pandas: powerful Python data analysis toolkit

Release 0.25.3

Wes McKinney& PyData Development Team

Nov 02, 2019

CONTENTS

Date: Nov 02, 2019 **Version**: 0.25.3

Download documentation: PDF Version | Zipped HTML

Useful links: Binary Installers | Source Repository | Issues & Ideas | Q&A Support | Mailing List

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

See the overview for more detail about whats in the library.

CHAPTER ONE

WHATS NEW IN 0.25.2 (OCTOBER 15, 2019)

These are the changes in pandas 0.25.2. See release for a full changelog including other versions of pandas.

Note: Pandas 0.25.2 adds compatibility for Python 3.8 (GH28147).

1.1 Bug fixes

1.1.1 Indexing

- Fix regression in `DataFrame.reindex()` not following the `limit` argument (GH28631).
- Fix regression in `RangeIndex.get_indexer()` for decreasing `RangeIndex` where target values may be improperly identified as missing/present (GH28678)

1.1.2 I/O

- Fix regression in notebook display where `<th>` tags were missing for `DataFrame.index` values (GH28204).
- Regression in `to_csv()` where writing a `Series` or `DataFrame` indexed by an `IntervalIndex` would incorrectly raise a `TypeError` (GH28210)
- Fix `to_csv()` with `ExtensionArray` with list-like values (GH28840).

1.1.3 Groupby/resample/rolling

- Bug incorrectly raising an `IndexError` when passing a list of quantiles to *`pandas.core.groupby.DataFrameGroupBy.quantile()`* (GH28113).
- Bug in `pandas.core.groupby.GroupBy.shift()`, *`pandas.core.groupby.GroupBy.bfill()`* and *`pandas.core.groupby.GroupBy.ffill()`* where timezone information would be dropped (GH19995, GH27992)

- Compatibility with Python 3.8 in `DataFrame.query()` (GH27261)
- Fix to ensure that tab-completion in an IPython console does not raise warnings for deprecated attributes (GH27900).

1.2 Contributors

A total of 8 people contributed patches to this release. People with a + by their names contributed a patch for the first time.

- Felix Divo +
- Jeremy Schendel
- Joris Van den Bossche
- MeeseeksMachine
- Tom Augspurger
- Will Ayd
- William Ayd
- jbrockmendel

{{ header }}

CHAPTER

TWO

INSTALLATION

The easiest way to install pandas is to install it as part of the Anaconda distribution, a cross platform distribution for data analysis and scientific computing. This is the recommended installation method for most users.

Instructions for installing from source, PyPI, ActivePython, various Linux distributions, or a development version are also provided.

2.1 Python version support

Officially Python 3.5.3 and above, 3.6, 3.7, and 3.8.

2.2 Installing pandas

2.2.1 Installing with Anaconda

Installing pandas and the rest of the NumPy and SciPy stack can be a little difficult for inexperienced users.

The simplest way to install not only pandas, but Python and the most popular packages that make up the SciPy stack (IPython, NumPy, Matplotlib,) is with Anaconda, a cross-platform (Linux, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

After running the installer, the user will have access to pandas and the rest of the SciPy stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for Anaconda can be found here.

A full list of the packages available as part of the Anaconda distribution can be found here.

Another advantage to installing Anaconda is that you dont need admin rights to install it. Anaconda can install in the users home directory, which makes it trivial to delete Anaconda if you decide (just delete that folder).

2.2.2 Installing with Miniconda

The previous section outlined how to get pandas installed as part of the Anaconda distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with Miniconda may be a better solution.

Conda is the package manager that the Anaconda distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

Miniconda allows you to create a minimal self contained Python installation, and then use the Conda command to install additional packages.

First you will need Conda to be installed and downloading and running the Miniconda will do this for you. The installer can be found here

The next step is to create a new conda environment. A conda environment is like a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:

```
source activate name_of_my_env
```

On Windows the command is:

```
activate name_of_my_env
```

The final step required is to install pandas. This can be done with the following command:

```
conda install pandas
```

To install a specific pandas version:

```
conda install pandas=0.20.3
```

To install other packages, IPython for example:

```
conda install ipython
```

To install the full Anaconda distribution:

```
conda install anaconda
```

If you need packages that are available to pip but not conda, then install pip, and then use pip to install those packages:

```
conda install pip
pip install django
```

2.2.3 Installing from PyPI

pandas can be installed via pip from PyPI.

```
pip install pandas
```

2.2.4 Installing with ActivePython

Installation instructions for ActivePython can be found here. Versions 2.7 and 3.5 include pandas.

2.2.5 Installing using your Linux distributions package manager.

The commands in this table will install pandas for Python 3 from your distribution. To install pandas for Python 2, you may need to use the `python-pandas` package.

Distribution	Status	Download / Repository Link	Install method
Debian	stable	official Debian repository	`sudo apt-get install python3-pandas`
Debian & Ubuntu	unstable (latest packages)	NeuroDebian	`sudo apt-get install python3-pandas`
Ubuntu	stable	official Ubuntu repository	`sudo apt-get install python3-pandas`
OpenSuse	stable	OpenSuse Repository	`zypper in python3-pandas`
Fedora	stable	official Fedora repository	`dnf install python3-pandas`
Centos/RHEL	stable	EPEL repository	`yum install python3-pandas`

However, the packages in the linux package managers are often a few versions behind, so to get the newest version of pandas, its recommended to install using the `pip` or `conda` methods described above.

2.2.6 Installing from source

See the *contributing guide* for complete instructions on building from the git source tree. Further, see *creating a development environment* if you wish to create a *pandas* development environment.

2.3 Running the test suite

pandas is equipped with an exhaustive set of unit tests, covering about 97% of the code base as of this writing. To run it on your machine to verify that everything is working (and that you have all of the dependencies, soft and hard, installed), make sure you have pytest >= 4.0.2 and Hypothesis >= 3.58, then run:

```
>>> pd.test()
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-
↪packages\pandas
============================= test session starts =============================
platform win32 -- Python 3.6.2, pytest-3.6.0, py-1.4.34, pluggy-0.4.0
rootdir: C:\Users\TP\Documents\Python\pandasdev\pandas, inifile: setup.cfg
collected 12145 items / 3 skipped

..................................................................S......
........S................................................................
.........................................................................

==================== 12130 passed, 12 skipped in 368.339 seconds =====================
```

2.4 Dependencies

Package	Minimum supported version
setuptools	24.2.0
NumPy	1.13.3
python-dateutil	2.6.1
pytz	2017.2

2.4.1 Recommended dependencies

- numexpr: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups. If installed, must be Version 2.6.2 or higher.
- bottleneck: for accelerating certain types of `nan` evaluations. `bottleneck` uses specialized cython routines to achieve large speedups. If installed, must be Version 1.2.1 or higher.

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

2.4.2 Optional dependencies

Pandas has many optional dependencies that are only used for specific methods. For example, *`pandas.read_hdf()`* requires the `pytables` package. If the optional dependency is not installed, pandas will raise an `ImportError` when the method requiring that dependency is called.

Dependency	Minimum Version	Notes
BeautifulSoup4	4.6.0	HTML parser for read_html (see *note*)
Jinja2		Conditional formatting with DataFrame.style
PyQt4		Clipboard I/O
PyQt5		Clipboard I/O
PyTables	3.4.2	HDF5-based reading / writing
SQLAlchemy	1.1.4	SQL support for databases other than sqlite
SciPy	0.19.0	Miscellaneous statistical functions
XLsxWriter	0.9.8	Excel writing
blosc		Compression for msgpack
fastparquet	0.2.1	Parquet reading / writing
gcsfs	0.2.2	Google Cloud Storage access
html5lib		HTML parser for read_html (see *note*)
lxml	3.8.0	HTML parser for read_html (see *note*)
matplotlib	2.2.2	Visualization
openpyxl	2.4.8	Reading / writing for xlsx files
pandas-gbq	0.8.0	Google Big Query access
psycopg2		PostgreSQL engine for sqlalchemy
pyarrow	0.9.0	Parquet and feather reading / writing
pymysql	0.7.11	MySQL engine for sqlalchemy
pyreadstat		SPSS files (.sav) reading
pytables	3.4.2	HDF5 reading / writing
qtpy		Clipboard I/O
s3fs	0.0.8	Amazon S3 access
xarray	0.8.2	pandas-like API for N-dimensional data
xclip		Clipboard I/O on linux
xlrd	1.1.0	Excel reading
xlwt	1.2.0	Excel writing
xsel		Clipboard I/O on linux
zlib		Compression for msgpack

Optional dependencies for parsing HTML

One of the following combinations of libraries is needed to use the top-level `read_html()` function:

Changed in version 0.23.0.

- BeautifulSoup4 and html5lib
- BeautifulSoup4 and lxml
- BeautifulSoup4 and html5lib and lxml
- Only lxml, although see *HTML Table Parsing* 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 Table Parsing gotchas*. It explains issues surrounding the installation and usage of the above three libraries.

{{ header }}

CHAPTER

THREE

GETTING STARTED

{{ header }}

3.1 Package overview

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 Rs `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.

3.1.1 Data structures

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

Why more than one 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 Series is a container for scalars. 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. 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
```

3.1.2 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 we like to **favor immutability** where sensible.

3.1.3 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.

3.1.4 Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to all of our contributors.

If youre interested in contributing, please visit the *contributing guide*.

pandas is a NumFOCUS sponsored project. This will help ensure the success of development of pandas as a world-class open-source project, and makes it possible to donate to the project.

3.1.5 Project governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in Project Governance documents. The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).

3.1.6 Development team

The list of the Core Team members and more detailed information can be found on the peoples page of the governance repo.

3.1.7 Institutional partners

The information about current institutional partners can be found on pandas website page.

3.1.8 License

```
BSD 3-Clause License

Copyright (c) 2008-2012, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData␣
↪Development Team
All rights reserved.

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

(continues on next page)

(continued from previous page)

```
* 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 its
  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 HOLDERS 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 HOLDER 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.
```

{{ header }}

3.2 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 numpy as np

In [2]: import pandas as pd
```

3.2.1 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
Out[4]:
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
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
Out[6]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

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

In [8]: df
Out[8]:
                   A         B         C         D
2013-01-01  1.832747  1.515386  1.793547 -0.360634
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
2013-01-05  0.139488 -0.259328  1.082034 -0.902452
2013-01-06 -0.130327 -0.372906  1.072236 -0.424347
```

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=list(range(4)), dtype='float32'),
   ...:                     'D': np.array([3] * 4, dtype='int32'),
   ...:                     'E': pd.Categorical(["test", "train", "test", "train"]),
   ...:                     'F': 'foo'})
   ...:

In [10]: df2
Out[10]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

The columns of the resulting `DataFrame` have different *dtypes*.

```
In [11]: df2.dtypes
Out[11]:
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

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

```
In [12]: df2.<TAB>  # noqa: E225, E999
df2.A                  df2.bool
df2.abs                df2.boxplot
df2.add                df2.C
```

(continues on next page)

(continued from previous page)

```
df2.add_prefix          df2.clip
df2.add_suffix          df2.clip_lower
df2.align               df2.clip_upper
df2.all                 df2.columns
df2.any                 df2.combine
df2.append              df2.combine_first
df2.apply               df2.compound
df2.applymap            df2.consolidate
df2.D
```

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

3.2.2 Viewing data

See the *Basics section*.

Here is how to view the top and bottom rows of the frame:

```
In [13]: df.head()
Out[13]:
                   A         B         C         D
2013-01-01  1.832747  1.515386  1.793547 -0.360634
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
2013-01-05  0.139488 -0.259328  1.082034 -0.902452

In [14]: df.tail(3)
Out[14]:
                   A         B         C         D
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
2013-01-05  0.139488 -0.259328  1.082034 -0.902452
2013-01-06 -0.130327 -0.372906  1.072236 -0.424347
```

Display the index, columns:

```
In [15]: df.index
Out[15]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [16]: df.columns
Out[16]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

`DataFrame.to_numpy()` gives a NumPy representation of the underlying data. Note that this can be an expensive operation when your `DataFrame` has columns with different data types, which comes down to a fundamental difference between pandas and NumPy: **NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column**. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. This may end up being `object`, which requires casting every value to a Python object.

For `df`, our `DataFrame` of all floating-point values, `DataFrame.to_numpy()` is fast and doesnt require copying data.

```
In [17]: df.to_numpy()
Out[17]:
array([[ 1.83274697e+00,  1.51538609e+00,  1.79354724e+00,
        -3.60634458e-01],
       [-9.13435768e-01,  3.51414290e-02,  3.43748191e+00,
        -1.10691447e+00],
       [-1.32365020e+00,  4.27354647e-01,  8.35343007e-01,
        -6.97782589e-04],
       [ 5.09859417e-01, -2.76958615e+00,  1.00052083e+00,
        -8.65747849e-01],
       [ 1.39488428e-01, -2.59327906e-01,  1.08203428e+00,
        -9.02451725e-01],
       [-1.30327388e-01, -3.72906082e-01,  1.07223611e+00,
        -4.24346700e-01]])
```

For `df2`, the `DataFrame` with multiple dtypes, `DataFrame.to_numpy()` is relatively expensive.

```
In [18]: df2.to_numpy()
Out[18]:
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

Note: `DataFrame.to_numpy()` does *not* include the index or column labels in the output.

`describe()` shows a quick statistic summary of your data:

```
In [19]: df.describe()
Out[19]:
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.019114 -0.237323  1.536861 -0.610132
std    1.117102  1.415574  0.988006  0.416115
min   -1.323650 -2.769586  0.835343 -1.106914
25%   -0.717659 -0.344512  1.018450 -0.893276
50%    0.004581 -0.112093  1.077135 -0.645047
75%    0.417267  0.329301  1.615669 -0.376563
max    1.832747  1.515386  3.437482 -0.000698
```

Transposing your data:

```
In [20]: df.T
Out[20]:
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A    1.832747   -0.913436   -1.323650    0.509859    0.139488   -0.130327
B    1.515386    0.035141    0.427355   -2.769586   -0.259328   -0.372906
C    1.793547    3.437482    0.835343    1.000521    1.082034    1.072236
D   -0.360634   -1.106914   -0.000698   -0.865748   -0.902452   -0.424347
```

Sorting by an axis:

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

(continues on next page)

(continued from previous page)

```
                   D         C         B         A
2013-01-01 -0.360634  1.793547  1.515386  1.832747
2013-01-02 -1.106914  3.437482  0.035141 -0.913436
2013-01-03 -0.000698  0.835343  0.427355 -1.323650
2013-01-04 -0.865748  1.000521 -2.769586  0.509859
2013-01-05 -0.902452  1.082034 -0.259328  0.139488
2013-01-06 -0.424347  1.072236 -0.372906 -0.130327
```

Sorting by values:

```
In [22]: df.sort_values(by='B')
Out[22]: 
                   A         B         C         D
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
2013-01-06 -0.130327 -0.372906  1.072236 -0.424347
2013-01-05  0.139488 -0.259328  1.082034 -0.902452
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698
2013-01-01  1.832747  1.515386  1.793547 -0.360634
```

3.2.3 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` and `.iloc`.

See the indexing documentation *Indexing and Selecting Data* and *MultiIndex / Advanced Indexing*.

Getting

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

```
In [23]: df['A']
Out[23]: 
2013-01-01    1.832747
2013-01-02   -0.913436
2013-01-03   -1.323650
2013-01-04    0.509859
2013-01-05    0.139488
2013-01-06   -0.130327
Freq: D, Name: A, dtype: float64
```

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

```
In [24]: df[0:3]
Out[24]: 
                   A         B         C         D
2013-01-01  1.832747  1.515386  1.793547 -0.360634
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698

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

```
Out[25]:
                   A         B         C         D
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
```

Selection by label

See more in *Selection by Label*.

For getting a cross section using a label:

```
In [26]: df.loc[dates[0]]
Out[26]:
A    1.832747
B    1.515386
C    1.793547
D   -0.360634
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label:

```
In [27]: df.loc[:, ['A', 'B']]
Out[27]:
                   A         B
2013-01-01  1.832747  1.515386
2013-01-02 -0.913436  0.035141
2013-01-03 -1.323650  0.427355
2013-01-04  0.509859 -2.769586
2013-01-05  0.139488 -0.259328
2013-01-06 -0.130327 -0.372906
```

Showing label slicing, both endpoints are *included*:

```
In [28]: df.loc['20130102':'20130104', ['A', 'B']]
Out[28]:
                   A         B
2013-01-02 -0.913436  0.035141
2013-01-03 -1.323650  0.427355
2013-01-04  0.509859 -2.769586
```

Reduction in the dimensions of the returned object:

```
In [29]: df.loc['20130102', ['A', 'B']]
Out[29]:
A   -0.913436
B    0.035141
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], 'A']
Out[30]: 1.8327469709663295
```

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

```
In [31]: df.at[dates[0], 'A']
Out[31]: 1.8327469709663295
```

Selection by position

See more in *Selection by Position*.

Select via the position of the passed integers:

```
In [32]: df.iloc[3]
Out[32]:
A    0.509859
B   -2.769586
C    1.000521
D   -0.865748
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python:

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
                   A         B
2013-01-04  0.509859 -2.769586
2013-01-05  0.139488 -0.259328
```

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

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
                   A         C
2013-01-02 -0.913436  3.437482
2013-01-03 -1.323650  0.835343
2013-01-05  0.139488  1.082034
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
                   A         B         C         D
2013-01-02 -0.913436  0.035141  3.437482 -1.106914
2013-01-03 -1.323650  0.427355  0.835343 -0.000698
```

For slicing columns explicitly:

```
In [36]: df.iloc[:, 1:3]
Out[36]:
                   B         C
2013-01-01  1.515386  1.793547
2013-01-02  0.035141  3.437482
2013-01-03  0.427355  0.835343
2013-01-04 -2.769586  1.000521
2013-01-05 -0.259328  1.082034
2013-01-06 -0.372906  1.072236
```

For getting a value explicitly:

```
In [37]: df.iloc[1, 1]
Out[37]: 0.03514142900432859
```

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

```
In [38]: df.iat[1, 1]
Out[38]: 0.03514142900432859
```

Boolean indexing

Using a single columns values to select data.

```
In [39]: df[df.A > 0]
Out[39]: 
                   A         B         C         D
2013-01-01  1.832747  1.515386  1.793547 -0.360634
2013-01-04  0.509859 -2.769586  1.000521 -0.865748
2013-01-05  0.139488 -0.259328  1.082034 -0.902452
```

Selecting values from a DataFrame where a boolean condition is met.

```
In [40]: df[df > 0]
Out[40]: 
                   A         B         C   D
2013-01-01  1.832747  1.515386  1.793547 NaN
2013-01-02       NaN  0.035141  3.437482 NaN
2013-01-03       NaN  0.427355  0.835343 NaN
2013-01-04  0.509859       NaN  1.000521 NaN
2013-01-05  0.139488       NaN  1.082034 NaN
2013-01-06       NaN       NaN  1.072236 NaN
```

Using the `isin()` method for filtering:

```
In [41]: df2 = df.copy()

In [42]: df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']

In [43]: df2
Out[43]: 
                   A         B         C         D      E
2013-01-01  1.832747  1.515386  1.793547 -0.360634    one
2013-01-02 -0.913436  0.035141  3.437482 -1.106914    one
2013-01-03 -1.323650  0.427355  0.835343 -0.000698    two
2013-01-04  0.509859 -2.769586  1.000521 -0.865748  three
2013-01-05  0.139488 -0.259328  1.082034 -0.902452   four
2013-01-06 -0.130327 -0.372906  1.072236 -0.424347  three

In [44]: df2[df2['E'].isin(['two', 'four'])]
Out[44]: 
                   A         B         C         D     E
2013-01-03 -1.323650  0.427355  0.835343 -0.000698   two
2013-01-05  0.139488 -0.259328  1.082034 -0.902452  four
```

Setting

Setting a new column automatically aligns the data by the indexes.

```
In [45]: s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range('20130102',␣
↪periods=6))

In [46]: s1
Out[46]:
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

In [47]: df['F'] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], 'A'] = 0
```

Setting values by position:

```
In [49]: df.iat[0, 1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, 'D'] = np.array([5] * len(df))
```

The result of the prior setting operations.

```
In [51]: df
Out[51]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000  1.793547  5  NaN
2013-01-02 -0.913436  0.035141  3.437482  5  1.0
2013-01-03 -1.323650  0.427355  0.835343  5  2.0
2013-01-04  0.509859 -2.769586  1.000521  5  3.0
2013-01-05  0.139488 -0.259328  1.082034  5  4.0
2013-01-06 -0.130327 -0.372906  1.072236  5  5.0
```

A `where` operation with setting.

```
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -1.793547 -5  NaN
2013-01-02 -0.913436 -0.035141 -3.437482 -5 -1.0
2013-01-03 -1.323650 -0.427355 -0.835343 -5 -2.0
2013-01-04 -0.509859 -2.769586 -1.000521 -5 -3.0
2013-01-05 -0.139488 -0.259328 -1.082034 -5 -4.0
2013-01-06 -0.130327 -0.372906 -1.072236 -5 -5.0
```

3.2.4 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 [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])

In [56]: df1.loc[dates[0]:dates[1], 'E'] = 1

In [57]: df1
Out[57]: 
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000  1.793547  5  NaN  1.0
2013-01-02 -0.913436  0.035141  3.437482  5  1.0  1.0
2013-01-03 -1.323650  0.427355  0.835343  5  2.0  NaN
2013-01-04  0.509859 -2.769586  1.000521  5  3.0  NaN
```

To drop any rows that have missing data.

