate together

- Altered binary operations on differently-indexed SparseSeries objects to use the integer-based (dense) alignment logic which is faster with a larger number of blocks (GH205)
- Refactored *Series.__repr__* to be a bit more clean and consistent

API Changes

- *Series.describe* and *DataFrame.describe* now bring the 25% and 75% quartiles instead of the 10% and 90% deciles. The other outputs have not changed
- *Series.toString* will print deprecation warning, has been de-camelCased to *to_string*

Bug fixes

- Fix broken interaction between *Index* and *Int64Index* when calling intersection. Implement *Int64Index.intersection*
- *MultiIndex.sortlevel* discarded the level names (GH202)
- Fix bugs in groupby, join, and append due to improper concatenation of *MultiIndex* objects (GH201)
- Fix regression from 0.4.1, *isnull* and *notnull* ceased to work on other kinds of Python scalar objects like *datetime.datetime*
- Raise more helpful exception when attempting to write empty DataFrame or LongPanel to *HDFStore* (GH204)
- Use stdlib csv module to properly escape strings with commas in *DataFrame.to_csv* (GH206, Thomas Kluyver)
- Fix Python ndarray access in Cython code for sparse blocked index integrity check
- Fix bug writing Series to CSV in Python 3 (GH209)
- Miscellaneous Python 3 bugfixes

26.8 Thanks

- Thomas Kluyver
- rsamson

26.8.1 pandas 0.4.2

26.9 Release notes

Release date: 10/3/2011

This is a performance optimization release with several bug fixes. The new Int64Index and new merging / joining Cython code and related Python infrastructure are the main new additions

New features / modules

- Added fast *Int64Index* type with specialized join, union, intersection. Will result in significant performance enhancements for int64-based time series (e.g. using NumPy's datetime64 one day) and also faster operations on DataFrame objects storing record array-like data.
- Refactored *Index* classes to have a *join* method and associated data alignment routines throughout the codebase to be able to leverage optimized joining / merging routines.
- Added *Series.align* method for aligning two series with choice of join method
- Wrote faster Cython data alignment / merging routines resulting in substantial speed increases

- Added *is_monotonic* property to *Index* classes with associated Cython code to evaluate the monotonicity of the *Index* values
- Add method *get_level_values* to *MultiIndex*
- Implemented shallow copy of *BlockManager* object in *DataFrame* internals

Improvements to existing features

- Improved performance of *isnull* and *notnull*, a regression from v0.3.0 (GH187)
- Wrote templating / code generation script to auto-generate Cython code for various functions which need to be available for the 4 major data types used in pandas (float64, bool, object, int64)
- Refactored code related to *DataFrame.join* so that intermediate aligned copies of the data in each *DataFrame* argument do not need to be created. Substantial performance increases result (GH176)
- Substantially improved performance of generic *Index.intersection* and *Index.union*
- Improved performance of *DateRange.union* with overlapping ranges and non-cacheable offsets (like Minute). Implemented analogous fast *DateRange.intersection* for overlapping ranges.
- Implemented *BlockManager.take* resulting in significantly faster *take* performance on mixed-type *DataFrame* objects (GH104)
- Improved performance of *Series.sort_index*
- Significant groupby performance enhancement: removed unnecessary integrity checks in DataFrame internals that were slowing down slicing operations to retrieve groups
- Added informative Exception when passing dict to DataFrame groupby aggregation with axis != 0

API Changes

None

Bug fixes

- Fixed minor unhandled exception in Cython code implementing fast groupby aggregation operations
- Fixed bug in unstacking code manifesting with more than 3 hierarchical levels
- Throw exception when step specified in label-based slice (GH185)
- Fix isnull to correctly work with np.float32. Fix upstream bug described in GH182
- Finish implementation of as_index=False in groupby for DataFrame aggregation (GH181)
- Raise SkipTest for pre-epoch HDFStore failure. Real fix will be sorted out via datetime64 dtype

26.10 Thanks

- Uri Laserson
- Scott Sinclair

26.10.1 pandas 0.4.1

26.11 Release notes

Release date: 9/25/2011

This is primarily a bug fix release but includes some new features and improvements