```
In [58]: df1.dropna(how='any')
Out[58]: 
                   A         B         C  D    F    E
2013-01-02 -0.913436  0.035141  3.437482  5  1.0  1.0
```

Filling missing data.

```
In [59]: df1.fillna(value=5)
Out[59]: 
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000  1.793547  5  5.0  1.0
2013-01-02 -0.913436  0.035141  3.437482  5  1.0  1.0
2013-01-03 -1.323650  0.427355  0.835343  5  2.0  5.0
2013-01-04  0.509859 -2.769586  1.000521  5  3.0  5.0
```

To get the boolean mask where values are `nan`.

```
In [60]: pd.isna(df1)
Out[60]: 
                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
```

3.2.5 Operations

See the *Basic section on Binary Ops*.

Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A   -0.286344
B   -0.489887
C    1.536861
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    1.698387
2013-01-02    1.711838
2013-01-03    1.387809
2013-01-04    1.348159
2013-01-05    1.992439
2013-01-06    2.113801
Freq: D, dtype: float64
```

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

```
In [63]: s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)

In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
Freq: D, dtype: float64

In [65]: df.sub(s, axis='index')
Out[65]:
                   A         B         C    D    F
2013-01-01       NaN       NaN       NaN  NaN  NaN
2013-01-02       NaN       NaN       NaN  NaN  NaN
2013-01-03 -2.323650 -0.572645 -0.164657  4.0  1.0
2013-01-04 -2.490141 -5.769586 -1.999479  2.0  0.0
2013-01-05 -4.860512 -5.259328 -3.917966  0.0 -1.0
2013-01-06       NaN       NaN       NaN  NaN  NaN
```

Apply

Applying functions to the data:

```
In [66]: df.apply(np.cumsum)
Out[66]:
                   A         B         C   D    F
2013-01-01  0.000000  0.000000  1.793547   5  NaN
2013-01-02 -0.913436  0.035141  5.231029  10  1.0
```

```
2013-01-03 -2.237086  0.462496  6.066372  15   3.0
2013-01-04 -1.727227 -2.307090  7.066893  20   6.0
2013-01-05 -1.587738 -2.566418  8.148927  25  10.0
2013-01-06 -1.718066 -2.939324  9.221163  30  15.0

In [67]: df.apply(lambda x: x.max() - x.min())
Out[67]: 
A    1.833510
B    3.196941
C    2.602139
D    0.000000
F    4.000000
dtype: float64
```

Histogramming

See more at *Histogramming and Discretization*.

```
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))

In [69]: s
Out[69]: 
0    3
1    5
2    1
3    4
4    3
5    3
6    5
7    3
8    1
9    1
dtype: int64

In [70]: s.value_counts()
Out[70]: 
3    4
1    3
5    2
4    1
dtype: int64
```

String Methods

Series is equipped with a set of string processing methods in the *str* attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in *str* generally uses regular expressions by default (and in some cases always uses them). See more at *Vectorized String Methods*.

```
In [71]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [72]: s.str.lower()
Out[72]: 
0       a
```

(continues on next page)

(continued from previous page)

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

3.2.6 Merge

Concat

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

See the *Merging section*.

Concatenating pandas objects together with `concat()`:

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]: 
          0         1         2         3
0 -0.060367  1.524865  0.131370 -1.258189
1  1.350664  0.281335 -1.137282  0.837561
2 -0.693000  0.319506 -0.629133  0.372674
3  0.044135  0.012920  0.369063  0.966930
4  2.401162 -0.910612  0.105657 -1.556792
5  0.633341 -1.540587 -1.379657  0.402392
6 -1.098101  2.017028 -0.309634  0.421307
7  0.098253 -1.233080  0.729444 -0.055611
8 -1.361441 -0.220021 -0.670728 -0.796977
9 -0.430638 -0.515560  1.171513 -0.172970

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]

In [76]: pd.concat(pieces)
Out[76]: 
          0         1         2         3
0 -0.060367  1.524865  0.131370 -1.258189
1  1.350664  0.281335 -1.137282  0.837561
2 -0.693000  0.319506 -0.629133  0.372674
3  0.044135  0.012920  0.369063  0.966930
4  2.401162 -0.910612  0.105657 -1.556792
5  0.633341 -1.540587 -1.379657  0.402392
6 -1.098101  2.017028 -0.309634  0.421307
7  0.098253 -1.233080  0.729444 -0.055611
8 -1.361441 -0.220021 -0.670728 -0.796977
9 -0.430638 -0.515560  1.171513 -0.172970
```

Join

SQL style merges. See the *Database style joining* section.

```
In [77]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [78]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [79]: left
Out[79]: 
   key  lval
0  foo     1
1  foo     2

In [80]: right
Out[80]: 
   key  rval
0  foo     4
1  foo     5

In [81]: pd.merge(left, right, on='key')
Out[81]: 
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Another example that can be given is:

```
In [82]: left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})

In [83]: right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

In [84]: left
Out[84]: 
   key  lval
0  foo     1
1  bar     2

In [85]: right
Out[85]: 
   key  rval
0  foo     4
1  bar     5

In [86]: pd.merge(left, right, on='key')
Out[86]: 
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

Append

Append rows to a dataframe. See the *Appending* section.

```
In [87]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])

In [88]: df
Out[88]:
          A         B         C         D
0 -0.588928 -0.915521 -1.117137  0.203822
1 -1.122360  0.304649  1.552535  0.002581
2  0.508278 -0.514235  1.658575 -0.641554
3  1.350919  1.331824 -0.207267  2.652396
4 -0.899836  2.550564 -2.093739  1.951529
5 -0.868126 -1.450904  0.513953 -1.340940
6  0.704139 -0.254053 -1.231656  0.412464
7  1.165165 -0.985036 -0.380646  0.322498

In [89]: s = df.iloc[3]

In [90]: df.append(s, ignore_index=True)
Out[90]:
          A         B         C         D
0 -0.588928 -0.915521 -1.117137  0.203822
1 -1.122360  0.304649  1.552535  0.002581
2  0.508278 -0.514235  1.658575 -0.641554
3  1.350919  1.331824 -0.207267  2.652396
4 -0.899836  2.550564 -2.093739  1.951529
5 -0.868126 -1.450904  0.513953 -1.340940
6  0.704139 -0.254053 -1.231656  0.412464
7  1.165165 -0.985036 -0.380646  0.322498
8  1.350919  1.331824 -0.207267  2.652396
```

3.2.7 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 [91]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ....:                          'foo', 'bar', 'foo', 'foo'],
   ....:                    'B': ['one', 'one', 'two', 'three',
   ....:                          'two', 'two', 'one', 'three'],
   ....:                    'C': np.random.randn(8),
   ....:                    'D': np.random.randn(8)})
   ....:

In [92]: df
Out[92]:
     A      B         C         D
0  foo    one -0.800850 -0.401946
1  bar    one  0.574070 -0.239254
2  foo    two  0.076080 -0.851860
3  bar  three  0.658058  1.362899
4  foo    two  0.014033 -0.724381
```

(continues on next page)

(continued from previous page)

```
5  bar    two -2.027835 -0.554717
6  foo    one -0.871462 -0.654966
7  foo  three  0.898298  0.280791
```

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

```
In [93]: df.groupby('A').sum()
Out[93]:
            C         D
A
bar -0.795706  0.568928
foo -0.683901 -2.352362
```

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum` function.

```
In [94]: df.groupby(['A', 'B']).sum()
Out[94]:
                  C         D
A   B
bar one    0.574070 -0.239254
    three  0.658058  1.362899
    two   -2.027835 -0.554717
foo one   -1.672312 -1.056912
    three  0.898298  0.280791
    two    0.090113 -1.576241
```

3.2.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

Stack

```
In [95]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
   ....:                      'foo', 'foo', 'qux', 'qux'],
   ....:                     ['one', 'two', 'one', 'two',
   ....:                      'one', 'two', 'one', 'two']]))
   ....:

In [96]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [97]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [98]: df2 = df[:4]

In [99]: df2
Out[99]:
                     A         B
first second
bar   one     0.572341 -0.753047
      two     2.561035 -2.172735
baz   one    -0.747480 -1.381895
      two    -1.271458  1.926008
```

The `stack()` method compresses a level in the DataFrames columns.

```
In [100]: stacked = df2.stack()

In [101]: stacked
Out[101]: 
first  second   
bar    one     A    0.572341
               B   -0.753047
       two     A    2.561035
               B   -2.172735
baz    one     A   -0.747480
               B   -1.381895
       two     A   -1.271458
               B    1.926008
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 [102]: stacked.unstack()
Out[102]: 
                     A         B
first second                    
bar   one     0.572341 -0.753047
      two     2.561035 -2.172735
baz   one    -0.747480 -1.381895
      two    -1.271458  1.926008

In [103]: stacked.unstack(1)
Out[103]: 
second        one       two
first                      
bar   A  0.572341  2.561035
      B -0.753047 -2.172735
baz   A -0.747480 -1.271458
      B -1.381895  1.926008

In [104]: stacked.unstack(0)
Out[104]: 
first          bar       baz
second                      
one    A  0.572341 -0.747480
       B -0.753047 -1.381895
two    A  2.561035 -1.271458
       B -2.172735  1.926008
```

Pivot tables

See the section on *Pivot Tables*.

```
In [105]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
   .....:                    'B': ['A', 'B', 'C'] * 4,
   .....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
   .....:                    'D': np.random.randn(12),
   .....:                    'E': np.random.randn(12)})
```

(continues on next page)

(continued from previous page)

```
   .....:

In [106]: df
Out[106]:
        A  B    C         D         E
0     one  A  foo -0.780671 -0.261755
1     one  B  foo -0.294512  0.557932
2     two  C  foo  0.214464 -0.014909
3   three  A  bar  0.176056  3.227353
4     one  B  bar -0.836311 -0.832515
5     one  C  bar  0.579569  0.072856
6     two  A  foo -0.746106 -0.620299
7   three  B  foo -0.214858  1.084295
8     one  C  foo  0.206084  0.244296
9     one  A  bar  0.505611 -0.589272
10    two  B  bar -0.181058 -1.117857
11  three  C  bar  0.527833 -1.061391
```

We can produce pivot tables from this data very easily:

```
In [107]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[107]:
C             bar       foo
A     B
one   A  0.505611 -0.780671
      B -0.836311 -0.294512
      C  0.579569  0.206084
three A  0.176056       NaN
      B       NaN -0.214858
      C  0.527833       NaN
two   A       NaN -0.746106
      B -0.181058       NaN
      C       NaN  0.214464
```

3.2.9 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 [108]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [109]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [110]: ts.resample('5Min').sum()
Out[110]:
2012-01-01    24418
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [111]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

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

(continues on next page)

(continued from previous page)

```
In [113]: ts
Out[113]:
2012-03-06    0.198218
2012-03-07   -0.649384
2012-03-08    0.743513
2012-03-09   -0.541289
2012-03-10   -0.594013
Freq: D, dtype: float64

In [114]: ts_utc = ts.tz_localize('UTC')

In [115]: ts_utc
Out[115]:
2012-03-06 00:00:00+00:00    0.198218
2012-03-07 00:00:00+00:00   -0.649384
2012-03-08 00:00:00+00:00    0.743513
2012-03-09 00:00:00+00:00   -0.541289
2012-03-10 00:00:00+00:00   -0.594013
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [116]: ts_utc.tz_convert('US/Eastern')
Out[116]:
2012-03-05 19:00:00-05:00    0.198218
2012-03-06 19:00:00-05:00   -0.649384
2012-03-07 19:00:00-05:00    0.743513
2012-03-08 19:00:00-05:00   -0.541289
2012-03-09 19:00:00-05:00   -0.594013
Freq: D, dtype: float64
```

Converting between time span representations:

```
In [117]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [118]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [119]: ts
Out[119]:
2012-01-31   -0.579681
2012-02-29    0.235443
2012-03-31    0.121223
2012-04-30    1.630841
2012-05-31   -0.795301
Freq: M, dtype: float64

In [120]: ps = ts.to_period()

In [121]: ps
Out[121]:
2012-01   -0.579681
2012-02    0.235443
2012-03    0.121223
2012-04    1.630841
2012-05   -0.795301
Freq: M, dtype: float64
```

```
In [122]: ps.to_timestamp()
Out[122]:
2012-01-01   -0.579681
2012-02-01    0.235443
2012-03-01    0.121223
2012-04-01    1.630841
2012-05-01   -0.795301
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 [123]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [124]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [125]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [126]: ts.head()
Out[126]:
1990-03-01 09:00    0.456987
1990-06-01 09:00   -1.222265
1990-09-01 09:00    0.773708
1990-12-01 09:00   -1.138070
1991-03-01 09:00   -0.860873
Freq: H, dtype: float64
```

3.2.10 Categoricals

pandas can include categorical data in a `DataFrame`. For full docs, see the *categorical introduction* and the *API documentation*.

```
In [127]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
   .....:                    "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
   .....:
```

Convert the raw grades to a categorical data type.

```
In [128]: df["grade"] = df["raw_grade"].astype("category")

In [129]: df["grade"]
Out[129]:
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

Rename the categories to more meaningful names (assigning to `Series.cat.categories` is inplace!).

```
In [130]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

Reorder the categories and simultaneously add the missing categories (methods under `Series .cat` return a new `Series` by default).

```
In [131]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                               "good", "very good"])
   .....:

In [132]: df["grade"]
Out[132]:
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

Sorting is per order in the categories, not lexical order.

```
In [133]: df.sort_values(by="grade")
Out[133]:
   id raw_grade      grade
5   6         e   very bad
1   2         b       good
2   3         b       good
0   1         a  very good
3   4         a  very good
4   5         a  very good
```

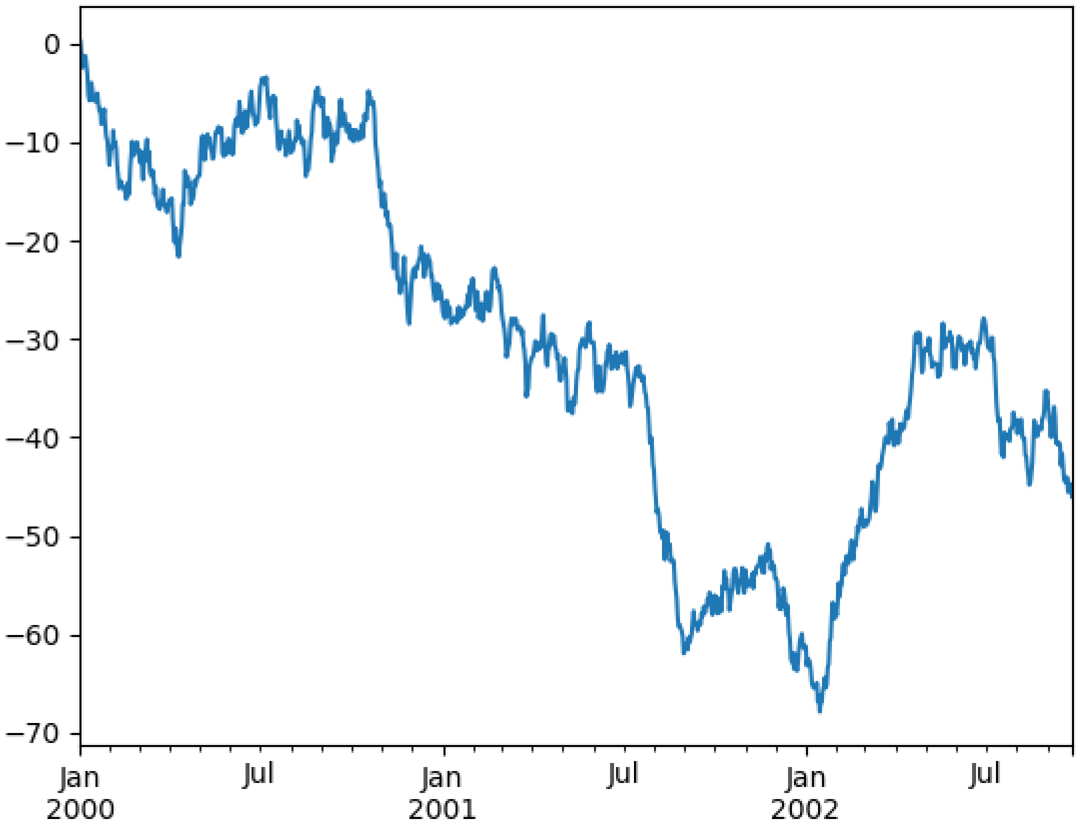
Grouping by a categorical column also shows empty categories.

```
In [134]: df.groupby("grade").size()
Out[134]:
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

3.2.11 Plotting

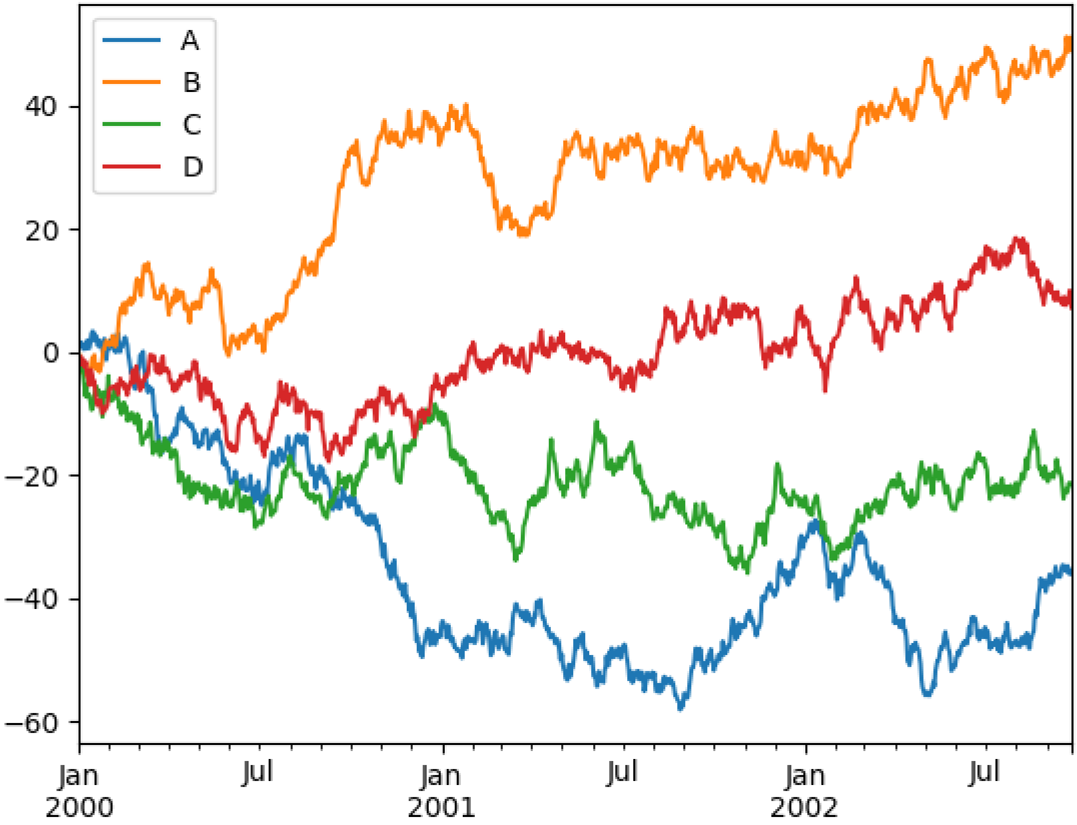
See the *Plotting* docs.

```
In [135]: ts = pd.Series(np.random.randn(1000),
   .....:                index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [136]: ts = ts.cumsum()

In [137]: ts.plot()
Out[137]: <matplotlib.axes._subplots.AxesSubplot at 0x124144910>
```



On a DataFrame, the `plot()` method is a convenience to plot all of the columns with labels:

```
In [138]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
   .....:                   columns=['A', 'B', 'C', 'D'])
   .....:

In [139]: df = df.cumsum()

In [140]: plt.figure()
Out[140]: <Figure size 640x480 with 0 Axes>

In [141]: df.plot()
Out[141]: <matplotlib.axes._subplots.AxesSubplot at 0x129c58f90>

In [142]: plt.legend(loc='best')
Out[142]: <matplotlib.legend.Legend at 0x129c5df10>
```



3.2.12 Getting data in/out

CSV

Writing to a csv file.

```
In [143]: df.to_csv('foo.csv')
```

Reading from a csv file.

```
In [144]: pd.read_csv('foo.csv')
Out[144]:
     Unnamed: 0          A          B          C          D
0    2000-01-01  -0.327225   2.629468   0.152017   0.095606
1    2000-01-02  -0.924297   0.950886   0.363196  -1.492713
2    2000-01-03  -0.987506   0.909911   0.357891  -1.526351
3    2000-01-04  -2.537033   1.483865   0.264791  -2.119897
4    2000-01-05  -2.152314   2.492344   1.618494  -2.466438
..          ...        ...        ...        ...        ...
995  2002-09-22  11.523422 -30.110846  69.683435  68.775420
996  2002-09-23  11.002664 -29.664630  69.359104  67.976973
997  2002-09-24  11.327230 -28.225995  69.602334  66.540402
```

(continues on next page)

(continued from previous page)

```
998  2002-09-25  12.654992 -28.103791  70.420846  66.271805
999  2002-09-26  13.310373 -27.917046  71.319203  67.595342

[1000 rows x 5 columns]
```

HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store.

```
In [145]: df.to_hdf('foo.h5', 'df')
```

Reading from a HDF5 Store.

```
In [146]: pd.read_hdf('foo.h5', 'df')
Out[146]: 
                    A          B          C          D
2000-01-01  -0.327225   2.629468   0.152017   0.095606
2000-01-02  -0.924297   0.950886   0.363196  -1.492713
2000-01-03  -0.987506   0.909911   0.357891  -1.526351
2000-01-04  -2.537033   1.483865   0.264791  -2.119897
2000-01-05  -2.152314   2.492344   1.618494  -2.466438
...               ...        ...        ...        ...
2002-09-22  11.523422 -30.110846  69.683435  68.775420
2002-09-23  11.002664 -29.664630  69.359104  67.976973
2002-09-24  11.327230 -28.225995  69.602334  66.540402
2002-09-25  12.654992 -28.103791  70.420846  66.271805
2002-09-26  13.310373 -27.917046  71.319203  67.595342

[1000 rows x 4 columns]
```

Excel

Reading and writing to *MS Excel*.

Writing to an excel file.

```
In [147]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

Reading from an excel file.

```
In [148]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
Out[148]: 
    Unnamed: 0          A          B          C          D
0   2000-01-01  -0.327225   2.629468   0.152017   0.095606
1   2000-01-02  -0.924297   0.950886   0.363196  -1.492713
2   2000-01-03  -0.987506   0.909911   0.357891  -1.526351
3   2000-01-04  -2.537033   1.483865   0.264791  -2.119897
4   2000-01-05  -2.152314   2.492344   1.618494  -2.466438
..         ...        ...        ...        ...        ...
995 2002-09-22  11.523422 -30.110846  69.683435  68.775420
996 2002-09-23  11.002664 -29.664630  69.359104  67.976973
997 2002-09-24  11.327230 -28.225995  69.602334  66.540402
```

(continues on next page)

(continued from previous page)

```
998 2002-09-25  12.654992 -28.103791  70.420846  66.271805
999 2002-09-26  13.310373 -27.917046  71.319203  67.595342

[1000 rows x 5 columns]
```

3.2.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

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

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

See *Gotchas* as well. {{ header }}

3.3 Essential basic functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. Heres how to create some of the objects used in the examples from the previous section:

```
In [1]: index = pd.date_range('1/1/2000', periods=8)

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

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

3.3.1 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 [4]: long_series = pd.Series(np.random.randn(1000))

In [5]: long_series.head()
Out[5]:
0   -0.183284
1   -1.084859
2    0.804220
3   -2.278089
4    0.736586
dtype: float64

In [6]: long_series.tail(3)
Out[6]:
997   -0.935665
```

```
998    0.865342
999   -0.798301
dtype: float64
```

3.3.2 Attributes and underlying data

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*

Note, **these attributes can be safely assigned to**!

```
In [7]: df[:2]
Out[7]:
                   A         B         C
2000-01-01 -0.216974  0.840135 -0.733559
2000-01-02  0.054693 -2.419175 -0.196602

In [8]: df.columns = [x.lower() for x in df.columns]

In [9]: df
Out[9]:
                   a         b         c
2000-01-01 -0.216974  0.840135 -0.733559
2000-01-02  0.054693 -2.419175 -0.196602
2000-01-03  0.088430  0.295848 -0.321244
2000-01-04 -0.405187  0.377840 -0.991461
2000-01-05 -0.204197 -0.949996 -0.892288
2000-01-06 -1.144473  0.336703 -1.526753
2000-01-07 -1.591884 -1.341427 -1.075799
2000-01-08  0.101217  0.253586  1.122798
```

Pandas objects (`Index`, `Series`, `DataFrame`) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may *extend* NumPys type system to add support for custom arrays (see *dtypes*).

To get the actual data inside a `Index` or `Series`, use the `.array` property

```
In [10]: s.array
Out[10]:
<PandasArray>
[-0.0065828035649045415,    -2.7167036660839705,     1.2287390078978135,
     0.9324879342551502,     0.2940471792704379]
Length: 5, dtype: float64

In [11]: s.index.array
Out[11]:
<PandasArray>
['a', 'b', 'c', 'd', 'e']
Length: 5, dtype: object
```

`array` will always be an *`ExtensionArray`*. The exact details of what an *`ExtensionArray`* is and why pandas uses them is a bit beyond the scope of this introduction. See *dtypes* for more.

If you know you need a NumPy array, use `to_numpy()` or `numpy.asarray()`.

```
In [12]: s.to_numpy()
Out[12]: array([-0.0065828 , -2.71670367,  1.22873901,  0.93248793,  0.
↪29404718])

In [13]: np.asarray(s)
Out[13]: array([-0.0065828 , -2.71670367,  1.22873901,  0.93248793,  0.
↪29404718])
```

When the Series or Index is backed by an *ExtensionArray*, `to_numpy()` may involve copying data and coercing values. See *dtypes* for more.

`to_numpy()` gives some control over the `dtype` of the resulting `numpy.ndarray`. For example, consider datetimes with timezones. NumPy doesnt have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype `numpy.ndarray` with `Timestamp` objects, each with the correct `tz`
2. A `datetime64[ns]` -dtype `numpy.ndarray`, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with `dtype=object`

```
In [14]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))

In [15]: ser.to_numpy(dtype=object)
Out[15]: 
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or thrown away with `dtype='datetime64[ns]'`

```
In [16]: ser.to_numpy(dtype="datetime64[ns]")
Out[16]: 
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
      dtype='datetime64[ns]')
```

Getting the raw data inside a `DataFrame` is possibly a bit more complex. When your `DataFrame` only has a single data type for all the columns, `DataFrame.to_numpy()` will return the underlying data:

```
In [17]: df.to_numpy()
Out[17]: 
array([[-0.21697434,  0.84013463, -0.73355917],
       [ 0.05469281, -2.41917478, -0.19660224],
       [ 0.08843011,  0.29584755, -0.32124436],
       [-0.40518687,  0.37784039, -0.99146082],
       [-0.20419686, -0.94999608, -0.89228838],
       [-1.14447294,  0.33670262, -1.52675331],
       [-1.59188382, -1.34142681, -1.07579903],
       [ 0.1012167 ,  0.2535859 ,  1.12279785]])
```

If a DataFrame 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 DataFrames 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.

In the past, pandas recommended `Series.values` or `DataFrame.values` for extracting the data from a Series or DataFrame. Youll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an *extension type*, its unclear whether `Series.values` returns a NumPy array or the extension array. `Series.array` will always return an *`ExtensionArray`*, and will never copy data. `Series.to_numpy()` will always return a NumPy array, potentially at the cost of copying / coercing values.
2. When your DataFrame contains a mixture of data types, `DataFrame.values` may involve copying data and coercing values to a common dtype, a relatively expensive operation. `DataFrame.to_numpy()`, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

3.3.3 Accelerated operations

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

These are both enabled to be used by default, you can control this by setting the options:

New in version 0.20.0.

```
pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)
```

3.3.4 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.

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 [18]: df = pd.DataFrame({
   ....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   ....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   ....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   ....:

In [19]: df
Out[19]:
        one       two     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [20]: row = df.iloc[1]

In [21]: column = df['two']

In [22]: df.sub(row, axis='columns')
Out[22]:
        one       two     three
a -0.889039 -0.441476       NaN
b  0.000000  0.000000  0.000000
c -0.159719 -0.767122 -2.296662
d       NaN -0.095880 -0.210291

In [23]: df.sub(row, axis=1)
Out[23]:
        one       two     three
a -0.889039 -0.441476       NaN
b  0.000000  0.000000  0.000000
c -0.159719 -0.767122 -2.296662
d       NaN -0.095880 -0.210291

In [24]: df.sub(column, axis='index')
Out[24]:
        one  two     three
a  0.036045  0.0       NaN
b  0.483609  0.0  0.377604
c  1.091012  0.0 -1.151936
d       NaN  0.0  0.263193

In [25]: df.sub(column, axis=0)
Out[25]:
        one  two     three
a  0.036045  0.0       NaN
b  0.483609  0.0  0.377604
c  1.091012  0.0 -1.151936
d       NaN  0.0  0.263193
```

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

```
In [26]: dfmi = df.copy()
```

(continues on next page)

(continued from previous page)

```
In [27]: dfmi.index = pd.MultiIndex.from_tuples([(1, 'a'), (1, 'b'),
   ....:                                          (1, 'c'), (2, 'a')],
   ....:                                         names=['first', 'second'])
   ....: 

In [28]: dfmi.sub(column, axis=0, level='second')
Out[28]: 
                   one       two     three
first second                              
1     a       0.036045  0.000000       NaN
      b       0.483609  0.000000  0.377604
      c       1.091012  0.000000 -1.151936
2     a            NaN  0.345596  0.608789
```

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [29]: s = pd.Series(np.arange(10))

In [30]: s
Out[30]: 
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64

In [31]: div, rem = divmod(s, 3)

In [32]: div
Out[32]: 
0    0
1    0
2    0
3    1
4    1
5    1
6    2
7    2
8    2
9    3
dtype: int64

In [33]: rem
Out[33]: 
0    0
1    1
2    2
```

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

In [34]: idx = pd.Index(np.arange(10))

In [35]: idx
Out[35]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [36]: div, rem = divmod(idx, 3)

In [37]: div
Out[37]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [38]: rem
Out[38]: Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

We can also do elementwise `divmod()`:

```
In [39]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

In [40]: div
Out[40]: 
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64

In [41]: rem
Out[41]: 
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int64
```

Missing data / operations with fill values

In Series and DataFrame, 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 [42]: df
Out[42]:
        one       two     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [43]: df2
Out[43]:
        one       two     three
a  0.280390  0.244345  1.000000
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [44]: df + df2
Out[44]:
        one       two     three
a  0.560780  0.488689       NaN
b  2.338859  1.371642  2.126849
c  2.019421 -0.162603 -2.466474
d       NaN  1.179882  1.706268

In [45]: df.add(df2, fill_value=0)
Out[45]:
        one       two     three
a  0.560780  0.488689  1.000000
b  2.338859  1.371642  2.126849
c  2.019421 -0.162603 -2.466474
d       NaN  1.179882  1.706268
```

Flexible comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

```
In [46]: df.gt(df2)
Out[46]:
     one    two  three
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [47]: df2.ne(df)
Out[47]:
     one    two  three
a  False  False   True
b  False  False  False
c  False  False  False
d   True  False  False
```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These `boolean` objects can be used in indexing operations, see the section on *Boolean indexing*.

Boolean reductions

You can apply the reductions: `empty`, `any()`, `all()`, and `bool()` to provide a way to summarize a boolean result.

```
In [48]: (df > 0).all()
Out[48]:
one      False
two      False
three    False
dtype: bool

In [49]: (df > 0).any()
Out[49]:
one      True
two      True
three    True
dtype: bool
```

You can reduce to a final boolean value.

```
In [50]: (df > 0).any().any()
Out[50]: True
```

You can test if a pandas object is empty, via the `empty` property.

```
In [51]: df.empty
Out[51]: False

In [52]: pd.DataFrame(columns=list('ABC')).empty
Out[52]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```
In [53]: pd.Series([True]).bool()
Out[53]: True

In [54]: pd.Series([False]).bool()
Out[54]: False

In [55]: pd.DataFrame([[True]]).bool()
Out[55]: True

In [56]: pd.DataFrame([[False]]).bool()
Out[56]: False
```

Warning: You might be tempted to do the following:

```
>>> if df:
...     pass
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.:

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.
↪all().
```

See *gotchas* for a more detailed discussion.

Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df + df` and `df * 2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df + df == df * 2).all()`. But in fact, this expression is False:

```
In [57]: df + df == df * 2
Out[57]:
     one   two  three
a   True  True  False
b   True  True   True
c   True  True   True
d  False  True   True

In [58]: (df + df == df * 2).all()
Out[58]:
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df + df == df * 2` contains some False values! This is because NaNs do not compare as equals:

```
In [59]: np.nan == np.nan
Out[59]: False
```

So, NDFrames (such as Series and DataFrames) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [60]: (df + df).equals(df * 2)
Out[60]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [61]: df1 = pd.DataFrame({'col': ['foo', 0, np.nan]})

In [62]: df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2, 1, 0])

In [63]: df1.equals(df2)
```

```
Out[63]: False

In [64]: df1.equals(df2.sort_index())
Out[64]: True
```

Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [65]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
Out[65]:
0     True
1    False
2    False
dtype: bool

In [66]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
Out[66]: array([ True, False, False])
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [67]: pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
Out[67]:
0     True
1     True
2    False
dtype: bool

In [68]: pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
Out[68]:
0     True
1     True
2    False
dtype: bool
```

Trying to compare `Index` or `Series` objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([2])
Out[69]: array([False,  True, False])
```

or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

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

In [72]: df2 = pd.DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
   ....:                     'B': [np.nan, np.nan, 3., 4., 6., 8.]})
   ....:

In [73]: df1
Out[73]:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  3.0
3  5.0  NaN
4  NaN  6.0

In [74]: df2
Out[74]:
     A    B
0  5.0  NaN
1  2.0  NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [75]: df1.combine_first(df2)
Out[75]:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

General DataFrame combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```
In [76]: def combiner(x, y):
   ....:     return np.where(pd.isna(x), y, x)
   ....:
```

3.3.5 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*. 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)

For example:

```
In [77]: df
Out[77]:
        one       two     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [78]: df.mean(0)
Out[78]:
one      0.819843
two      0.359701
three    0.227774
dtype: float64

In [79]: df.mean(1)
Out[79]:
a    0.262367
b    0.972892
c   -0.101609
d    0.721537
dtype: float64
```

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

```
In [80]: df.sum(0, skipna=False)
Out[80]:
one           NaN
two      1.438805
three         NaN
dtype: float64

In [81]: df.sum(axis=1, skipna=True)
Out[81]:
a    0.524735
b    2.918675
c   -0.304828
```

```
d    1.443075
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 [82]: ts_stand = (df - df.mean()) / df.std()

In [83]: ts_stand.std()
Out[83]:
one      1.0
two      1.0
three    1.0
dtype: float64

In [84]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [85]: xs_stand.std(1)
Out[85]:
a    1.0
b    1.0
c    1.0
d    1.0
dtype: float64
```

Note that methods like `cumsum()` and `cumprod()` preserve the location of `NaN` values. This is somewhat different from `expanding()` and `rolling()`. For more details please see *this note*.

```
In [86]: df.cumsum()
Out[86]:
       one       two     three
a  0.28039  0.244345       NaN
b  1.44982  0.930166  1.063425
c  2.45953  0.848864 -0.169812
d      NaN  1.438805  0.683321
```

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-NA observations
`sum`	Sum of values
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Bessel-corrected sample standard deviation
`var`	Unbiased variance
`sem`	Standard error of the mean
`skew`	Sample skewness (3rd moment)
`kurt`	Sample 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 [87]: np.mean(df['one'])
Out[87]: 0.8198434389015307

In [88]: np.mean(df['one'].to_numpy())
Out[88]: nan
```

`Series.nunique()` will return the number of unique non-NA values in a Series:

```
In [89]: series = pd.Series(np.random.randn(500))

In [90]: series[20:500] = np.nan

In [91]: series[10:20] = 5

In [92]: series.nunique()
Out[92]: 11
```

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 [93]: series = pd.Series(np.random.randn(1000))

In [94]: series[::2] = np.nan

In [95]: series.describe()
Out[95]: 
count    500.000000
```

(continues on next page)

(continued from previous page)

```
mean       0.002283
std        0.975484
min       -2.857251
25%       -0.633802
50%       -0.017143
75%        0.693621
max        3.144426
dtype: float64

In [96]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ....:                      columns=['a', 'b', 'c', 'd', 'e'])
   ....:

In [97]: frame.iloc[::2] = np.nan

In [98]: frame.describe()
Out[98]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean    -0.015788   -0.044768    0.002118    0.071407   -0.006535
std      1.032578    0.988722    1.024366    0.992153    1.024544
min     -3.301982   -3.304828   -4.185081   -3.080779   -3.260973
25%     -0.620001   -0.756503   -0.691177   -0.564064   -0.709754
50%     -0.004659   -0.076543   -0.021568    0.018575    0.057651
75%      0.630194    0.673233    0.693157    0.744467    0.762700
max      3.210008    2.340774    3.014330    3.111480    2.846670
```

You can select specific percentiles to include in the output:

```
In [99]: series.describe(percentiles=[.05, .25, .75, .95])
Out[99]:
count    500.000000
mean       0.002283
std        0.975484
min       -2.857251
5%        -1.595479
25%       -0.633802
50%       -0.017143
75%        0.693621
95%        1.571126
max        3.144426
dtype: float64
```

By default, the median is always included.

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

```
In [100]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [101]: s.describe()
Out[101]:
count     9
unique    4
top       a
freq      5
dtype: object
```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [102]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [103]: frame.describe()
Out[103]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

This behavior can be controlled by providing a list of types as `include`/`exclude` arguments. The special value `all` can also be used:

```
In [104]: frame.describe(include=['object'])
Out[104]:
         a
count    4
unique   2
top     No
freq     2

In [105]: frame.describe(include=['number'])
Out[105]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000

In [106]: frame.describe(include='all')
Out[106]:
          a         b
count     4  4.000000
unique    2       NaN
top      No       NaN
freq      2       NaN
mean    NaN  1.500000
std     NaN  1.290994
min     NaN  0.000000
25%     NaN  0.750000
50%     NaN  1.500000
75%     NaN  2.250000
max     NaN  3.000000
```

That feature relies on *select_dtypes*. Refer to there for details about accepted inputs.

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 [107]: s1 = pd.Series(np.random.randn(5))

In [108]: s1
Out[108]: 
0   -0.203845
1    0.699335
2    1.576849
3    0.013191
4    0.222092
dtype: float64

In [109]: s1.idxmin(), s1.idxmax()
Out[109]: (0, 2)

In [110]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

In [111]: df1
Out[111]: 
          A         B         C
0 -0.432209 -0.068295  0.381522
1  0.414550 -0.113251  0.718740
2  1.841494  0.060288 -1.210947
3  0.119289 -0.496862  1.078065
4 -0.288879  0.362025 -3.061166

In [112]: df1.idxmin(axis=0)
Out[112]: 
A    0
B    3
C    4
dtype: int64

In [113]: df1.idxmax(axis=1)
Out[113]: 
0    C
1    C
2    A
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 [114]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'],␣
↪index=list('edcba'))

In [115]: df3
Out[115]: 
     A
```

```
e   2.0
d   1.0
c   1.0
b   3.0
a   NaN

In [116]: df3['A'].idxmin()
Out[116]: 'd'
```

Note: `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

Value counts (histogramming) / mode

The `value_counts()` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [117]: data = np.random.randint(0, 7, size=50)

In [118]: data
Out[118]: 
array([0, 0, 2, 3, 2, 5, 4, 0, 3, 4, 6, 4, 1, 5, 6, 5, 3, 1, 3, 6, 2, 0,
       0, 0, 3, 5, 5, 0, 5, 1, 3, 3, 2, 5, 6, 1, 1, 1, 0, 1, 0, 1, 5, 6,
       5, 3, 0, 0, 5, 2])

In [119]: s = pd.Series(data)

In [120]: s.value_counts()
Out[120]: 
0    11
5    10
3     8
1     8
6     5
2     5
4     3
dtype: int64

In [121]: pd.value_counts(data)
Out[121]: 
0    11
5    10
3     8
1     8
6     5
2     5
4     3
dtype: int64
```

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```
In [122]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])
```

(continues on next page)

(continued from previous page)

```
In [123]: s5.mode()
Out[123]:
0    3
1    7
dtype: int64

In [124]: df5 = pd.DataFrame({"A": np.random.randint(0, 7, size=50),
   .....:                     "B": np.random.randint(-10, 15, size=50)})
   .....:

In [125]: df5.mode()
Out[125]:
   A  B
0  3 -5
1  6  9
```

Discretization and quantiling

Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

```
In [126]: arr = np.random.randn(20)

In [127]: factor = pd.cut(arr, 4)

In [128]: factor
Out[128]:
[(0.883, 1.671], (-1.483, -0.692], (-0.692, 0.0955], (0.0955, 0.883], (0.883, 1.671],␣
↪..., (0.0955, 0.883], (-1.483, -0.692], (0.883, 1.671], (-1.483, -0.692], (0.0955,␣
↪0.883]]
Length: 20
Categories (4, interval[float64]): [(-1.483, -0.692] < (-0.692, 0.0955] < (0.0955, 0.
↪883] <
                                    (0.883, 1.671]]

In [129]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [130]: factor
Out[130]:
[(1, 5], (-5, -1], (-1, 0], (0, 1], (1, 5], ..., (0, 1], (-1, 0], (1, 5], (-1, 0], (0,
↪ 1]]
Length: 20
Categories (4, interval[int64]): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]
```

`qcut()` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [131]: arr = np.random.randn(30)

In [132]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [133]: factor
Out[133]:
[(-0.32, 0.378], (0.378, 1.888], (-1.054, -0.32], (-2.8859999999999997, -1.
↪054], (-0.32, 0.378], ..., (-2.8859999999999997, -1.054], (-1.054, -0.32],␣
↪(-1.054, -0.32], (-0.32, 0.378], (-2.8859999999999997, -1.054]]
Length: 30
Categories (4, interval[float64]): [(-2.8859999999999997, -1.054] < (-1.054, -
↪0.32] < (-0.32, 0.378] <
                                    (0.378, 1.888]]

In [134]: pd.value_counts(factor)
Out[134]:
(0.378, 1.888]                   8
(-2.8859999999999997, -1.054]    8
(-0.32, 0.378]                   7
(-1.054, -0.32]                  7
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [135]: arr = np.random.randn(20)

In [136]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [137]: factor
Out[137]:
[(-inf, 0.0], (-inf, 0.0], (0.0, inf], (-inf, 0.0], (0.0, inf], ..., (-inf, 0.0], (0.
↪0, inf], (0.0, inf], (-inf, 0.0], (0.0, inf]]
Length: 20
Categories (2, interval[float64]): [(-inf, 0.0] < (0.0, inf]]
```

3.3.6 Function application

To apply your own or another librarys functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. *Tablewise Function Application*: `pipe()`
2. *Row or Column-wise Function Application*: `apply()`
3. *Aggregation API*: `agg()` and `transform()`
4. *Applying Elementwise Functions*: `applymap()`

Tablewise function application

`DataFrames` and `Series` can of course just be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method. Compare the following

```
# f, g, and h are functions taking and returning ``DataFrames``
>>> f(g(h(df), arg1=1), arg2=2, arg3=3)
```

with the equivalent

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

Pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another librarys functions in method chains, alongside pandas methods.