New features / modules

- Added new *DataFrame* methods *get_dtype_counts* and property *dtypes*
- Setting of values using `.ix` indexing attribute in mixed-type DataFrame objects has been implemented (fixes GH135)
- *read_csv* can read multiple columns into a *MultiIndex*. DataFrame's *to_csv* method will properly write out a *MultiIndex* which can be read back (GH151, thanks to Skipper Seabold)
- Wrote fast time series merging / joining methods in Cython. Will be integrated later into DataFrame.join and related functions
- Added *ignore_index* option to *DataFrame.append* for combining unindexed records stored in a DataFrame

Improvements to existing features

- Some speed enhancements with internal Index type-checking function
- *DataFrame.rename* has a new *copy* parameter which can rename a DataFrame in place
- Enable unstacking by level name (GH142)
- Enable sortlevel to work by level name (GH141)
- *read_csv* can automatically "sniff" other kinds of delimiters using *csv.Sniffer* (GH146)
- Improved speed of unit test suite by about 40%
- Exception will not be raised calling *HDFStore.remove* on non-existent node with where clause
- Optimized *_ensure_index* function resulting in performance savings in type-checking Index objects

API Changes

None

Bug fixes

- Fixed DataFrame constructor bug causing downstream problems (e.g. .copy() failing) when passing a Series as the values along with a column name and index
- Fixed single-key groupby on DataFrame with as_index=False (GH160)
- *Series.shift* was failing on integer Series (GH154)
- *unstack* methods were producing incorrect output in the case of duplicate hierarchical labels. An exception will now be raised (GH147)
- Calling *count* with level argument caused reduceat failure or segfault in earlier NumPy (GH169)
- Fixed *DataFrame.corrwith* to automatically exclude non-numeric data (GH GH144)
- Unicode handling bug fixes in *DataFrame.to_string* (GH138)
- Excluding OLS degenerate unit test case that was causing platform specific failure (GH149)
- Skip blosc-dependent unit tests for PyTables < 2.2 (GH137)
- Calling *copy* on *DateRange* did not copy over attributes to the new object (GH168)
- Fix bug in *HDFStore* in which Panel data could be appended to a Table with different item order, thus resulting in an incorrect result read back

26.12 Thanks

- Yaroslav Halchenko
- Jeff Reback
- Skipper Seabold
- Dan Lovell
- Nick Pentreath

26.12.1 pandas 0.4.0

26.13 Release notes

Release date: 9/12/2011

New features / modules

- *pandas.core.sparse* module: "Sparse" (mostly-NA, or some other fill value) versions of *Series*, *DataFrame*, and *Panel*. For low-density data, this will result in significant performance boosts, and smaller memory footprint. Added *to_sparse* methods to *Series*, *DataFrame*, and *Panel*. See online documentation for more on these
- Fancy indexing operator on Series / DataFrame, e.g. via .ix operator. Both getting and setting of values is supported; however, setting values will only currently work on homogeneously-typed DataFrame objects. Things like:
  - series.ix[[d1, d2, d3]]
  - frame.ix[5:10, ['C', 'B', 'A']], frame.ix[5:10, 'A':'C']
  - frame.ix[date1:date2]
- Significantly enhanced *groupby* functionality
  - Can groupby multiple keys, e.g. df.groupby(['key1', 'key2']). Iteration with multiple groupings products a flattened tuple
  - "Nuisance" columns (non-aggregatable) will automatically be excluded from DataFrame aggregation operations
  - Added automatic "dispatching to Series / DataFrame methods to more easily invoke methods on groups. e.g. s.groupby(crit).std() will work even though *std* is not implemented on the *GroupBy* class
- Hierarchical / multi-level indexing
  - New the *MultiIndex* class. Integrated *MultiIndex* into *Series* and *DataFrame* fancy indexing, slicing, __getitem__ and __setitem, reindexing, etc. Added *level* keyword argument to *groupby* to enable grouping by a level of a *MultiIndex*
- New data reshaping functions: *stack* and *unstack* on DataFrame and Series
  - Integrate with MultiIndex to enable sophisticated reshaping of data
- *Index* objects (labels for axes) are now capable of holding tuples
- *Series.describe*, *DataFrame.describe*: produces an R-like table of summary statistics about each data column
- *DataFrame.quantile*, *Series.quantile* for computing sample quantiles of data across requested axis