In the example above, the functions `f`, `g`, and `h` each expected the `DataFrame` as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide `pipe` with a tuple of `(callable, data_keyword)`. `.pipe` will route the `DataFrame` to the argument specified in the tuple.

For example, we can fit a regression using statsmodels. Their API expects a formula first and a `DataFrame` as the second argument, `data`. We pass in the function, keyword pair `(sm.ols, 'data')` to `pipe`:

```
In [138]: import statsmodels.formula.api as sm

In [139]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [140]: (bb.query('h > 0')
   .....:    .assign(ln_h=lambda df: np.log(df.h))
   .....:    .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
   .....:    .fit()
   .....:    .summary()
   .....:  )
   .....:
Out[140]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                     hr   R-squared:                       0.685
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Sat, 02 Nov 2019   Prob (F-statistic):           3.48e-15
Time:                        16:04:51   Log-Likelihood:                -205.92
No. Observations:                  68   AIC:                             421.8
Df Residuals:                      63   BIC:                             432.9
Df Model:                           4
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept   -8484.7720   4664.146     -1.819      0.074   -1.78e+04     835.780
C(lg)[T.NL]    -2.2736      1.325     -1.716      0.091      -4.922       0.375
ln_h           -1.3542      0.875     -1.547      0.127      -3.103       0.395
year            4.2277      2.324      1.819      0.074      -0.417       8.872
g               0.1841      0.029      6.258      0.000       0.125       0.243
==============================================================================
Omnibus:                       10.875   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               17.298
Skew:                           0.537   Prob(JB):                     0.000175
Kurtosis:                       5.225   Cond. No.                     1.49e+07
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular `(%>%)` (read pipe) operator for R. The implementation of `pipe` here is quite clean and feels right at home in python.

We encourage you to view the source code of `pipe()`.

Row or column-wise function application

Arbitrary functions can be applied along the axes of a DataFrame using the `apply()` method, which, like the descriptive statistics methods, takes an optional `axis` argument:

```
In [141]: df.apply(np.mean)
Out[141]:
one      0.819843
two      0.359701
three    0.227774
dtype: float64

In [142]: df.apply(np.mean, axis=1)
Out[142]:
a    0.262367
b    0.972892
c   -0.101609
d    0.721537
dtype: float64

In [143]: df.apply(lambda x: x.max() - x.min())
Out[143]:
one      0.889039
two      0.767122
three    2.296662
dtype: float64

In [144]: df.apply(np.cumsum)
Out[144]:
       one       two     three
a  0.28039  0.244345       NaN
b  1.44982  0.930166  1.063425
c  2.45953  0.848864 -0.169812
d      NaN  1.438805  0.683321

In [145]: df.apply(np.exp)
Out[145]:
        one       two     three
a  1.323646  1.276784       NaN
b  3.220155  1.985401  2.896273
c  2.744807  0.921916  0.291348
d       NaN  1.803882  2.346990
```

The `apply()` method will also dispatch on a string method name.

```
In [146]: df.apply('mean')
Out[146]:
one      0.819843
two      0.359701
three    0.227774
dtype: float64

In [147]: df.apply('mean', axis=1)
```

```
Out[147]:
a    0.262367
b    0.972892
c   -0.101609
d    0.721537
dtype: float64
```

The return type of the function passed to `apply()` affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-likes return values expand (or not) to a `DataFrame`.

`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 [148]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [149]: tsdf.apply(lambda x: x.idxmax())
Out[149]:
A   2000-02-28
B   2001-02-03
C   2002-04-25
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 [150]: tsdf
Out[150]:
                   A         B         C
2000-01-01 -0.482395  0.259874 -0.377157
2000-01-02  1.775855  0.614308  0.014913
2000-01-03 -1.122091  0.798609  0.259841
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 -0.543235 -0.939397 -0.534679
2000-01-09 -0.857516 -0.277678  0.631568
2000-01-10 -0.169672 -0.135461 -1.163331

In [151]: tsdf.apply(pd.Series.interpolate)
```

```
Out[151]:
                   A         B         C
2000-01-01 -0.482395  0.259874 -0.377157
2000-01-02  1.775855  0.614308  0.014913
2000-01-03 -1.122091  0.798609  0.259841
2000-01-04 -1.006319  0.451007  0.100937
2000-01-05 -0.890548  0.103406 -0.057967
2000-01-06 -0.774777 -0.244195 -0.216871
2000-01-07 -0.659006 -0.591796 -0.375775
2000-01-08 -0.543235 -0.939397 -0.534679
2000-01-09 -0.857516 -0.277678  0.631568
2000-01-10 -0.169672 -0.135461 -1.163331
```

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.

Aggregation API

New in version 0.20.0.

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see *groupby API*, the *window functions API*, and the *resample API*. The entry point for aggregation is `DataFrame.aggregate()`, or the alias `DataFrame.agg()`.

We will use a similar starting frame from above:

```
In [152]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [153]: tsdf.iloc[3:7] = np.nan

In [154]: tsdf
Out[154]:
                   A         B         C
2000-01-01 -1.505167  0.208159  0.666916
2000-01-02  1.198501  0.575635  0.519301
2000-01-03  0.787026 -1.428255 -0.778372
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 -0.529666  0.296110  0.843091
2000-01-09 -2.084206  0.578940  0.672280
2000-01-10  1.161593 -1.926451 -0.851579
```

Using a single function is equivalent to `apply()`. You can also pass named methods as strings. These will return a `Series` of the aggregated output:

```
In [155]: tsdf.agg(np.sum)
Out[155]:
A   -0.971919
B   -1.695863
C    1.071637
dtype: float64
```

```
In [156]: tsdf.agg('sum')
Out[156]:
A   -0.971919
B   -1.695863
C    1.071637
dtype: float64

# these are equivalent to a ``.sum()`` because we are aggregating
# on a single function
In [157]: tsdf.sum()
Out[157]:
A   -0.971919
B   -1.695863
C    1.071637
dtype: float64
```

Single aggregations on a `Series` this will return a scalar value:

```
In [158]: tsdf.A.agg('sum')
Out[158]: -0.9719194560759219
```

Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting `DataFrame`. These are naturally named from the aggregation function.

```
In [159]: tsdf.agg(['sum'])
Out[159]:
            A         B         C
sum -0.971919 -1.695863  1.071637
```

Multiple functions yield multiple rows:

```
In [160]: tsdf.agg(['sum', 'mean'])
Out[160]:
             A         B         C
sum  -0.971919 -1.695863  1.071637
mean -0.161987 -0.282644  0.178606
```

On a `Series`, multiple functions return a `Series`, indexed by the function names:

```
In [161]: tsdf.A.agg(['sum', 'mean'])
Out[161]:
sum    -0.971919
mean   -0.161987
Name: A, dtype: float64
```

Passing a `lambda` function will yield a `<lambda>` named row:

```
In [162]: tsdf.A.agg(['sum', lambda x: x.mean()])
Out[162]:
sum        -0.971919
<lambda>   -0.161987
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [163]: def mymean(x):
   .....:     return x.mean()
   .....:

In [164]: tsdf.A.agg(['sum', mymean])
Out[164]:
sum      -0.971919
mymean   -0.161987
Name: A, dtype: float64
```

Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to `DataFrame.agg` allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an `OrderedDict` instead to guarantee ordering.

```
In [165]: tsdf.agg({'A': 'mean', 'B': 'sum'})
Out[165]:
A   -0.161987
B   -1.695863
dtype: float64
```

Passing a list-like will generate a `DataFrame` output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be `NaN`:

```
In [166]: tsdf.agg({'A': ['mean', 'min'], 'B': 'sum'})
Out[166]:
             A         B
mean -0.161987       NaN
min  -2.084206       NaN
sum        NaN -1.695863
```

Mixed dtypes

When presented with mixed dtypes that cannot aggregate, `.agg` will only take the valid aggregations. This is similar to how groupby `.agg` works.

```
In [167]: mdf = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [1., 2., 3.],
   .....:                     'C': ['foo', 'bar', 'baz'],
   .....:                     'D': pd.date_range('20130101', periods=3)})
   .....:

In [168]: mdf.dtypes
Out[168]:
A             int64
B           float64
C            object
D    datetime64[ns]
dtype: object
```

```
In [169]: mdf.agg(['min', 'sum'])
Out[169]:
```

(continues on next page)

(continued from previous page)

```
     A    B          C          D
min  1  1.0        bar 2013-01-01
sum  6  6.0  foobarbaz        NaT
```

Custom describe

With `.agg()` is it possible to easily create a custom describe function, similar to the built in *describe function*.

```
In [170]: from functools import partial

In [171]: q_25 = partial(pd.Series.quantile, q=0.25)

In [172]: q_25.__name__ = '25%'

In [173]: q_75 = partial(pd.Series.quantile, q=0.75)

In [174]: q_75.__name__ = '75%'

In [175]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
Out[175]:
               A         B         C
count   6.000000  6.000000  6.000000
mean   -0.161987 -0.282644  0.178606
std     1.423915  1.101757  0.776770
min    -2.084206 -1.926451 -0.851579
25%    -1.261292 -1.019151 -0.453954
median  0.128680  0.252134  0.593108
75%     1.067951  0.505754  0.670939
max     1.198501  0.578940  0.843091
```

Transform API

New in version 0.20.0.

The `transform()` method returns an object that is indexed the same (same size) as the original. This API allows you to provide *multiple* operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

```
In [176]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [177]: tsdf.iloc[3:7] = np.nan

In [178]: tsdf
Out[178]:
                   A         B         C
2000-01-01 -0.279295  0.417231  0.537995
2000-01-02  0.807265  1.387190 -0.982545
2000-01-03 -0.538076  0.192555 -1.012581
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-07       NaN       NaN       NaN
2000-01-08 -1.981527  0.274648 -0.747219
2000-01-09  1.473561  0.232948  0.192742
2000-01-10  0.263927 -0.153731  0.831473
```

Transform the entire frame. `.transform()` allows input functions as: a NumPy function, a string function name or a user defined function.

```
In [179]: tsdf.transform(np.abs)
Out[179]:
                   A         B         C
2000-01-01  0.279295  0.417231  0.537995
2000-01-02  0.807265  1.387190  0.982545
2000-01-03  0.538076  0.192555  1.012581
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  1.981527  0.274648  0.747219
2000-01-09  1.473561  0.232948  0.192742
2000-01-10  0.263927  0.153731  0.831473

In [180]: tsdf.transform('abs')
Out[180]:
                   A         B         C
2000-01-01  0.279295  0.417231  0.537995
2000-01-02  0.807265  1.387190  0.982545
2000-01-03  0.538076  0.192555  1.012581
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  1.981527  0.274648  0.747219
2000-01-09  1.473561  0.232948  0.192742
2000-01-10  0.263927  0.153731  0.831473

In [181]: tsdf.transform(lambda x: x.abs())
Out[181]:
                   A         B         C
2000-01-01  0.279295  0.417231  0.537995
2000-01-02  0.807265  1.387190  0.982545
2000-01-03  0.538076  0.192555  1.012581
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  1.981527  0.274648  0.747219
2000-01-09  1.473561  0.232948  0.192742
2000-01-10  0.263927  0.153731  0.831473
```

Here `transform()` received a single function; this is equivalent to a ufunc application.

```
In [182]: np.abs(tsdf)
Out[182]:
```

(continues on next page)

(continued from previous page)

```
                   A         B         C
2000-01-01  0.279295  0.417231  0.537995
2000-01-02  0.807265  1.387190  0.982545
2000-01-03  0.538076  0.192555  1.012581
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  1.981527  0.274648  0.747219
2000-01-09  1.473561  0.232948  0.192742
2000-01-10  0.263927  0.153731  0.831473
```

Passing a single function to `.transform()` with a `Series` will yield a single `Series` in return.

```
In [183]: tsdf.A.transform(np.abs)
Out[183]: 
2000-01-01    0.279295
2000-01-02    0.807265
2000-01-03    0.538076
2000-01-04         NaN
2000-01-05         NaN
2000-01-06         NaN
2000-01-07         NaN
2000-01-08    1.981527
2000-01-09    1.473561
2000-01-10    0.263927
Freq: D, Name: A, dtype: float64
```

Transform with multiple functions

Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [184]: tsdf.transform([np.abs, lambda x: x + 1])
Out[184]: 
                   A                   B                   C          
            absolute  <lambda>  absolute  <lambda>  absolute  <lambda>
2000-01-01  0.279295  0.720705  0.417231  1.417231  0.537995  1.537995
2000-01-02  0.807265  1.807265  1.387190  2.387190  0.982545  0.017455
2000-01-03  0.538076  0.461924  0.192555  1.192555  1.012581 -0.012581
2000-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  1.981527 -0.981527  0.274648  1.274648  0.747219  0.252781
2000-01-09  1.473561  2.473561  0.232948  1.232948  0.192742  1.192742
2000-01-10  0.263927  1.263927  0.153731  0.846269  0.831473  1.831473
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [185]: tsdf.A.transform([np.abs, lambda x: x + 1])
Out[185]: 
            absolute  <lambda>
2000-01-01  0.279295  0.720705
```

(continues on next page)

(continued from previous page)

```
2000-01-02  0.807265  1.807265
2000-01-03  0.538076  0.461924
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  1.981527 -0.981527
2000-01-09  1.473561  2.473561
2000-01-10  0.263927  1.263927
```

Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [186]: tsdf.transform({'A': np.abs, 'B': lambda x: x + 1})
Out[186]:
                   A         B
2000-01-01  0.279295  1.417231
2000-01-02  0.807265  2.387190
2000-01-03  0.538076  1.192555
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  1.981527  1.274648
2000-01-09  1.473561  1.232948
2000-01-10  0.263927  0.846269
```

Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms.

```
In [187]: tsdf.transform({'A': np.abs, 'B': [lambda x: x + 1, 'sqrt']})
Out[187]:
                   A         B
            absolute  <lambda>      sqrt
2000-01-01  0.279295  1.417231  0.645935
2000-01-02  0.807265  2.387190  1.177790
2000-01-03  0.538076  1.192555  0.438811
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  1.981527  1.274648  0.524068
2000-01-09  1.473561  1.232948  0.482647
2000-01-10  0.263927  0.846269       NaN
```

Applying elementwise 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 [188]: df4
Out[188]:
         one       two     three
```

```
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [189]: def f(x):
   .....:     return len(str(x))
   .....:

In [190]: df4['one'].map(f)
Out[190]:
a    19
b    18
c    18
d     3
Name: one, dtype: int64

In [191]: df4.applymap(f)
Out[191]:
   one  two  three
a   19   19      3
b   18   18     18
c   18   20     19
d    3   18     18
```

`Series.map()` has an additional feature; it can be used to easily link or map values defined by a secondary series. This is closely related to *merging/joining functionality*:

```
In [192]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'],
   .....:               index=['a', 'b', 'c', 'd', 'e'])
   .....:

In [193]: t = pd.Series({'six': 6., 'seven': 7.})

In [194]: s
Out[194]:
a      six
b    seven
c      six
d    seven
e      six
dtype: object

In [195]: s.map(t)
Out[195]:
a    6.0
b    7.0
c    6.0
d    7.0
e    6.0
dtype: float64
```

3.3.7 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 [196]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [197]: s
Out[197]:
a   -1.577151
b    1.165881
c    0.408422
d    1.733967
e   -1.179138
dtype: float64

In [198]: s.reindex(['e', 'b', 'f', 'd'])
Out[198]:
e   -1.179138
b    1.165881
f         NaN
d    1.733967
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 [199]: df
Out[199]:
        one       two     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [200]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[200]:
      three       two       one
c -1.233237 -0.081301  1.009711
f       NaN       NaN       NaN
b  1.063425  0.685821  1.169430
```

You may also use `reindex` with an `axis` keyword:

```
In [201]: df.reindex(['c', 'f', 'b'], axis='index')
Out[201]:
        one       two     three
c  1.009711 -0.081301 -1.233237
```

(continues on next page)

(continued from previous page)

```
f       NaN       NaN       NaN
b  1.169430  0.685821  1.063425
```

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

In [203]: rs
Out[203]:
a   -1.577151
b    1.165881
c    0.408422
d    1.733967
dtype: float64

In [204]: rs.index is df.index
Out[204]: True
```

This means that the reindexed Seriess index is the same Python object as the DataFrames index.

New in version 0.21.0.

`DataFrame.reindex()` also supports an axis-style calling convention, where you specify a single `labels` argument and the `axis` it applies to.

```
In [205]: df.reindex(['c', 'f', 'b'], axis='index')
Out[205]:
        one       two     three
c  1.009711 -0.081301 -1.233237
f       NaN       NaN       NaN
b  1.169430  0.685821  1.063425

In [206]: df.reindex(['three', 'two', 'one'], axis='columns')
Out[206]:
      three       two       one
a       NaN  0.244345  0.280390
b  1.063425  0.685821  1.169430
c -1.233237 -0.081301  1.009711
d  0.853134  0.589941       NaN
```

See also:

MultiIndex / Advanced Indexing is an even more concise way of doing reindexing.

Note: When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

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 [207]: df2
Out[207]:
        one       two
a  0.280390  0.244345
b  1.169430  0.685821
c  1.009711 -0.081301

In [208]: df3
Out[208]:
        one       two
a -0.539453 -0.038610
b  0.349586  0.402866
c  0.189867 -0.364256

In [209]: df.reindex_like(df2)
Out[209]:
        one       two
a  0.280390  0.244345
b  1.169430  0.685821
c  1.009711 -0.081301
```

Aligning objects with each other with `align`

The `align()` method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling objects index
- `join='right'`: use the passed objects index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [210]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [211]: s1 = s[:4]

In [212]: s2 = s[1:]

In [213]: s1.align(s2)
Out[213]:
(a   -0.673094
 b    1.111754
 c    0.936573
 d    1.666941
 e         NaN
 dtype: float64, a         NaN
 b    1.111754
 c    0.936573
 d    1.666941
 e   -0.170962
 dtype: float64)
```

```
In [214]: s1.align(s2, join='inner')
Out[214]:
(b    1.111754
 c    0.936573
 d    1.666941
 dtype: float64, b    1.111754
 c    0.936573
 d    1.666941
 dtype: float64)

In [215]: s1.align(s2, join='left')
Out[215]:
(a   -0.673094
 b    1.111754
 c    0.936573
 d    1.666941
 dtype: float64, a         NaN
 b    1.111754
 c    0.936573
 d    1.666941
 dtype: float64)
```

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

```
In [216]: df.align(df2, join='inner')
Out[216]:
(        one       two
 a  0.280390  0.244345
 b  1.169430  0.685821
 c  1.009711 -0.081301,         one       two
 a  0.280390  0.244345
 b  1.169430  0.685821
 c  1.009711 -0.081301)
```

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

```
In [217]: df.align(df2, join='inner', axis=0)
Out[217]:
(        one       two     three
 a  0.280390  0.244345       NaN
 b  1.169430  0.685821  1.063425
 c  1.009711 -0.081301 -1.233237,         one       two
 a  0.280390  0.244345
 b  1.169430  0.685821
 c  1.009711 -0.081301)
```

If you pass a Series to DataFrame.align(), you can choose to align both objects either on the DataFrames index or columns using the axis argument:

```
In [218]: df.align(df2.iloc[0], axis=1)
Out[218]:
(        one     three       two
 a  0.280390       NaN  0.244345
 b  1.169430  1.063425  0.685821
 c  1.009711 -1.233237 -0.081301
```

(continues on next page)

(continued from previous page)

```
d       NaN  0.853134  0.589941, one      0.280390
three         NaN
two      0.244345
Name: a, dtype: float64)
```

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
nearest	Fill from the nearest index value

We illustrate these fill methods on a simple Series:

```
In [219]: rng = pd.date_range('1/3/2000', periods=8)

In [220]: ts = pd.Series(np.random.randn(8), index=rng)

In [221]: ts2 = ts[[0, 3, 6]]

In [222]: ts
Out[222]: 
2000-01-03   -1.352366
2000-01-04   -0.102184
2000-01-05   -0.992602
2000-01-06   -0.324007
2000-01-07   -0.589906
2000-01-08   -1.324784
2000-01-09    1.426685
2000-01-10    0.213322
Freq: D, dtype: float64

In [223]: ts2
Out[223]: 
2000-01-03   -1.352366
2000-01-06   -0.324007
2000-01-09    1.426685
dtype: float64

In [224]: ts2.reindex(ts.index)
Out[224]: 
2000-01-03   -1.352366
2000-01-04         NaN
2000-01-05         NaN
2000-01-06   -0.324007
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    1.426685
2000-01-10         NaN
Freq: D, dtype: float64
```

```
In [225]: ts2.reindex(ts.index, method='ffill')
Out[225]:
2000-01-03   -1.352366
2000-01-04   -1.352366
2000-01-05   -1.352366
2000-01-06   -0.324007
2000-01-07   -0.324007
2000-01-08   -0.324007
2000-01-09    1.426685
2000-01-10    1.426685
Freq: D, dtype: float64

In [226]: ts2.reindex(ts.index, method='bfill')
Out[226]:
2000-01-03   -1.352366
2000-01-04   -0.324007
2000-01-05   -0.324007
2000-01-06   -0.324007
2000-01-07    1.426685
2000-01-08    1.426685
2000-01-09    1.426685
2000-01-10         NaN
Freq: D, dtype: float64

In [227]: ts2.reindex(ts.index, method='nearest')
Out[227]:
2000-01-03   -1.352366
2000-01-04   -1.352366
2000-01-05   -0.324007
2000-01-06   -0.324007
2000-01-07   -0.324007
2000-01-08    1.426685
2000-01-09    1.426685
2000-01-10    1.426685
Freq: D, dtype: float64
```

These methods require that the indexes are **ordered** increasing or decreasing.

Note that the same result could have been achieved using *fillna* (except for `method='nearest'`) or *interpolate*:

```
In [228]: ts2.reindex(ts.index).fillna(method='ffill')
Out[228]:
2000-01-03   -1.352366
2000-01-04   -1.352366
2000-01-05   -1.352366
2000-01-06   -0.324007
2000-01-07   -0.324007
2000-01-08   -0.324007
2000-01-09    1.426685
2000-01-10    1.426685
Freq: D, dtype: float64
```

`reindex()` will raise a ValueError if the index is not monotonically increasing or decreasing. `fillna()` and `interpolate()` will not perform any checks on the order of the index.

Limits on filling while reindexing

The limit and tolerance arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches:

```
In [229]: ts2.reindex(ts.index, method='ffill', limit=1)
Out[229]:
2000-01-03   -1.352366
2000-01-04   -1.352366
2000-01-05         NaN
2000-01-06   -0.324007
2000-01-07   -0.324007
2000-01-08         NaN
2000-01-09    1.426685
2000-01-10    1.426685
Freq: D, dtype: float64
```

In contrast, tolerance specifies the maximum distance between the index and indexer values:

```
In [230]: ts2.reindex(ts.index, method='ffill', tolerance='1 day')
Out[230]:
2000-01-03   -1.352366
2000-01-04   -1.352366
2000-01-05         NaN
2000-01-06   -0.324007
2000-01-07   -0.324007
2000-01-08         NaN
2000-01-09    1.426685
2000-01-10    1.426685
Freq: D, dtype: float64
```

Notice that when used on a DatetimeIndex, TimedeltaIndex or PeriodIndex, tolerance will coerced into a Timedelta if possible. This allows you to specify tolerance with appropriate strings.

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 [231]: df
Out[231]:
        one       two     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [232]: df.drop(['a', 'd'], axis=0)
Out[232]:
        one       two     three
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237

In [233]: df.drop(['one'], axis=1)
Out[233]:
        two     three
a  0.244345       NaN
```

```
b  0.685821  1.063425
c -0.081301 -1.233237
d  0.589941  0.853134
```

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

```
In [234]: df.reindex(df.index.difference(['a', 'd']))
Out[234]: 
        one       two     three
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
```

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 [235]: s
Out[235]: 
a   -0.673094
b    1.111754
c    0.936573
d    1.666941
e   -0.170962
dtype: float64

In [236]: s.rename(str.upper)
Out[236]: 
A   -0.673094
B    1.111754
C    0.936573
D    1.666941
E   -0.170962
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). A dict or Series can also be used:

```
In [237]: df.rename(columns={'one': 'foo', 'two': 'bar'},
   .....:           index={'a': 'apple', 'b': 'banana', 'd': 'durian'})
   .....: 
Out[237]: 
             foo       bar     three
apple   0.280390  0.244345       NaN
banana  1.169430  0.685821  1.063425
c       1.009711 -0.081301 -1.233237
durian       NaN  0.589941  0.853134
```

If the mapping doesnt include a column/index label, it isnt renamed. Note that extra labels in the mapping dont throw an error.

New in version 0.21.0.

DataFrame.rename() also supports an axis-style calling convention, where you specify a single mapper and the axis to apply that mapping to.

```
In [238]: df.rename({'one': 'foo', 'two': 'bar'}, axis='columns')
Out[238]: 
        foo       bar     three
a  0.280390  0.244345       NaN
b  1.169430  0.685821  1.063425
c  1.009711 -0.081301 -1.233237
d       NaN  0.589941  0.853134

In [239]: df.rename({'a': 'apple', 'b': 'banana', 'd': 'durian'},␣
↪axis='index')
Out[239]: 
             one       two     three
apple   0.280390  0.244345       NaN
banana  1.169430  0.685821  1.063425
c       1.009711 -0.081301 -1.233237
durian       NaN  0.589941  0.853134
```

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.

New in version 0.18.0.

Finally, `rename()` also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [240]: s.rename("scalar-name")
Out[240]: 
a   -0.673094
b    1.111754
c    0.936573
d    1.666941
e   -0.170962
Name: scalar-name, dtype: float64
```

New in version 0.24.0.

The methods `rename_axis()` and `rename_axis()` allow specific names of a *MultiIndex* to be changed (as opposed to the labels).

```
In [241]: df = pd.DataFrame({'x': [1, 2, 3, 4, 5, 6],
   .....:                    'y': [10, 20, 30, 40, 50, 60]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b', 'c'],
↪ [1, 2]],
   .....:                   names=['let', 'num']))
   .....: 

In [242]: df
Out[242]: 
         x   y
let num       
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [243]: df.rename_axis(index={'let': 'abc'})
Out[243]: 
         x   y
```

```
abc num
a   1     1  10
    2     2  20
b   1     3  30
    2     4  40
c   1     5  50
    2     6  60

In [244]: df.rename_axis(index=str.upper)
Out[244]: 
          x   y
LET NUM        
a   1     1  10
    2     2  20
b   1     3  30
    2     4  40
c   1     5  50
    2     6  60
```

3.3.8 Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. DataFrames follow the dict-like convention of iterating over the keys of the objects.

In short, basic iteration (`for i in object`) produces:

- **Series**: values
- **DataFrame**: column labels

Thus, for example, iterating over a DataFrame gives you the column names:

```
In [245]: df = pd.DataFrame({'col1': np.random.randn(3),
   .....:                    'col2': np.random.randn(3)}, index=['a', 'b', 'c'])
   .....: 

In [246]: for col in df:
   .....:     print(col)
   .....: 
col1
col2
```

Pandas objects also have the dict-like `items()` method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning: Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing,
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use `apply()` instead of iterating over the values. See the docs on *function application*.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the *enhancing performance* section for some examples of this approach.

Warning: You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```
In [247]: df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})

In [248]: for index, row in df.iterrows():
   .....:     row['a'] = 10
   .....:

In [249]: df
Out[249]:
   a  b
0  1  a
1  2  b
2  3  c
```

items

Consistent with the dict-like interface, `items()` iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs

For example:

```
In [250]: for label, ser in df.items():
   .....:     print(label)
   .....:     print(ser)
   .....:
a
0    1
1    2
2    3
Name: a, dtype: int64
b
0    a
1    b
2    c
Name: b, dtype: object
```

iterrows

iterrows() allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```
In [251]: for row_index, row in df.iterrows():
   .....:     print(row_index, row, sep='\n')
   .....:
0
a    1
b    a
Name: 0, dtype: object
1
a    2
b    b
Name: 1, dtype: object
2
a    3
b    c
Name: 2, dtype: object
```

Note: Because iterrows() returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [252]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])

In [253]: df_orig.dtypes
Out[253]:
int        int64
float    float64
dtype: object

In [254]: row = next(df_orig.iterrows())[1]

In [255]: row
Out[255]:
int      1.0
float    1.5
Name: 0, dtype: float64
```

All values in row, returned as a Series, are now upcasted to floats, also the original integer value in column *x*:

```
In [256]: row['int'].dtype
Out[256]: dtype('float64')

In [257]: df_orig['int'].dtype
Out[257]: dtype('int64')
```

To preserve dtypes while iterating over the rows, it is better to use itertuples() which returns namedtuples of the values and which is generally much faster than iterrows().

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

```
In [258]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [259]: print(df2)
   x  y
```

```
0  1  4
1  2  5
2  3  6

In [260]: print(df2.T)
   0  1  2
x  1  2  3
y  4  5  6

In [261]: df2_t = pd.DataFrame({idx: values for idx, values in df2.iterrows()}
↪)

In [262]: print(df2_t)
   0  1  2
x  1  2  3
y  4  5  6
```

itertuples

The `itertuples()` method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the rows corresponding index value, while the remaining values are the row values.

For instance:

```
In [263]: for row in df.itertuples():
   .....:     print(row)
   .....:
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, `itertuples()` preserves the data type of the values and is generally faster as `iterrows()`.

Note: The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

3.3.9 .dt accessor

`Series` has an accessor to succinctly return datetime like properties for the *values* of the Series, if it is a datetime/period like Series. This will return a Series, indexed like the existing Series.

```
# datetime
In [264]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [265]: s
Out[265]:
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
dtype: datetime64[ns]
```

```
In [266]: s.dt.hour
Out[266]:
0    9
1    9
2    9
3    9
dtype: int64

In [267]: s.dt.second
Out[267]:
0    12
1    12
2    12
3    12
dtype: int64

In [268]: s.dt.day
Out[268]:
0    1
1    2
2    3
3    4
dtype: int64
```

This enables nice expressions like this:

```
In [269]: s[s.dt.day == 2]
Out[269]:
1   2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [270]: stz = s.dt.tz_localize('US/Eastern')

In [271]: stz
Out[271]:
0   2013-01-01 09:10:12-05:00
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]

In [272]: stz.dt.tz
Out[272]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

You can also chain these types of operations:

```
In [273]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[273]:
0   2013-01-01 04:10:12-05:00
1   2013-01-02 04:10:12-05:00
2   2013-01-03 04:10:12-05:00
3   2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also format datetime values as strings with `Series.dt.strftime()` which supports the same format as the standard strftime().

```
# DatetimeIndex
In [274]: s = pd.Series(pd.date_range('20130101', periods=4))

In [275]: s
Out[275]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
dtype: datetime64[ns]

In [276]: s.dt.strftime('%Y/%m/%d')
Out[276]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

```
# PeriodIndex
In [277]: s = pd.Series(pd.period_range('20130101', periods=4))

In [278]: s
Out[278]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [279]: s.dt.strftime('%Y/%m/%d')
Out[279]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

The .dt accessor works for period and timedelta dtypes.

```
# period
In [280]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [281]: s
Out[281]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [282]: s.dt.year
Out[282]:
0    2013
```

```
1    2013
2    2013
3    2013
dtype: int64

In [283]: s.dt.day
Out[283]:
0    1
1    2
2    3
3    4
dtype: int64
```

```
# timedelta
In [284]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4,␣
 ↪freq='s'))

In [285]: s
Out[285]:
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
3   1 days 00:00:08
dtype: timedelta64[ns]

In [286]: s.dt.days
Out[286]:
0    1
1    1
2    1
3    1
dtype: int64

In [287]: s.dt.seconds
Out[287]:
0    5
1    6
2    7
3    8
dtype: int64

In [288]: s.dt.components
Out[288]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0
```

Note: `Series.dt` will raise a `TypeError` if you access with a non-datetime-like values.

3.3.10 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Seriess `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```
In [289]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog
↪', 'cat'])

In [290]: s.str.lower()
Out[290]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them).

Please see *Vectorized String Methods* for a complete description.

3.3.11 Sorting

Pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

By index

The `Series.sort_index()` and `DataFrame.sort_index()` methods are used to sort a pandas object by its index levels.

```
In [291]: df = pd.DataFrame({
   .....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   .....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c',␣
↪'d']),
   .....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   .....:

In [292]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....:                          columns=['three', 'two', 'one'])
   .....:

In [293]: unsorted_df
Out[293]:
      three       two       one
a       NaN -0.910346  1.695397
d  1.795269 -0.834629       NaN
c -0.228802  0.349813 -0.668278
b -0.907416  0.713418  0.349313
```

```
# DataFrame
In [294]: unsorted_df.sort_index()
Out[294]:
      three       two       one
a       NaN -0.910346  1.695397
b -0.907416  0.713418  0.349313
c -0.228802  0.349813 -0.668278
d  1.795269 -0.834629       NaN

In [295]: unsorted_df.sort_index(ascending=False)
Out[295]:
      three       two       one
d  1.795269 -0.834629       NaN
c -0.228802  0.349813 -0.668278
b -0.907416  0.713418  0.349313
a       NaN -0.910346  1.695397

In [296]: unsorted_df.sort_index(axis=1)
Out[296]:
        one     three       two
a  1.695397       NaN -0.910346
d       NaN  1.795269 -0.834629
c -0.668278 -0.228802  0.349813
b  0.349313 -0.907416  0.713418

# Series
In [297]: unsorted_df['three'].sort_index()
Out[297]:
a         NaN
b   -0.907416
c   -0.228802
d    1.795269
Name: three, dtype: float64
```

By values

The `Series.sort_values()` method is used to sort a *Series* by its values. The `DataFrame.sort_values()` method is used to sort a *DataFrame* by its column or row values. The optional `by` parameter to `DataFrame.sort_values()` may used to specify one or more columns to use to determine the sorted order.

```
In [298]: df1 = pd.DataFrame({'one': [2, 1, 1, 1],
   .....:                     'two': [1, 3, 2, 4],
   .....:                     'three': [5, 4, 3, 2]})
   .....:

In [299]: df1.sort_values(by='two')
Out[299]:
   one  two  three
0    2    1      5
2    1    2      3
1    1    3      4
3    1    4      2
```

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

```
In [300]: df1[['one', 'two', 'three']].sort_values(by=['one', 'two'])
Out[300]: 
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

These methods have special treatment of NA values via the `na_position` argument:

```
In [301]: s[2] = np.nan

In [302]: s.sort_values()
Out[302]: 
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
2     NaN
5     NaN
dtype: object

In [303]: s.sort_values(na_position='first')
Out[303]: 
2     NaN
5     NaN
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
dtype: object
```

By indexes and values

New in version 0.23.0.

Strings passed as the `by` parameter to `DataFrame.sort_values()` may refer to either columns or index level names.

```
# Build MultiIndex
In [304]: idx = pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('a', 2),
   .....:                                  ('b', 2), ('b', 1), ('b', 1)])
   .....: 

In [305]: idx.names = ['first', 'second']

# Build DataFrame
In [306]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
```

(continues on next page)

(continued from previous page)

```
   .....:                              index=idx)
   .....:

In [307]: df_multi
Out[307]:
              A
first second
a     1       6
      2       5
      2       4
b     2       3
      1       2
      1       1
```

Sort by second (index) and A (column)

```
In [308]: df_multi.sort_values(by=['second', 'A'])
Out[308]:
              A
first second
b     1       1
      1       2
a     1       6
b     2       3
a     2       4
      2       5
```

Note: If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

searchsorted

Series has the `searchsorted()` method, which works similarly to `numpy.ndarray.searchsorted()`.

```
In [309]: ser = pd.Series([1, 2, 3])

In [310]: ser.searchsorted([0, 3])
Out[310]: array([0, 2])

In [311]: ser.searchsorted([0, 4])
Out[311]: array([0, 3])

In [312]: ser.searchsorted([1, 3], side='right')
Out[312]: array([1, 3])

In [313]: ser.searchsorted([1, 3], side='left')
Out[313]: array([0, 2])

In [314]: ser = pd.Series([3, 1, 2])

In [315]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[315]: array([0, 2])
```

smallest / largest values

Series has the nsmallest() and nlargest() methods which return the smallest or largest n values. For a large Series this can be much faster than sorting the entire Series and calling head(n) on the result.

```
In [316]: s = pd.Series(np.random.permutation(10))

In [317]: s
Out[317]:
0    4
1    7
2    8
3    1
4    9
5    3
6    6
7    0
8    2
9    5
dtype: int64

In [318]: s.sort_values()
Out[318]:
7    0
3    1
8    2
5    3
0    4
9    5
6    6
1    7
2    8
4    9
dtype: int64

In [319]: s.nsmallest(3)
Out[319]:
7    0
3    1
8    2
dtype: int64

In [320]: s.nlargest(3)
Out[320]:
4    9
2    8
1    7
dtype: int64
```

DataFrame also has the nlargest and nsmallest methods.

```
In [321]: df = pd.DataFrame({'a': [-2, -1, 1, 10, 8, 11, -1],
   .....:                    'b': list('abdceff'),
   .....:                    'c': [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0]})
   .....:
```

```
In [322]: df.nlargest(3, 'a')
Out[322]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN

In [323]: df.nlargest(5, ['a', 'c'])
Out[323]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN
2   1  d  4.0
6  -1  f  4.0

In [324]: df.nsmallest(3, 'a')
Out[324]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0

In [325]: df.nsmallest(5, ['a', 'c'])
Out[325]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0
2  1  d  4.0
4  8  e  NaN
```

Sorting by a MultiIndex column

You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to by.

```
In [326]: df1.columns = pd.MultiIndex.from_tuples([('a', 'one'),
   .....:                                           ('a', 'two'),
   .....:                                           ('b', 'three')])
   .....:

In [327]: df1.sort_values(by=('a', 'two'))
Out[327]:
    a         b
  one two three
0   2   1     5
2   1   2     3
1   1   3     4
3   1   4     2
```

3.3.12 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 method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

3.3.13 dtypes

For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for `float`, `int`, `bool`, `timedelta64[ns]` and `datetime64[ns]` (note that NumPy does not support timezone-aware datetimes).

Pandas and third-party libraries *extend* NumPys type system in a few places. This section describes the extensions pandas has made internally. See *Extension types* for how to write your own extension that works with pandas. See *Extension data types* for a list of third-party libraries that have implemented an extension.

The following table lists all of pandas extension types. See the respective documentation sections for more on each type.

Kind of Data	Data Type	Scalar	Array	Documentation
tz-aware datetime	`DatetimeTZDtype`	`Timestamp`	`arrays.DatetimeArray`	*Time zone handling*
Categorical	`CategoricalDtype`	(none)	`Categorical`	categorical
period (time spans)	`PeriodDtype`	`Period`	`arrays.PeriodArray`	*Time span representation*
sparse	`SparseDtype`	(none)	`arrays.SparseArray`	sparse
intervals	`IntervalDtype`	`Interval`	`arrays.IntervalArray`	*IntervalIndex*
nullable integer	`Int64Dtype`,	(none)	`arrays.IntegerArray`	*Nullable integer data type*

Pandas uses the `object` dtype for storing strings.

Finally, arbitrary objects may be stored using the `object` dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See *object conversion*).

A convenient `dtypes` attribute for DataFrame returns a Series with the data type of each column.

```
In [328]: dft = pd.DataFrame({'A': np.random.rand(3),
   .....:                     'B': 1,
   .....:                     'C': 'foo',
   .....:                     'D': pd.Timestamp('20010102'),
   .....:                     'E': pd.Series([1.0] * 3).astype('float32'),
   .....:                     'F': False,
   .....:                     'G': pd.Series([1] * 3, dtype='int8')})
   .....:

In [329]: dft
```

```
Out[329]:
          A  B    C          D    E      F  G
0  0.409005  1  foo 2001-01-02  1.0  False  1
1  0.620879  1  foo 2001-01-02  1.0  False  1
2  0.760605  1  foo 2001-01-02  1.0  False  1

In [330]: dft.dtypes
Out[330]:
A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
F              bool
G              int8
dtype: object
```

On a Series object, use the dtype attribute.

```
In [331]: dft['A'].dtype
Out[331]: dtype('float64')
```

If a pandas object contains data with 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 [332]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[332]:
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [333]: pd.Series([1, 2, 3, 6., 'foo'])
Out[333]:
0      1
1      2
2      3
3      6
4    foo
dtype: object
```

The number of columns of each type in a DataFrame can be found by calling DataFrame.dtypes.value_counts().