- Added general *DataFrame.dropna* method to replace *dropIncompleteRows* and *dropEmptyRows*, deprecated those.
- *Series* arithmetic methods with optional fill_value for missing data, e.g. a.add(b, fill_value=0). If a location is missing for both it will still be missing in the result though.
- fill_value option has been added to *DataFrame*.{add, mul, sub, div} methods similar to *Series*
- Boolean indexing with *DataFrame* objects: data[data > 0.1] = 0.1 or data[data> other] = 1.
- *pytz* / tzinfo support in *DateRange*
  - *tz_localize*, *tz_normalize*, and *tz_validate* methods added
- Added *ExcelFile* class to *pandas.io.parsers* for parsing multiple sheets out of a single Excel 2003 document
- *GroupBy* aggregations can now optionally *broadcast*, e.g. produce an object of the same size with the aggregated value propagated
- Added *select* function in all data structures: reindex axis based on arbitrary criterion (function returning boolean value), e.g. frame.select(lambda x: 'foo' in x, axis=1)
- *DataFrame.consolidate* method, API function relating to redesigned internals
- *DataFrame.insert* method for inserting column at a specified location rather than the default __setitem__ behavior (which puts it at the end)
- *HDFStore* class in *pandas.io.pytables* has been largely rewritten using patches from Jeff Reback from others. It now supports mixed-type *DataFrame* and *Series* data and can store *Panel* objects. It also has the option to query *DataFrame* and *Panel* data. Loading data from legacy *HDFStore* files is supported explicitly in the code
- Added *set_printoptions* method to modify appearance of DataFrame tabular output
- *rolling_quantile* functions; a moving version of *Series.quantile* / *DataFrame.quantile*
- Generic *rolling_apply* moving window function
- New *drop* method added to *Series*, *DataFrame*, etc. which can drop a set of labels from an axis, producing a new object
- *reindex* methods now sport a *copy* option so that data is not forced to be copied then the resulting object is indexed the same
- Added *sort_index* methods to Series and Panel. Renamed *DataFrame.sort* to *sort_index*. Leaving *DataFrame.sort* for now.
- Added `skipna` option to statistical instance methods on all the data structures
- *pandas.io.data* module providing a consistent interface for reading time series data from several different sources

Improvements to existing features

- The 2-dimensional *DataFrame* and *DataMatrix* classes have been extensively redesigned internally into a single class *DataFrame*, preserving where possible their optimal performance characteristics. This should reduce confusion from users about which class to use.
  - Note that under the hood there is a new essentially "lazy evaluation" scheme within respect to adding columns to DataFrame. During some operations, like-typed blocks will be "consolidated" but not before.
- *DataFrame* accessing columns repeatedly is now significantly faster than *DataMatrix* used to be in 0.3.0 due to an internal Series caching mechanism (which are all views on the underlying data)
- Column ordering for mixed type data is now completely consistent in *DataFrame*. In prior releases, there was inconsistent column ordering in *DataMatrix*
- Improved console / string formatting of DataMatrix with negative numbers

- Improved tabular data parsing functions, *read_table* and *read_csv*:
  - Added *skiprows* and *na_values* arguments to *pandas.io.parsers* functions for more flexible IO
  - *parseCSV* / *read_csv* functions and others in *pandas.io.parsers* now can take a list of custom NA values, and also a list of rows to skip
- Can slice *DataFrame* and get a view of the data (when homogeneously typed), e.g. frame.xs(idx, copy=False) or frame.ix[idx]
- Many speed optimizations throughout *Series* and *DataFrame*
- Eager evaluation of groups when calling `groupby` functions, so if there is an exception with the grouping function it will raised immediately versus sometime later on when the groups are needed
- *datetools.WeekOfMonth* offset can be parameterized with *n* different than 1 or -1.
- Statistical methods on DataFrame like *mean*, *std*, *var*, *skew* will now ignore non-numerical data. Before a not very useful error message was generated. A flag *numeric_only* has been added to *DataFrame.sum* and *DataFrame.count* to enable this behavior in those methods if so desired (disabled by default)
- *DataFrame.pivot* generalized to enable pivoting multiple columns into a *DataFrame* with hierarchical columns
- *DataFrame* constructor can accept structured / record arrays
- *Panel* constructor can accept a dict of DataFrame-like objects. Do not need to use *from_dict* anymore (*from_dict* is there to stay, though).