```
In [334]: dft.dtypes.value_counts()
Out[334]:
datetime64[ns]    1
object            1
int8              1
bool              1
float64           1
```

(continues on next page)

(continued from previous page)

```
int64              1
float32            1
dtype: int64
```

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 [335]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'],␣
↪dtype='float32')

In [336]: df1
Out[336]: 
          A
0  1.289112
1 -0.073190
2 -0.302878
3  1.074794
4  0.771071
5 -0.806465
6  0.145373
7 -0.325009

In [337]: df1.dtypes
Out[337]: 
A    float32
dtype: object

In [338]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8),␣
↪dtype='float16'),
   .....:                     'B': pd.Series(np.random.randn(8)),
   .....:                     'C': pd.Series(np.array(np.random.randn(8),
   .....:                                         dtype='uint8'))})
   .....: 

In [339]: df2
Out[339]: 
          A         B  C
0 -0.530273  1.006054  1
1  0.092712 -0.639761  0
2  0.259521 -0.577626  1
3  0.835449 -0.484128  0
4  0.478760  1.201647  0
5 -0.703613 -1.065766  0
6  0.287598  0.098521  0
7  0.812988 -1.754450  0

In [340]: df2.dtypes
Out[340]: 
A    float16
B    float64
C      uint8
dtype: object
```

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 [341]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[341]:
a    int64
dtype: object

In [342]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[342]:
a    int64
dtype: object

In [343]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[343]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [344]: frame = pd.DataFrame(np.array([1, 2]))
```

upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [345]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [346]: df3
Out[346]:
          A         B    C
0  0.758839  1.006054  1.0
1  0.019523 -0.639761  0.0
2 -0.043356 -0.577626  1.0
3  1.910244 -0.484128  0.0
4  1.249831  1.201647  0.0
5 -1.510078 -1.065766  0.0
6  0.432970  0.098521  0.0
7  0.487979 -1.754450  0.0

In [347]: df3.dtypes
Out[347]:
A    float32
B    float64
C    float64
dtype: object
```

`DataFrame.to_numpy()` will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped NumPy array. This can force some *upcasting*.

```
In [348]: df3.to_numpy().dtype
Out[348]: dtype('float64')
```

astype

You can use the `astype()` method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **numpy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [349]: df3
Out[349]:
          A         B    C
0  0.758839  1.006054  1.0
1  0.019523 -0.639761  0.0
2 -0.043356 -0.577626  1.0
3  1.910244 -0.484128  0.0
4  1.249831  1.201647  0.0
5 -1.510078 -1.065766  0.0
6  0.432970  0.098521  0.0
7  0.487979 -1.754450  0.0

In [350]: df3.dtypes
Out[350]:
A    float32
B    float64
C    float64
dtype: object

# conversion of dtypes
In [351]: df3.astype('float32').dtypes
Out[351]:
A    float32
B    float32
C    float32
dtype: object
```

Convert a subset of columns to a specified type using `astype()`.

```
In [352]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [353]: dft[['a', 'b']] = dft[['a', 'b']].astype(np.uint8)

In [354]: dft
Out[354]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [355]: dft.dtypes
Out[355]:
```

```
a    uint8
b    uint8
c    int64
dtype: object
```

New in version 0.19.0.

Convert certain columns to a specific dtype by passing a dict to `astype()`.

```
In [356]: dft1 = pd.DataFrame({'a': [1, 0, 1], 'b': [4, 5, 6], 'c': [7, 8, 9]}
↪)

In [357]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})

In [358]: dft1
Out[358]:
       a  b    c
0   True  4  7.0
1  False  5  8.0
2   True  6  9.0

In [359]: dft1.dtypes
Out[359]:
a       bool
b      int64
c    float64
dtype: object
```

Note: When trying to convert a subset of columns to a specified type using `astype()` and `loc()`, upcasting occurs.

`loc()` tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```
In [360]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [361]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out[361]:
a    uint8
b    uint8
dtype: object

In [362]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [363]: dft.dtypes
Out[363]:
a    int64
b    int64
c    int64
dtype: object
```

object conversion

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the `DataFrame.infer_objects()` and `Series.infer_objects()` methods can be used to soft convert to the correct type.

```
In [364]: import datetime

In [365]: df = pd.DataFrame([[1, 2],
   .....:                    ['a', 'b'],
   .....:                    [datetime.datetime(2016, 3, 2),
   .....:                     datetime.datetime(2016, 3, 2)]])
   .....:

In [366]: df = df.T

In [367]: df
Out[367]:
   0  1          2
0  1  a 2016-03-02
1  2  b 2016-03-02

In [368]: df.dtypes
Out[368]:
0            object
1            object
2    datetime64[ns]
dtype: object
```

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

```
In [369]: df.infer_objects().dtypes
Out[369]:
0             int64
1            object
2    datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- *`to_numeric()`* (conversion to numeric dtypes)

```
In [370]: m = ['1.1', 2, 3]

In [371]: pd.to_numeric(m)
Out[371]: array([1.1, 2. , 3. ])
```

- *`to_datetime()`* (conversion to datetime objects)

```
In [372]: import datetime

In [373]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]

In [374]: pd.to_datetime(m)
Out[374]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]',␣
↪freq=None)
```

- *`to_timedelta()`* (conversion to timedelta objects)

```
In [375]: m = ['5us', pd.Timedelta('1day')]
```

(continues on next page)

(continued from previous page)

```
In [376]: pd.to_timedelta(m)
Out[376]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'], dtype=
↪'timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [377]: import datetime

In [378]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [379]: pd.to_datetime(m, errors='coerce')
Out[379]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [380]: m = ['apple', 2, 3]

In [381]: pd.to_numeric(m, errors='coerce')
Out[381]: array([nan,  2.,  3.])

In [382]: m = ['apple', pd.Timedelta('1day')]

In [383]: pd.to_timedelta(m, errors='coerce')
Out[383]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [384]: import datetime

In [385]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [386]: pd.to_datetime(m, errors='ignore')
Out[386]: Index(['apple', 2016-03-02 00:00:00], dtype='object')

In [387]: m = ['apple', 2, 3]

In [388]: pd.to_numeric(m, errors='ignore')
Out[388]: array(['apple', 2, 3], dtype=object)

In [389]: m = ['apple', pd.Timedelta('1day')]

In [390]: pd.to_timedelta(m, errors='ignore')
Out[390]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, *`to_numeric()`* provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [391]: m = ['1', 2, 3]

In [392]: pd.to_numeric(m, downcast='integer')   # smallest signed int dtype
Out[392]: array([1, 2, 3], dtype=int8)

In [393]: pd.to_numeric(m, downcast='signed')    # same as 'integer'
```

```
Out[393]: array([1, 2, 3], dtype=int8)

In [394]: pd.to_numeric(m, downcast='unsigned')  # smallest unsigned int dtype
Out[394]: array([1, 2, 3], dtype=uint8)

In [395]: pd.to_numeric(m, downcast='float')     # smallest float dtype
Out[395]: array([1., 2., 3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with `apply()`, we can apply the function over each column efficiently:

```
In [396]: import datetime

In [397]: df = pd.DataFrame([
   .....:     ['2016-07-09', datetime.datetime(2016, 3, 2)]] * 2, dtype='O')
   .....:

In [398]: df
Out[398]: 
            0                    1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [399]: df.apply(pd.to_datetime)
Out[399]: 
           0          1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [400]: df = pd.DataFrame([['1.1', 2, 3]] * 2, dtype='O')

In [401]: df
Out[401]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [402]: df.apply(pd.to_numeric)
Out[402]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [403]: df = pd.DataFrame([['5us', pd.Timedelta('1day')]] * 2, dtype='O')

In [404]: df
Out[404]: 
     0                1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00

In [405]: df.apply(pd.to_timedelta)
Out[405]: 
                0      1
```

```
0 00:00:00.000005 1 days
1 00:00:00.000005 1 days
```

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. See also *Support for integer NA*.

```
In [406]: dfi = df3.astype('int32')

In [407]: dfi['E'] = 1

In [408]: dfi
Out[408]:
   A  B  C  E
0  0  1  1  1
1  0  0  0  1
2  0  0  1  1
3  1  0  0  1
4  1  1  0  1
5 -1 -1  0  1
6  0  0  0  1
7  0 -1  0  1

In [409]: dfi.dtypes
Out[409]:
A    int32
B    int32
C    int32
E    int64
dtype: object

In [410]: casted = dfi[dfi > 0]

In [411]: casted
Out[411]:
     A    B    C  E
0  NaN  1.0  1.0  1
1  NaN  NaN  NaN  1
2  NaN  NaN  1.0  1
3  1.0  NaN  NaN  1
4  1.0  1.0  NaN  1
5  NaN  NaN  NaN  1
6  NaN  NaN  NaN  1
7  NaN  NaN  NaN  1

In [412]: casted.dtypes
Out[412]:
A    float64
B    float64
C    float64
E      int64
dtype: object
```

While float dtypes are unchanged.

```
In [413]: dfa = df3.copy()

In [414]: dfa['A'] = dfa['A'].astype('float32')

In [415]: dfa.dtypes
Out[415]:
A    float32
B    float64
C    float64
dtype: object

In [416]: casted = dfa[df2 > 0]

In [417]: casted
Out[417]:
          A         B    C
0       NaN  1.006054  1.0
1  0.019523       NaN  NaN
2 -0.043356       NaN  1.0
3  1.910244       NaN  NaN
4  1.249831  1.201647  NaN
5       NaN       NaN  NaN
6  0.432970  0.098521  NaN
7  0.487979       NaN  NaN

In [418]: casted.dtypes
Out[418]:
A    float32
B    float64
C    float64
dtype: object
```

3.3.14 Selecting columns based on `dtype`

The `select_dtypes()` method implements subsetting of columns based on their `dtype`.

First, lets create a `DataFrame` with a slew of different dtypes:

```
In [419]: df = pd.DataFrame({'string': list('abc'),
   .....:                    'int64': list(range(1, 4)),
   .....:                    'uint8': np.arange(3, 6).astype('u1'),
   .....:                    'float64': np.arange(4.0, 7.0),
   .....:                    'bool1': [True, False, True],
   .....:                    'bool2': [False, True, False],
   .....:                    'dates': pd.date_range('now', periods=3),
   .....:                    'category': pd.Series(list("ABC")).astype('category')})
   .....:

In [420]: df['tdeltas'] = df.dates.diff()

In [421]: df['uint64'] = np.arange(3, 6).astype('u8')

In [422]: df['other_dates'] = pd.date_range('20130101', periods=3)
```

(continues on next page)

(continued from previous page)

```
In [423]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')

In [424]: df
Out[424]: 
  string  int64  uint8  float64  bool1  ...  category tdeltas uint64 other_dates      ␣
↪       tz_aware_dates
0      a      1      3      4.0   True  ...         A     NaT      3  2013-01-01 2013-
↪01-01 00:00:00-05:00
1      b      2      4      5.0  False  ...         B  1 days      4  2013-01-02 2013-
↪01-02 00:00:00-05:00
2      c      3      5      6.0   True  ...         C  1 days      5  2013-01-03 2013-
↪01-03 00:00:00-05:00

[3 rows x 12 columns]
```

And the dtypes:

```
In [425]: df.dtypes
Out[425]: 
string                                object
int64                                  int64
uint8                                  uint8
float64                              float64
bool1                                   bool
bool2                                   bool
dates                         datetime64[ns]
category                            category
tdeltas                      timedelta64[ns]
uint64                                uint64
other_dates                   datetime64[ns]
tz_aware_dates    datetime64[ns, US/Eastern]
dtype: object
```

`select_dtypes()` has two parameters `include` and `exclude` that allow you to say give me the columns *with* these dtypes (`include`) and/or give the columns *without* these dtypes (`exclude`).

For example, to select `bool` columns:

```
In [426]: df.select_dtypes(include=[bool])
Out[426]: 
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the NumPy dtype hierarchy:

```
In [427]: df.select_dtypes(include=['bool'])
Out[427]: 
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

`select_dtypes()` also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [428]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[428]: 
   int64  float64  bool1  bool2 tdeltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the `object` dtype:

```
In [429]: df.select_dtypes(include=['object'])
Out[429]: 
  string
0      a
1      b
2      c
```

To see all the child dtypes of a generic `dtype` like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [430]: def subdtypes(dtype):
   .....:     subs = dtype.__subclasses__()
   .....:     if not subs:
   .....:         return dtype
   .....:     return [dtype, [subdtypes(dt) for dt in subs]]
   .....: 
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [431]: subdtypes(np.generic)
Out[431]: 
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.int64,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
       [numpy.uint8,
        numpy.uint16,
        numpy.uint32,
        numpy.uint64,
        numpy.uint64]]]],
    [numpy.inexact,
     [[numpy.floating,
       [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
      [numpy.complexfloating,
       [numpy.complex64, numpy.complex128, numpy.complex256]]]]]],
  [numpy.flexible,
   [[numpy.character, [numpy.bytes_, numpy.str_]],
    [numpy.void, [numpy.record]]]],
  numpy.bool_,
  numpy.datetime64,
  numpy.object_]]
```

Note: Pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and wont show up with the above function.

{{ header }}

3.4 Intro to data structures

Well 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

In [2]: import pandas as pd
```

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.

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

3.4.1 Series

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

```
>>> s = pd.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 [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [4]: s
Out[4]:
a    1.585681
b   -0.737876
c   -1.603758
d   -0.668269
e   -1.104086
dtype: float64
```

```
In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
0    0.873370
1   -1.516058
2    1.625341
3    0.561420
4   -0.009922
dtype: float64
```

Note: 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

Series can be instantiated from dicts:

```
In [7]: d = {'b': 1, 'a': 0, 'c': 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

Note: When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dicts insertion order, if youre using Python version >= 3.6 and Pandas version >= 0.23.

If youre using Python < 3.6 or Pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a Pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {'a': 0., 'b': 1., 'c': 2.}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=['b', 'c', 'd', 'a'])
Out[11]:
b    1.0
c    2.0
d    NaN
```

```
a    0.0
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 [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]: 
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 1.5856810982636127

In [14]: s[:3]
Out[14]: 
a    1.585681
b   -0.737876
c   -1.603758
dtype: float64

In [15]: s[s > s.median()]
Out[15]: 
a    1.585681
d   -0.668269
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]: 
e   -1.104086
d   -0.668269
b   -0.737876
dtype: float64

In [17]: np.exp(s)
Out[17]: 
a    4.882616
b    0.478129
c    0.201139
d    0.512595
e    0.331514
dtype: float64
```

Note: We will address array-based indexing like `s[[4, 3, 1]]` in *section*.

Like a NumPy array, a pandas Series has a `dtype`.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPys type system in a few places, in which case the dtype would be a *ExtensionDtype*. Some examples within pandas are categorical and *Nullable integer data type*. See *dtypes* for more.

If you need the actual array backing a `Series`, use `Series.array`.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 1.5856810982636127, -0.7378757380201878, -1.6037579905058967,
  -0.668268701321749, -1.1040860124173795]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable *automatic alignment*, for example).

`Series.array` will always be an *ExtensionArray*. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a `numpy.ndarray`. Pandas knows how to take an `ExtensionArray` and store it in a `Series` or a column of a `DataFrame`. See *dtypes* for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use `Series.to_numpy()`.

```
In [20]: s.to_numpy()
Out[20]: array([ 1.5856811 , -0.73787574, -1.60375799, -0.6682687 , -1.10408601])
```

Even if the Series is backed by a *ExtensionArray*, `Series.to_numpy()` will return a NumPy ndarray.

Series is dict-like

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

```
In [21]: s['a']
Out[21]: 1.5856810982636127

In [22]: s['e'] = 12.

In [23]: s
Out[23]:
a     1.585681
b    -0.737876
c    -1.603758
d    -0.668269
e    12.000000
dtype: float64

In [24]: 'e' in s
Out[24]: True
```

```
In [25]: 'f' in s
Out[25]: 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 [26]: s.get('f')

In [27]: s.get('f', np.nan)
Out[27]: nan
```

See also the *section on attribute access*.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a     3.171362
b    -1.475751
c    -3.207516
d    -1.336537
e    24.000000
dtype: float64

In [29]: s * 2
Out[29]:
a     3.171362
b    -1.475751
c    -3.207516
d    -1.336537
e    24.000000
dtype: float64

In [30]: np.exp(s)
Out[30]:
a         4.882616
b         0.478129
c         0.201139
d         0.512595
e    162754.791419
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 [31]: s[1:] + s[:-1]
Out[31]:
```

(continues on next page)

(continued from previous page)

```
a         NaN
b   -1.475751
c   -3.207516
d   -1.336537
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.

Name attribute

Series can also have a `name` attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name='something')

In [33]: s
Out[33]:
0   -0.987980
1    1.649534
2   -0.333885
3    0.303428
4    1.388549
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

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

New in version 0.18.0.

You can rename a Series with the *`pandas.Series.rename()`* method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that `s` and `s2` refer to different objects.

3.4.2 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.

Note: When the data is a dict, and `columns` is not specified, the `DataFrame` columns will be ordered by the dicts insertion order, if you are using Python version >= 3.6 and Pandas >= 0.23.

If you are using Python < 3.6 or Pandas < 0.23, and `columns` is not specified, the `DataFrame` columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

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

```
In [37]: d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   ....:      'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
   ....:

In [38]: df = pd.DataFrame(d)

In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=['d', 'b', 'a'])
Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   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 [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```

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

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C',␣
↪'a10')])

In [48]: data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

In [49]: pd.DataFrame(data)
Out[49]:
   A    B         C
```

```
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=['first', 'second'])
Out[50]: 
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=['C', 'A', 'B'])
Out[51]: 
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

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

From a list of dicts

```
In [52]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [53]: pd.DataFrame(data2)
Out[53]: 
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=['first', 'second'])
Out[54]: 
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=['a', 'b'])
Out[55]: 
   a   b
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:               ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:               ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:               ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:               ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
   ....: 
Out[56]: 
       a              b
```

(continues on next page)

(continued from previous page)

```
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

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, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

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.

```
In [57]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]))
Out[57]: 
   A  B
0  1  4
1  2  5
2  3  6
```

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

```
In [58]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]),
   ....:                        orient='index', columns=['one', 'two', 'three'])
   ....: 
Out[58]: 
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal `DataFrame` constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [59]: data
Out[59]: 
array([(1, 2., b'Hello'), (2, 3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [60]: pd.DataFrame.from_records(data, index='C')
Out[60]: 
```

```
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

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 [61]: df['one']
Out[61]:
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [62]: df['three'] = df['one'] * df['two']

In [63]: df['flag'] = df['one'] > 2

In [64]: df
Out[64]:
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [65]: del df['two']

In [66]: three = df.pop('three')

In [67]: df
Out[67]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

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

```
In [68]: df['foo'] = 'bar'

In [69]: df
Out[69]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   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 DataFrames index:

```
In [70]: df['one_trunc'] = df['one'][:2]

In [71]: df
Out[71]: 
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrames index.

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

```
In [72]: df.insert(1, 'bar', df['one'])

In [73]: df
Out[73]: 
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning new columns in method chains

Inspired by dplyrs `mutate` verb, DataFrame has an *`assign()`* method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [74]: iris = pd.read_csv('data/iris.data')

In [75]: iris.head()
Out[75]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [76]: (iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength'])
   ....:      .head())
   ....: 
Out[76]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

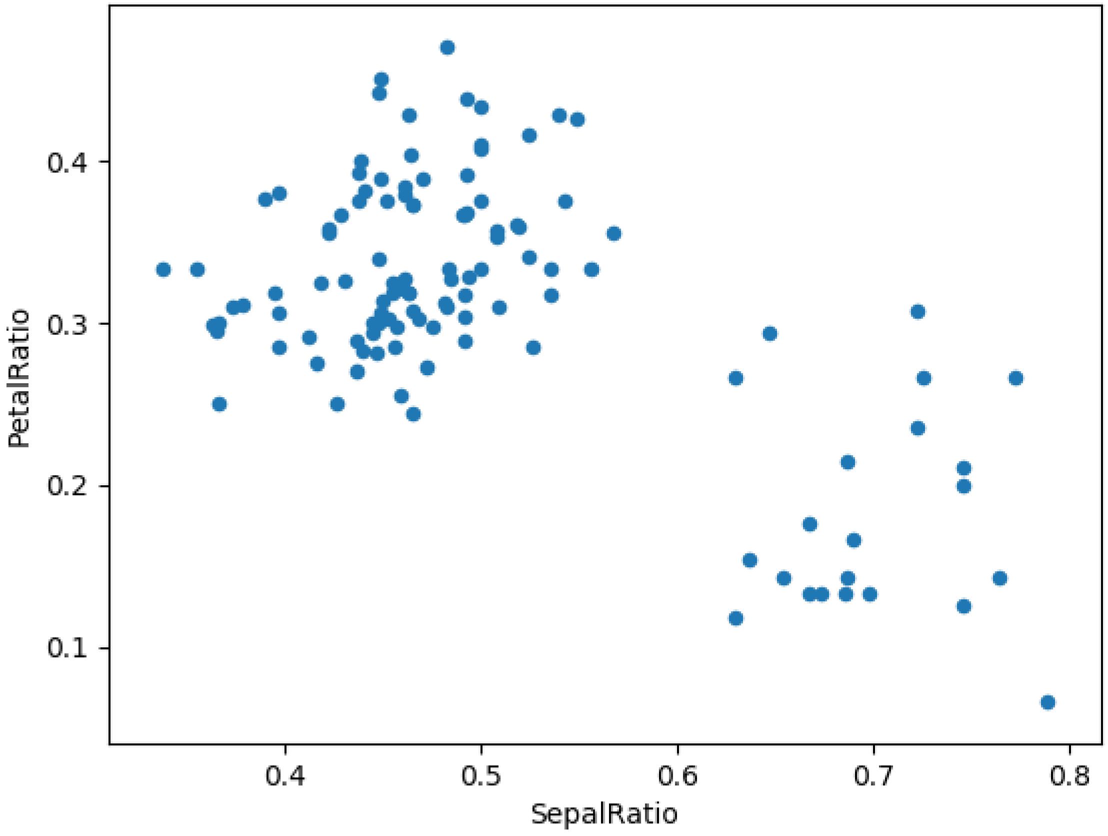
In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [77]: iris.assign(sepal_ratio=lambda x: (x['SepalWidth'] / x['SepalLength'])).
↪head()
Out[77]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

`assign` **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you dont have a reference to the DataFrame at hand. This is common when using `assign` in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [78]: (iris.query('SepalLength > 5')
   ....:      .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:              PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
   ....:      .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
   ....:
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x129542a90>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame thats been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didnt have a reference to the *filtered* DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. A *copy* of the original DataFrame is returned, with the new values inserted.