API Changes

- The *DataMatrix* variable now refers to *DataFrame*, will be removed within two releases
- *WidePanel* is now known as *Panel*. The *WidePanel* variable in the pandas namespace now refers to the renamed *Panel* class
- *LongPanel* and *Panel* / *WidePanel* now no longer have a common subclass. *LongPanel* is now a subclass of *DataFrame* having a number of additional methods and a hierarchical index instead of the old *LongPanelIndex* object, which has been removed. Legacy *LongPanel* pickles may not load properly
- Cython is now required to build *pandas* from a development branch. This was done to avoid continuing to check in cythonized C files into source control. Builds from released source distributions will not require Cython
- Cython code has been moved up to a top level *pandas/src* directory. Cython extension modules have been renamed and promoted from the *lib* subpackage to the top level, i.e.
  - *pandas.lib.tseries* -> *pandas._tseries*
  - *pandas.lib.sparse* -> *pandas._sparse*
- *DataFrame* pickling format has changed. Backwards compatibility for legacy pickles is provided, but it's recommended to consider PyTables-based *HDFStore* for storing data with a longer expected shelf life
- A *copy* argument has been added to the *DataFrame* constructor to avoid unnecessary copying of data. Data is no longer copied by default when passed into the constructor
- Handling of boolean dtype in *DataFrame* has been improved to support storage of boolean data with NA / NaN values. Before it was being converted to float64 so this should not (in theory) cause API breakage
- To optimize performance, Index objects now only check that their labels are unique when uniqueness matters (i.e. when someone goes to perform a lookup). This is a potentially dangerous tradeoff, but will lead to much better performance in many places (like groupby).
- Boolean indexing using Series must now have the same indices (labels)
- Backwards compatibility support for begin/end/nPeriods keyword arguments in DateRange class has been removed

- More intuitive / shorter filling aliases *ffill* (for *pad*) and *bfill* (for *backfill*) have been added to the functions that use them: *reindex*, *asfreq*, *fillna*.
- *pandas.core.mixins* code moved to *pandas.core.generic*
- *buffer* keyword arguments (e.g. *DataFrame.toString*) renamed to *buf* to avoid using Python built-in name
- *DataFrame.rows()* removed (use *DataFrame.index*)
- Added deprecation warning to *DataFrame.cols()*, to be removed in next release
- *DataFrame* deprecations and de-camelCasing: *merge*, *asMatrix*, *toDataMatrix*, *_firstTimeWithValue*, *_lastTimeWithValue*, *toRecords*, *fromRecords*, *tgroupby*, *toString*
- *pandas.io.parsers* method deprecations
  - *parseCSV* is now *read_csv* and keyword arguments have been de-camelCased
  - *parseText* is now *read_table*
  - *parseExcel* is replaced by the *ExcelFile* class and its *parse* method
- *fillMethod* arguments (deprecated in prior release) removed, should be replaced with *method*
- *Series.fill*, *DataFrame.fill*, and *Panel.fill* removed, use *fillna* instead
- *groupby* functions now exclude NA / NaN values from the list of groups. This matches R behavior with NAs in factors e.g. with the *tapply* function
- Removed *parseText*, *parseCSV* and *parseExcel* from pandas namespace
- *Series.combineFunc* renamed to *Series.combine* and made a bit more general with a *fill_value* keyword argument defaulting to NaN
- Removed *pandas.core.pytools* module. Code has been moved to *pandas.core.common*
- Tacked on *groupName* attribute for groups in GroupBy renamed to *name*
- Panel/LongPanel *dims* attribute renamed to *shape* to be more conformant
- Slicing a *Series* returns a view now
- More Series deprecations / renaming: *toCSV* to *to_csv*, *asOf* to *asof*, *merge* to *map*, *applymap* to *apply*, *toDict* to *to_dict*, *combineFirst* to *combine_first*. Will print *FutureWarning*.
- *DataFrame.to_csv* does not write an "index" column label by default anymore since the output file can be read back without it. However, there is a new `index_label` argument. So you can do `index_label='index'` to emulate the old behavior
- *datetools.Week* argument renamed from *dayOfWeek* to *weekday*
- *timeRule* argument in *shift* has been deprecated in favor of using the *offset* argument for everything. So you can still pass a time rule string to *offset*
- Added optional *encoding* argument to *read_csv*, *read_table*, *to_csv*, *from_csv* to handle unicode in python 2.x