Changed in version 0.23.0.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```
In [79]: dfa = pd.DataFrame({"A": [1, 2, 3],
   ....:                     "B": [4, 5, 6]})
   ....:

In [80]: dfa.assign(C=lambda x: x['A'] + x['B'],
   ....:            D=lambda x: x['A'] + x['C'])
   ....:
Out[80]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, `x['C']` will refer to the newly created column, thats equal to `dfa['A'] + dfa['B']`.

To write code compatible with all versions of Python, split the assignment in two.

```
In [81]: dependent = pd.DataFrame({"A": [1, 1, 1]})

In [82]: (dependent.assign(A=lambda x: x['A'] + 1)
   ....:           .assign(B=lambda x: x['A'] + 2))
   ....:
Out[82]:
   A  B
0  2  4
1  2  4
2  2  4
```

Warning: Dependent assignment may subtly change the behavior of your code between Python 3.6 and older versions of Python.

If you wish to write code that supports versions of python before and after 3.6, youll need to take care when passing `assign` expressions that

- Update an existing column
- Refer to the newly updated column in the same `assign`

For example, well update column A and then refer to it when creating B.

```
>>> dependent = pd.DataFrame({"A": [1, 1, 1]})
>>> dependent.assign(A=lambda x: x["A"] + 1, B=lambda x: x["A"] + 2)
```

For Python 3.5 and earlier the expression creating `B` refers to the old value of `A`, `[1, 1, 1]`. The output is then

```
   A  B
0  2  3
1  2  3
2  2  3
```

For Python 3.6 and later, the expression creating `A` refers to the new value of `A`, `[2, 2, 2]`, which results in

```
   A  B
0  2  4
1  2  4
2  2  4
```

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 [83]: df.loc['b']
Out[83]:
one              2
bar              2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object

In [84]: df.iloc[2]
Out[84]:
one             3
bar             3
flag         True
foo           bar
one_trunc     NaN
Name: c, dtype: object
```

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

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 [85]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])

In [86]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])

In [87]: df + df2
Out[87]:
```

(continues on next page)

(continued from previous page)

```
          A         B         C   D
0 -2.386114 -1.656789  0.925576 NaN
1 -0.504949 -0.468878 -1.876635 NaN
2  0.230231 -3.873474  2.610702 NaN
3 -0.564199  0.101636  0.369056 NaN
4  1.322597  0.820841 -1.877026 NaN
5 -0.311563 -0.185355  0.628811 NaN
6  1.943183 -0.098391  0.250144 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 [88]: df - df.iloc[0]
Out[88]: 
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1  1.167144  0.019752 -0.477114 -0.905114
2  1.506963 -2.859436  2.505428  1.842367
3  2.289411  0.669679  2.262291  0.641429
4  1.516210  1.316061 -1.161409  3.026566
5  1.333741 -0.282192  0.843573  0.798528
6  2.657332  1.228825 -0.475312  2.257556
7  1.965229  0.851194  2.054340 -0.250865
8  0.802379  0.151740 -0.152969  1.492131
9  0.620426  2.396284 -2.063297 -0.284884
```

In the special case of working with time series data, if the DataFrame index contains dates, the broadcasting will be column-wise:

```
In [89]: index = pd.date_range('1/1/2000', periods=8)

In [90]: df = pd.DataFrame(np.random.randn(8, 3), index=index,␣
↪columns=list('ABC'))

In [91]: df
Out[91]: 
                   A         B         C
2000-01-01  0.780948  0.587141  0.386433
2000-01-02 -1.150341  0.553978  1.047461
2000-01-03  0.021671 -1.720821  0.192664
2000-01-04  1.058264  0.507151 -0.622097
2000-01-05  0.722388 -0.072578  0.626614
2000-01-06 -0.899071  0.593542  1.557537
2000-01-07  0.446388  1.357955  0.979408
2000-01-08  1.434806 -1.811677 -0.556518

In [92]: type(df['A'])
Out[92]: pandas.core.series.Series

In [93]: df - df['A']
Out[93]: 
            2000-01-01 00:00:00  2000-01-02 00:00:00  2000-01-03 00:00:00  ...
```

```
↪   A   B   C
2000-01-01                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-02                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-03                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-04                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-05                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-06                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-07                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN
2000-01-08                  NaN                  NaN                  NaN  ...
↪ NaN NaN NaN

[8 rows x 11 columns]
```

Warning:

```
df - df['A']
```

is now deprecated and will be removed in a future release. The preferred way to replicate this behavior is

```
df.sub(df['A'], axis=0)
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [94]: df * 5 + 2
Out[94]:
                   A         B         C
2000-01-01  5.904738  4.935707  3.932165
2000-01-02 -3.751704  4.769892  7.237304
2000-01-03  2.108354 -6.604107  2.963318
2000-01-04  7.291318  4.535757 -1.110487
2000-01-05  5.611942  1.637111  5.133071
2000-01-06 -2.495355  4.967712  9.787685
2000-01-07  4.231938  8.789774  6.897039
2000-01-08  9.174028 -7.058386 -0.782592

In [95]: 1 / df
Out[95]:
                    A          B         C
2000-01-01   1.280496   1.703167  2.587770
2000-01-02  -0.869308   1.805125  0.954690
2000-01-03  46.144995  -0.581118  5.190396
2000-01-04   0.944944   1.971798 -1.607465
2000-01-05   1.384297 -13.778321  1.595878
2000-01-06  -1.112259   1.684799  0.642039
2000-01-07   2.240205   0.736402  1.021025
2000-01-08   0.696959  -0.551975 -1.796886
```

```
In [96]: df ** 4
Out[96]: 
                       A          B         C
2000-01-01  3.719525e-01   0.118842  0.022300
2000-01-02  1.751080e+00   0.094183  1.203791
2000-01-03  2.205472e-07   8.768861  0.001378
2000-01-04  1.254225e+00   0.066153  0.149773
2000-01-05  2.723222e-01   0.000028  0.154170
2000-01-06  6.533950e-01   0.124110  5.885094
2000-01-07  3.970532e-02   3.400487  0.920140
2000-01-08  4.238110e+00  10.772667  0.095922
```

Boolean operators work as well:

```
In [97]: df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1]}, dtype=bool)

In [98]: df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0]}, dtype=bool)

In [99]: df1 & df2
Out[99]: 
       a      b
0  False  False
1  False   True
2   True  False

In [100]: df1 | df2
Out[100]: 
      a     b
0  True  True
1  True  True
2  True  True

In [101]: df1 ^ df2
Out[101]: 
       a      b
0   True   True
1   True  False
2  False   True

In [102]: -df1
Out[102]: 
       a      b
0  False   True
1   True  False
2  False  False
```

Transposing

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

```
# only show the first 5 rows
In [103]: df[:5].T
Out[103]: 
```

(continues on next page)

(continued from previous page)

```
   2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A    0.780948   -1.150341    0.021671    1.058264    0.722388
B    0.587141    0.553978   -1.720821    0.507151   -0.072578
C    0.386433    1.047461    0.192664   -0.622097    0.626614
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt,) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```
In [104]: np.exp(df)
Out[104]: 
                   A         B         C
2000-01-01  2.183540  1.798839  1.471722
2000-01-02  0.316529  1.740162  2.850404
2000-01-03  1.021907  0.178919  1.212475
2000-01-04  2.881363  1.660554  0.536817
2000-01-05  2.059346  0.929993  1.871264
2000-01-06  0.406948  1.810390  4.747114
2000-01-07  1.562657  3.888233  2.662879
2000-01-08  4.198829  0.163380  0.573201

In [105]: np.asarray(df)
Out[105]: 
array([[ 0.78094758,  0.58714139,  0.38643309],
       [-1.15034079,  0.55397835,  1.04746083],
       [ 0.02167082, -1.7208214 ,  0.19266354],
       [ 1.0582636 ,  0.50715133, -0.62209742],
       [ 0.72238838, -0.07257778,  0.62661419],
       [-0.89907091,  0.59354246,  1.55753692],
       [ 0.44638764,  1.35795473,  0.97940772],
       [ 1.43480561, -1.81167717, -0.55651844]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

`Series` implements `__array_ufunc__`, which allows it to work with NumPys universal functions.

The ufunc is applied to the underlying array in a Series.

```
In [106]: ser = pd.Series([1, 2, 3, 4])

In [107]: np.exp(ser)
Out[107]: 
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

Changed in version 0.25.0: When multiple `Series` are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using `numpy.remainder()` on two `Series` with differently ordered labels will align before the operation.

```
In [108]: ser1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [109]: ser2 = pd.Series([1, 3, 5], index=['b', 'a', 'c'])

In [110]: ser1
Out[110]: 
a    1
b    2
c    3
dtype: int64

In [111]: ser2
Out[111]: 
b    1
a    3
c    5
dtype: int64

In [112]: np.remainder(ser1, ser2)
Out[112]: 
a    1
b    0
c    3
dtype: int64
```

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

```
In [113]: ser3 = pd.Series([2, 4, 6], index=['b', 'c', 'd'])

In [114]: ser3
Out[114]: 
b    2
c    4
d    6
dtype: int64

In [115]: np.remainder(ser1, ser3)
Out[115]: 
a    NaN
b    0.0
c    3.0
d    NaN
dtype: float64
```

When a binary ufunc is applied to a `Series` and `Index`, the Series implementation takes precedence and a Series is returned.

```
In [116]: ser = pd.Series([1, 2, 3])

In [117]: idx = pd.Index([4, 5, 6])

In [118]: np.maximum(ser, idx)
Out[118]: 
0    4
```

(continues on next page)

(continued from previous page)

```
1    5
2    6
dtype: int64
```

NumPy ufuncs are safe to apply to `Series` backed by non-ndarray arrays, for example `SparseArray` (see *Sparse calculation*). If possible, the ufunc is applied without converting the underlying data to an ndarray.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using *info()*. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [119]: baseball = pd.read_csv('data/baseball.csv')

In [120]: print(baseball)
       id     player  year  stint team  lg   g   ab   r    h  ...   rbi   sb ␣
↪ cs  bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006      2  CHN  NL  19   50   6   14  ...   2.0  1.0 ␣
↪1.0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006      1  BOS  AL  31    2   0    1  ...   0.0  0.0 ␣
↪0.0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..  ..  ...  ..  ...  ...   ...  ... ␣
↪...  ..   ...  ...  ...  ...  ...   ...
98  89533   aloumo01  2007      1  NYN  NL  87  328  51  112  ...  49.0  3.0 ␣
↪0.0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007      1  NYN  NL   8   22   1    3  ...   0.0  0.0 ␣
↪0.0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [121]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
id        100 non-null int64
player    100 non-null object
year      100 non-null int64
stint     100 non-null int64
team      100 non-null object
lg        100 non-null object
g         100 non-null int64
ab        100 non-null int64
r         100 non-null int64
h         100 non-null int64
X2b       100 non-null int64
X3b       100 non-null int64
hr        100 non-null int64
rbi       100 non-null float64
sb        100 non-null float64
cs        100 non-null float64
bb        100 non-null int64
so        100 non-null float64
ibb       100 non-null float64
```

```
hbp       100 non-null float64
sh        100 non-null float64
sf        100 non-null float64
gidp      100 non-null float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.1+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it wont always fit the console width:

```
In [122]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [123]: pd.DataFrame(np.random.randn(3, 12))
Out[123]:
         0         1         2         3         4   ...         7         8         9␣
↪        10        11
0 -1.533705 -0.176507  0.608559  0.978633 -1.593325  ...  0.648689  0.311338  1.
↪019847 -0.905475  0.979426
1  0.301441  1.043466 -0.585706 -0.300476 -0.631623  ...  1.778709  0.084805  0.
↪480836  0.003515  0.726600
2  0.261610 -0.446297  1.996565  0.344258  1.052674  ... -1.230991 -0.413491 -0.
↪145399 -0.301483  1.844836

[3 rows x 12 columns]
```

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

```
In [124]: pd.set_option('display.width', 40)  # default is 80

In [125]: pd.DataFrame(np.random.randn(3, 12))
Out[125]:
         0         1         2         3         4   ...         7         8         9␣
↪        10        11
0  0.821556 -0.392733  0.317696 -3.065966  0.216755  ...  0.883135 -1.962886 -1.
↪096080 -0.787760 -0.290642
```

(continues on next page)

(continued from previous page)

```
1 -0.127228  1.021397  0.061536  0.340173 -0.447509  ...  0.268224 -1.339587 -0.
↪476597  0.180935  0.841153
2  2.699198  0.291861 -0.295606 -1.319987 -0.114294  ...  1.565075 -0.317159 -1.
↪399831 -0.254755  0.225071

[3 rows x 12 columns]
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [126]: datafile = {'filename': ['filename_01', 'filename_02'],
   .....:             'path': ["media/user_name/storage/folder_01/filename_01",
   .....:                      "media/user_name/storage/folder_02/filename_02"]}
   .....:

In [127]: pd.set_option('display.max_colwidth', 30)

In [128]: pd.DataFrame(datafile)
Out[128]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [129]: pd.set_option('display.max_colwidth', 100)

In [130]: pd.DataFrame(datafile)
Out[130]:
      filename                                           path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

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

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [131]: df = pd.DataFrame({'foo1': np.random.randn(5),
   .....:                    'foo2': np.random.randn(5)})
   .....:

In [132]: df
Out[132]:
       foo1      foo2
0  0.486038 -0.803225
1 -0.223749 -1.166967
2  0.128104  0.344985
3 -0.367409 -1.789576
4 -0.684883 -1.035161

In [133]: df.foo1
Out[133]:
0    0.486038
1   -0.223749
2    0.128104
3   -0.367409
```

```
4   -0.684883
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>  # noqa: E225, E999
df.foo1  df.foo2
```

{{ header }}

3.5 Comparison with other tools

{{ header }}

3.5.1 Comparison with R / R libraries

Since `pandas` aims to provide a lot of the data manipulation and analysis functionality that people use R for, this page was started to provide a more detailed look at the R language and its many third party libraries as they relate to `pandas`. In comparisons with R and CRAN libraries, we care about the following things:

- **Functionality / flexibility**: what can/cannot be done with each tool
- **Performance**: how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use**: Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these R packages.

For transfer of `DataFrame` objects from `pandas` to R, one option is to use HDF5 files, see *External compatibility* for an example.

Quick reference

Well start off with a quick reference guide pairing some common R operations using dplyr with pandas equivalents.

Querying, filtering, sampling

R	pandas
`dim(df)`	`df.shape`
`head(df)`	`df.head()`
`slice(df, 1:10)`	`df.iloc[:9]`
`filter(df, col1 == 1, col2 == 1)`	`df.query('col1 == 1 & col2 == 1')`
`df[df$col1 == 1 & df$col2 == 1,]`	`df[(df.col1 == 1) & (df.col2 == 1)]`
`select(df, col1, col2)`	`df[['col1', 'col2']]`
`select(df, col1:col3)`	`df.loc[:, 'col1':'col3']`
`select(df, -(col1:col3))`	`df.drop(cols_to_drop, axis=1)` but see[1]
`distinct(select(df, col1))`	`df[['col1']].drop_duplicates()`
`distinct(select(df, col1, col2))`	`df[['col1', 'col2']].drop_duplicates()`
`sample_n(df, 10)`	`df.sample(n=10)`
`sample_frac(df, 0.01)`	`df.sample(frac=0.01)`

Sorting

R	pandas
`arrange(df, col1, col2)`	`df.sort_values(['col1', 'col2'])`
`arrange(df, desc(col1))`	`df.sort_values('col1', ascending=False)`

Transforming

R	pandas
`select(df, col_one = col1)`	`df.rename(columns={'col1':  'col_one'})['col_one']`
`rename(df, col_one = col1)`	`df.rename(columns={'col1':  'col_one'})`
`mutate(df, c=a-b)`	`df.assign(c=df.a-df.b)`

Grouping and summarizing

R	pandas
`summary(df)`	`df.describe()`
`gdf <- group_by(df, col1)`	`gdf = df.groupby('col1')`
`summarise(gdf, avg=mean(col1, na.rm=TRUE))`	`df.groupby('col1').agg({'col1': 'mean'})`
`summarise(gdf, total=sum(col1))`	`df.groupby('col1').sum()`

Base R

Slicing with Rs `c`

R makes it easy to access `data.frame` columns by name

```
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in `pandas` is straightforward

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list('abc'))

In [2]: df[['a', 'c']]
Out[2]:
          a         c
0  0.207136 -0.469146
```

[1] Rs shorthand for a subrange of columns (`select(df, col1:col3)`) can be approached cleanly in pandas, if you have the list of columns, for example `df[cols[1:3]]` or `df.drop(cols[1:3])`, but doing this by column name is a bit messy.

```
1  1.385538 -0.362643
2 -0.571230  1.414622
3 -0.860474  0.448659
4  0.198087 -0.138916
5 -0.236168  0.368541
6  0.077176  1.707149
7  1.212901  0.166563
8  1.116482  1.600544
9 -1.464652 -0.306999

In [3]: df.loc[:, ['a', 'c']]
Out[3]:
          a         c
0  0.207136 -0.469146
1  1.385538 -0.362643
2 -0.571230  1.414622
3 -0.860474  0.448659
4  0.198087 -0.138916
5 -0.236168  0.368541
6  0.077176  1.707149
7  1.212901  0.166563
8  1.116482  1.600544
9 -1.464652 -0.306999
```

Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list('abcdefg')

In [5]: n = 30

In [6]: columns = named + np.arange(len(named), n).tolist()

In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)

In [8]: df.iloc[:, np.r_[:10, 24:30]]
Out[8]:
          a         b         c         d         e  ...        25        26        ␣
↪27        28        29
0 -0.157020 -0.312100  2.101265  0.715737  0.161958  ... -0.592069  0.892628 -1.
↪426348  0.859143 -1.247781
1 -0.648453  0.427706 -1.167462 -0.693429  1.902080  ...  1.237770  0.661616  1.
↪144256  0.447000  0.198894
2 -0.574805 -1.204465 -0.201374  0.278877 -0.286763  ... -0.106527 -0.456767  0.
↪706188  0.133951  0.792075
3 -0.938013 -0.552114 -0.266028 -0.838281 -0.138009  ... -0.120739 -0.195936  0.
↪089482 -0.052919 -0.123138
4  0.348967 -1.399891 -1.166618  0.608277 -1.297820  ...  0.879326  1.234826  1.
↪130073 -0.820004 -1.120951
5  0.003281 -0.998084  1.300842  1.437468 -0.261703  ... -1.455117  0.004933 -1.
↪507294 -1.165402 -1.945610
6 -1.951386  1.894199  1.466576  0.731260 -0.068252  ...  1.449550  0.197532  0.
↪139688 -1.246171 -0.690641
7  2.119767  0.461675  1.116264  0.099048  1.471643  ... -0.444326 -1.369138  2.
↪162595  1.986876 -0.458651
8 -0.471204 -2.008981  0.609188  1.187271  0.178697  ...  1.538607 -0.038627 -0.
↪442633 -0.577955 -0.161513
```

(continues on next page)

(continued from previous page)

```
9  -1.080629 -1.573208  0.240054 -1.182698  1.380183  ... -0.158802  0.767407 -0.
↪541999  0.275594 -0.193256
10  1.212011  0.021855  0.787764 -0.111502  0.978365  ...  1.190005 -0.341142  0.
↪019357 -0.954364 -0.667115
11 -0.372254 -1.419194  1.284916 -2.567632 -0.394702  ... -0.076347  0.540433 -0.
↪272025 -0.397703  0.119171
12  0.782784 -0.120299  1.073066 -0.462378  0.709914  ... -2.521942  0.150246 -0.
↪365755 -0.004324  0.019941
13 -1.955448 -0.152551 -1.394527 -1.838862  0.116834  ...  0.678305 -0.439020 -0.
↪004064  0.211563  0.366738
14 -1.345105 -0.939819 -0.126120 -0.325684 -1.346863  ... -0.004781  1.457464 -0.
↪859323 -0.615783  0.958581
15  1.661150  0.229221 -0.207492  1.324775  0.095197  ...  0.337167 -0.536268 -1.
↪166466 -0.873786  0.970503
16 -1.721043  0.234624  1.131312  0.187112 -0.916265  ... -0.666787 -0.460773  0.
↪189052 -0.540372 -0.873749
17  0.530914  0.092203  0.004254 -1.445988 -1.683797  ... -0.338648 -1.736859  0.
↪924184 -0.086912 -1.122609
18  1.272979  0.039344 -0.723751 -0.926619  0.982875  ...  0.369011  0.601631  1.
↪047098  0.168620  1.486580
19  1.019541  1.022316 -0.989192  1.331725 -1.333733  ...  0.259385 -0.265833  0.
↪927853  1.725549  0.617201
20  0.211400  0.788539  0.191757  1.249484 -1.506977  ... -2.191752  0.028034 -0.
↪118943  1.828197 -1.588506
21  0.435479 -1.408082 -0.518882  0.547208  0.792259  ...  0.315643  1.377493  0.
↪694965 -0.107745 -0.824049
22 -0.265984  1.632759 -1.072779  1.101620  0.169399  ...  1.635833 -0.175988  0.
↪993764 -0.049508 -0.738073
23 -1.593320  0.940353 -0.641346  1.229798 -1.588487  ...  1.768347 -1.377505 -0.
↪013970  0.495098  0.928113
24 -1.044262  0.476848 -0.559825  0.128314  1.010380  ... -0.368830 -1.917762  0.
↪879227  2.214419 -1.710267
25 -0.656052  0.911241 -0.698254 -0.304604 -0.144525  ...  0.188098 -0.079259 -0.
↪655610 -0.699509  1.270956
26  1.079778  0.928320 -0.079028  0.163994  1.242867  ... -0.032398 -0.614531  0.
↪159650  1.161948 -0.289595
27  0.104740 -0.179138 -1.081585  0.366706  0.072617  ... -0.014595  0.042685 -0.
↪211296  1.429080 -0.246837
28 -0.683389  1.678525 -0.869742 -0.459486 -4.081324  ... -0.663183  0.054473  1.
↪520833 -2.107380 -0.497143
29 -1.643519  0.094721  1.377571 -1.433719  0.744651  ...  0.611787  0.655887  0.
↪327674  0.144085 -1.006701

[30 rows x 16 columns]
```

aggregate

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called df and splitting it into groups by1 and by2:

```
df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
```

(continues on next page)

(continued from previous page)

```
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)
```

The *groupby()* method is similar to base R aggregate function.