Bug fixes

- Column ordering in *pandas.io.parsers.parseCSV* will match CSV in the presence of mixed-type data
- Fixed handling of Excel 2003 dates in *pandas.io.parsers*
- *DateRange* caching was happening with high resolution *DateOffset* objects, e.g. *DateOffset(seconds=1)*. This has been fixed
- Fixed __truediv__ issue in *DataFrame*
- Fixed *DataFrame.toCSV* bug preventing IO round trips in some cases

- Fixed bug in *Series.plot* causing matplotlib to barf in exceptional cases
- Disabled *Index* objects from being hashable, like ndarrays
- Added *__ne__* implementation to *Index* so that operations like ts[ts != idx] will work
- Added *__ne__* implementation to *DataFrame*
- Bug / unintuitive result when calling *fillna* on unordered labels
- Bug calling *sum* on boolean DataFrame
- Bug fix when creating a DataFrame from a dict with scalar values
- Series.{sum, mean, std, ...} now return NA/NaN when the whole Series is NA
- NumPy 1.4 through 1.6 compatibility fixes
- Fixed bug in bias correction in *rolling_cov*, was affecting *rolling_corr* too
- R-square value was incorrect in the presence of fixed and time effects in the *PanelOLS* classes
- *HDFStore* can handle duplicates in table format, will take

26.14 Thanks

- Joon Ro
- Michael Pennington
- Chris Uga
- Chris Withers
- Jeff Reback
- Ted Square
- Craig Austin
- William Ferreira
- Daniel Fortunov
- Tony Roberts
- Martin Felder
- John Marino
- Tim McNamara
- Justin Berka
- Dieter Vandenbussche
- Shane Conway
- Skipper Seabold
- Chris Jordan-Squire

26.14.1 pandas 0.3.0

26.15 Release notes

Release date: February 20, 2011

New features / modules

- *corrwith* function to compute column- or row-wise correlations between two DataFrame objects
- Can boolean-index DataFrame objects, e.g. df[df > 2] = 2, px[px > last_px] = 0
- Added comparison magic methods (__lt__, __gt__, etc.)
- Flexible explicit arithmetic methods (add, mul, sub, div, etc.)
- Added *reindex_like* method
- Added *reindex_like* method to WidePanel
- Convenience functions for accessing SQL-like databases in *pandas.io.sql* module
- Added (still experimental) HDFStore class for storing pandas data structures using HDF5 / PyTables in *pandas.io.pytables* module
- Added WeekOfMonth date offset
- *pandas.rpy* (experimental) module created, provide some interfacing / conversion between rpy2 and pandas

Improvements

- Unit test coverage: 100% line coverage of core data structures
- Speed enhancement to rolling_{median, max, min}
- **Column ordering between DataFrame and DataMatrix is now consistent: before** DataFrame would not respect column order
- **Improved {Series, DataFrame}.plot methods to be more flexible (can pass** matplotlib Axis arguments, plot DataFrame columns in multiple subplots, etc.)

API Changes

- Exponentially-weighted moment functions in *pandas.stats.moments* have a more consistent API and accept a min_periods argument like their regular moving counterparts.
- **fillMethod** argument in Series, DataFrame changed to **method**, *FutureWarning* added.
- **fill** method in Series, DataFrame/DataMatrix, WidePanel renamed to **fillna**, *FutureWarning* added to **fill**
- Renamed **DataFrame.getXS** to **xs**, *FutureWarning* added
- Removed **cap** and **floor** functions from DataFrame, renamed to **clip_upper** and **clip_lower** for consistency with NumPy

Bug fixes

- Fixed bug in IndexableSkiplist Cython code that was breaking rolling_max function
- Numerous numpy.int64-related indexing fixes
- Several NumPy 1.4.0 NaN-handling fixes
- Bug fixes to pandas.io.parsers.parseCSV
- Fixed *DateRange* caching issue with unusual date offsets

- Fixed bug in *DateRange.union*
- Fixed corner case in *IndexableSkiplist* implementation

PYTHON MODULE INDEX

PYTHON MODULE INDEX