```
In [9]: df = pd.DataFrame(
   ...:     {'v1': [1, 3, 5, 7, 8, 3, 5, np.nan, 4, 5, 7, 9],
   ...:      'v2': [11, 33, 55, 77, 88, 33, 55, np.nan, 44, 55, 77, 99],
   ...:      'by1': ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],
   ...:      'by2': ["wet", "dry", 99, 95, np.nan, "damp", 95, 99, "red", 99, np.nan,
   ...:              np.nan]})
   ...: 

In [10]: g = df.groupby(['by1', 'by2'])

In [11]: g[['v1', 'v2']].mean()
Out[11]: 
            v1    v2
by1  by2            
1    95    5.0  55.0
     99    5.0  55.0
2    95    7.0  77.0
     99    NaN   NaN
big  damp  3.0  33.0
blue dry   3.0  33.0
red  red   4.0  44.0
     wet   1.0  11.0
```

For more details and examples see *the groupby documentation*.

match / %in%

A common way to select data in R is using %in% which is defined using the function match. The operator %in% is used to return a logical vector indicating if there is a match or not:

```
s <- 0:4
s %in% c(2,4)
```

The *isin()* method is similar to R %in% operator:

```
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)

In [13]: s.isin([2, 4])
Out[13]: 
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

The match function returns a vector of the positions of matches of its first argument in its second:

```
s <- 0:4
match(s, c(2,4))
```

For more details and examples see *the reshaping documentation*.

tapply

tapply is similar to aggregate, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called baseball, and retrieving information based on the array team:

```
baseball <-
  data.frame(team = gl(5, 5,
             labels = paste("Team", LETTERS[1:5])),
             player = sample(letters, 25),
             batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball.example$team,
       max)
```

In pandas we may use *pivot_table()* method to handle this:

```
In [14]: import random

In [15]: import string

In [16]: baseball = pd.DataFrame(
   ....:     {'team': ["team %d" % (x + 1) for x in range(5)] * 5,
   ....:      'player': random.sample(list(string.ascii_lowercase), 25),
   ....:      'batting avg': np.random.uniform(.200, .400, 25)})
   ....:

In [17]: baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)
Out[17]:
team           team 1    team 2    team 3    team 4    team 5
batting avg  0.287166  0.392928  0.372269  0.398628  0.393112
```

For more details and examples see *the reshaping documentation*.

subset

The *query()* method is similar to the base R subset function. In R you might want to get the rows of a data.frame where one columns values are less than another columns values:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,]  # note the comma
```

In pandas, there are a few ways to perform subsetting. You can use *query()* or pass an expression as if it were an index/slice as well as standard boolean indexing:

```
In [18]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.
↪randn(10)})

In [19]: df.query('a <= b')
Out[19]:
          a         b
0 -1.829965  0.318380
6  0.533221  0.686366
9  0.290722  0.761557

In [20]: df[df.a <= df.b]
```

```
Out[20]:
          a         b
0 -1.829965  0.318380
6  0.533221  0.686366
9  0.290722  0.761557

In [21]: df.loc[df.a <= df.b]
Out[21]:
          a         b
0 -1.829965  0.318380
6  0.533221  0.686366
9  0.290722  0.761557
```

For more details and examples see *the query documentation*.

with

An expression using a data.frame called `df` in R with the columns `a` and `b` would be evaluated using `with` like so:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b  # same as the previous expression
```

In `pandas` the equivalent expression, using the `eval()` method, would be:

```
In [22]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.
↪randn(10)})

In [23]: df.eval('a + b')
Out[23]:
0    1.134063
1   -1.663533
2    2.394069
3    2.476911
4    1.059003
5    1.941878
6   -0.499256
7    1.955427
8    1.029021
9   -1.430095
dtype: float64

In [24]: df.a + df.b  # same as the previous expression
Out[24]:
0    1.134063
1   -1.663533
2    2.394069
3    2.476911
4    1.059003
5    1.941878
6   -0.499256
7    1.955427
8    1.029021
9   -1.430095
```

```
dtype: float64
```

In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see *the eval documentation*.

plyr

`plyr` is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, `a` for `arrays`, `l` for `lists`, and `d` for `data.frame`. The table below shows how these data structures could be mapped in Python.

R	Python
array	list
lists	dictionary or list of objects
data.frame	dataframe

ddply

An expression using a data.frame called `df` in R where you want to summarize `x` by `month`:

```
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
      mean = round(mean(x), 2),
      sd = round(sd(x), 2))
```

In `pandas` the equivalent expression, using the *groupby()* method, would be:

```
In [25]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 120),
   ....:                    'y': np.random.uniform(7., 334., 120),
   ....:                    'z': np.random.uniform(1.7, 20.7, 120),
   ....:                    'month': [5, 6, 7, 8] * 30,
   ....:                    'week': np.random.randint(1, 4, 120)})
   ....:

In [26]: grouped = df.groupby(['month', 'week'])

In [27]: grouped['x'].agg([np.mean, np.std])
Out[27]:
                 mean        std
month week
5     1     54.610775  24.673729
      2     79.400383  54.221977
      3     87.966804  55.426962
6     1     75.458265  40.219732
      2     63.014659  30.576905
      3     82.579357  56.435221
```

(continues on next page)

(continued from previous page)

```
7    1    66.587862  40.926231
     2    92.141920  45.262135
     3    81.048190  48.547999
8    1    89.249868  50.984085
     2    84.233718  51.690704
     3    96.823561  43.465622
```

For more details and examples see *the groupby documentation*.

reshape / reshape2

melt.array

An expression using a 3 dimensional array called a in R where you want to melt it into a data.frame:

```
a <- array(c(1:23, NA), c(2,3,4))
data.frame(melt(a))
```

In Python, since a is a list, you can simply use list comprehension.

```
In [28]: a = np.array(list(range(1, 24)) + [np.NAN]).reshape(2, 3, 4)

In [29]: pd.DataFrame([tuple(list(x) + [val]) for x, val in np.ndenumerate(a)])
Out[29]: 
    0  1  2     3
0   0  0  0   1.0
1   0  0  1   2.0
2   0  0  2   3.0
3   0  0  3   4.0
4   0  1  0   5.0
5   0  1  1   6.0
6   0  1  2   7.0
7   0  1  3   8.0
8   0  2  0   9.0
9   0  2  1  10.0
10  0  2  2  11.0
11  0  2  3  12.0
12  1  0  0  13.0
13  1  0  1  14.0
14  1  0  2  15.0
15  1  0  3  16.0
16  1  1  0  17.0
17  1  1  1  18.0
18  1  1  2  19.0
19  1  1  3  20.0
20  1  2  0  21.0
21  1  2  1  22.0
22  1  2  2  23.0
23  1  2  3   NaN
```

melt.list

An expression using a list called a in R where you want to melt it into a data.frame:

```
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.

```
In [30]: a = list(enumerate(list(range(1, 5)) + [np.NAN]))

In [31]: pd.DataFrame(a)
Out[31]:
   0    1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  NaN
```

For more details and examples see *the Into to Data Structures documentation*.

melt.data.frame

An expression using a data.frame called `cheese` in R where you want to reshape the data.frame:

```
cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
)
melt(cheese, id=c("first", "last"))
```

In Python, the `melt()` method is the R equivalent:

```
In [32]: cheese = pd.DataFrame({'first': ['John', 'Mary'],
   ....:                        'last': ['Doe', 'Bo'],
   ....:                        'height': [5.5, 6.0],
   ....:                        'weight': [130, 150]})
   ....:

In [33]: pd.melt(cheese, id_vars=['first', 'last'])
Out[33]:
  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 [34]: cheese.set_index(['first', 'last']).stack()  # alternative way
Out[34]:
first  last
John   Doe   height      5.5
             weight    130.0
Mary   Bo    height      6.0
             weight    150.0
dtype: float64
```

For more details and examples see *the reshaping documentation*.

cast

In R `acast` is an expression using a data.frame called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7),4),
  week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python the best way is to make use of `pivot_table()`:

```
In [35]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 12),
   ....:                    'y': np.random.uniform(7., 334., 12),
   ....:                    'z': np.random.uniform(1.7, 20.7, 12),
   ....:                    'month': [5, 6, 7] * 4,
   ....:                    'week': [1, 2] * 6})
   ....: 

In [36]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [37]: pd.pivot_table(mdf, values='value', index=['variable', 'week'],
   ....:                columns=['month'], aggfunc=np.mean)
   ....: 
Out[37]: 
month                  5           6           7
variable week                                   
x        1     114.080088   85.737423  106.491489
         2      51.936679   42.642634  112.288362
y        1     119.753362  196.773182  177.983139
         2     289.007313  183.868930  217.038570
z        1       7.881457    5.557442   10.480275
         2      10.927325   13.645086    7.487791
```

Similarly for `dcast` which uses a data.frame called `df` in R to aggregate information based on `Animal` and `FeedType`:

```
df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
             'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))
```

Python can approach this in two different ways. Firstly, similar to above using `pivot_table()`:

```
In [38]: df = pd.DataFrame({
   ....:     'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
   ....:                'Animal2', 'Animal3'],
   ....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
   ....:     'Amount': [10, 7, 4, 2, 5, 6, 2],
   ....: })
   ....:

In [39]: df.pivot_table(values='Amount', index='Animal', columns='FeedType',
   ....:                aggfunc='sum')
   ....:
Out[39]:
FeedType      A     B
Animal
Animal1    10.0   5.0
Animal2     2.0  13.0
Animal3     6.0   NaN
```

The second approach is to use the `groupby()` method:

```
In [40]: df.groupby(['Animal', 'FeedType'])['Amount'].sum()
Out[40]:
Animal   FeedType
Animal1  A           10
         B            5
Animal2  A            2
         B           13
Animal3  A            6
Name: Amount, dtype: int64
```

For more details and examples see *the reshaping documentation* or *the groupby documentation*.

factor

pandas has a data type for categorical data.

```
cut(c(1,2,3,4,5,6), 3)
factor(c(1,2,3,2,2,3))
```

In pandas this is accomplished with `pd.cut` and `astype("category")`:

```
In [41]: pd.cut(pd.Series([1, 2, 3, 4, 5, 6]), 3)
Out[41]:
0    (0.995, 2.667]
1    (0.995, 2.667]
2    (2.667, 4.333]
3    (2.667, 4.333]
4      (4.333, 6.0]
5      (4.333, 6.0]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333,␣
↪6.0]]

In [42]: pd.Series([1, 2, 3, 2, 2, 3]).astype("category")
Out[42]:
0    1
```

```
1    2
2    3
3    2
4    2
5    3
dtype: category
Categories (3, int64): [1, 2, 3]
```

For more details and examples see *categorical introduction* and the *API documentation*. There is also a documentation regarding the *differences to Rs factor*. {{ header }}

3.5.2 Comparison with SQL

Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If youre new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

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

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

Most of the examples will utilize the `tips` dataset found within pandas tests. Well read the data into a DataFrame called *tips* and assume we have a database table of the same name and structure.

```
In [3]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/data/tips.csv')
   ...:

In [4]: tips = pd.read_csv(url)

In [5]: tips.head()
Out[5]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

SELECT

In SQL, selection is done using a comma-separated list of columns youd like to select (or a `*` to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
Out[6]:
```

(continues on next page)

(continued from previous page)

```
   total_bill   tip smoker    time
0       16.99  1.01     No  Dinner
1       10.34  1.66     No  Dinner
2       21.01  3.50     No  Dinner
3       23.68  3.31     No  Dinner
4       24.59  3.61     No  Dinner
```

Calling the DataFrame without the list of column names would display all columns (akin to SQLs *).

WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```
In [7]: tips[tips['time'] == 'Dinner'].head(5)
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The above statement is simply passing a `Series` of True/False objects to the DataFrame, returning all rows with True.

```
In [8]: is_dinner = tips['time'] == 'Dinner'

In [9]: is_dinner.value_counts()
Out[9]:
True     176
False     68
Name: time, dtype: int64

In [10]: tips[is_dinner].head(5)
Out[10]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Just like SQLs OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

```
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```

```
# tips of more than $5.00 at Dinner meals
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
Out[11]: 
     total_bill    tip     sex smoker  day    time  size
23        39.42   7.58    Male     No  Sat  Dinner     4
44        30.40   5.60    Male     No  Sun  Dinner     4
47        32.40   6.00    Male     No  Sun  Dinner     4
52        34.81   5.20  Female     No  Sun  Dinner     4
59        48.27   6.73    Male     No  Sat  Dinner     4
116       29.93   5.07    Male     No  Sun  Dinner     4
155       29.85   5.14  Female     No  Sun  Dinner     5
170       50.81  10.00    Male    Yes  Sat  Dinner     3
172        7.25   5.15    Male    Yes  Sun  Dinner     2
181       23.33   5.65    Male    Yes  Sun  Dinner     2
183       23.17   6.50    Male    Yes  Sun  Dinner     4
211       25.89   5.16    Male    Yes  Sat  Dinner     4
212       48.33   9.00    Male     No  Sat  Dinner     4
214       28.17   6.50  Female    Yes  Sat  Dinner     3
239       29.03   5.92    Male     No  Sat  Dinner     3
```

```
-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

```
# tips by parties of at least 5 diners OR bill total was more than $45
In [12]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
Out[12]: 
     total_bill    tip     sex smoker   day    time  size
59        48.27   6.73    Male     No   Sat  Dinner     4
125       29.80   4.20  Female     No  Thur   Lunch     6
141       34.30   6.70    Male     No  Thur   Lunch     6
142       41.19   5.00    Male     No  Thur   Lunch     5
143       27.05   5.00  Female     No  Thur   Lunch     6
155       29.85   5.14  Female     No   Sun  Dinner     5
156       48.17   5.00    Male     No   Sun  Dinner     6
170       50.81  10.00    Male    Yes   Sat  Dinner     3
182       45.35   3.50    Male    Yes   Sun  Dinner     3
185       20.69   5.00    Male     No   Sun  Dinner     5
187       30.46   2.00    Male    Yes   Sun  Dinner     5
212       48.33   9.00    Male     No   Sat  Dinner     4
216       28.15   3.00    Male    Yes   Sat  Dinner     5
```

NULL checking is done using the *notna()* and *isna()* methods.

```
In [13]: frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],
   ....:                       'col2': ['F', np.NaN, 'G', 'H', 'I']})
   ....: 

In [14]: frame
Out[14]: 
  col1 col2
0    A    F
1    B  NaN
2  NaN    G
3    C    H
4    D    I
```

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```
In [15]: frame[frame['col2'].isna()]
Out[15]:
  col1 col2
1    B  NaN
```

Getting items where `col1` IS NOT NULL can be done with *notna()*.

```
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```
In [16]: frame[frame['col1'].notna()]
Out[16]:
  col1 col2
0    A    F
1    B  NaN
3    C    H
4    D    I
```

GROUP BY

In pandas, SQLs GROUP BY operations are performed using the similarly named *groupby()* method. *groupby()* typically refers to a process where wed like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female     87
Male      157
*/
```

The pandas equivalent would be:

```
In [17]: tips.groupby('sex').size()
Out[17]:
sex
Female     87
Male      157
dtype: int64
```

Notice that in the pandas code we used *size()* and not *count()*. This is because *count()* applies the function to each column, returning the number of `not null` records within each.

```
In [18]: tips.groupby('sex').count()
Out[18]: 
        total_bill  tip  smoker  day  time  size
sex                                             
Female          87   87      87   87    87    87
Male           157  157     157  157   157   157
```

Alternatively, we could have applied the *count()* method to an individual column:

```
In [19]: tips.groupby('sex')['total_bill'].count()
Out[19]: 
sex
Female     87
Male      157
Name: total_bill, dtype: int64
```

Multiple functions can also be applied at once. For instance, say wed like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

```
SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
/*
Fri   2.734737   19
Sat   2.993103   87
Sun   3.255132   76
Thur  2.771452   62
*/
```

```
In [20]: tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
Out[20]: 
           tip  day
day                
Fri   2.734737   19
Sat   2.993103   87
Sun   3.255132   76
Thur  2.771452   62
```

Grouping by more than one column is done by passing a list of columns to the *groupby()* method.

```
SELECT smoker, day, COUNT(*), AVG(tip)
FROM tips
GROUP BY smoker, day;
/*
smoker day
No     Fri      4  2.812500
       Sat     45  3.102889
       Sun     57  3.167895
       Thur    45  2.673778
Yes    Fri     15  2.714000
       Sat     42  2.875476
       Sun     19  3.516842
       Thur    17  3.030000
*/
```

```
In [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})
Out[21]:
              tip
             size      mean
smoker day
No     Fri    4.0  2.812500
       Sat   45.0  3.102889
       Sun   57.0  3.167895
       Thur  45.0  2.673778
Yes    Fri   15.0  2.714000
       Sat   42.0  2.875476
       Sun   19.0  3.516842
       Thur  17.0  3.030000
```

JOIN

JOINs can be performed with `join()` or `merge()`. By default, `join()` will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

```
In [22]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [23]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....:
```

Assume we have two database tables of the same name and structure as our DataFrames.

Now lets go over the various types of JOINs.

INNER JOIN

```
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
```

```
# merge performs an INNER JOIN by default
In [24]: pd.merge(df1, df2, on='key')
Out[24]:
  key   value_x   value_y
0   B -1.152748 -0.878607
1   D -0.808546 -0.106710
2   D -0.808546  0.533295
```

`merge()` also offers parameters for cases when youd like to join one DataFrames column with another DataFrames index.

```
In [25]: indexed_df2 = df2.set_index('key')

In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)
Out[26]:
```

(continues on next page)

(continued from previous page)

```
  key   value_x   value_y
1   B -1.152748 -0.878607
3   D -0.808546 -0.106710
3   D -0.808546  0.533295
```

LEFT OUTER JOIN

```
-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df1
In [27]: pd.merge(df1, df2, on='key', how='left')
Out[27]: 
  key   value_x   value_y
0   A -1.421715       NaN
1   B -1.152748 -0.878607
2   C -0.129893       NaN
3   D -0.808546 -0.106710
4   D -0.808546  0.533295
```

RIGHT JOIN

```
-- show all records from df2
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df2
In [28]: pd.merge(df1, df2, on='key', how='right')
Out[28]: 
  key   value_x   value_y
0   B -1.152748 -0.878607
1   D -0.808546 -0.106710
2   D -0.808546  0.533295
3   E       NaN -1.408811
```

FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

```
-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from both frames
In [29]: pd.merge(df1, df2, on='key', how='outer')
Out[29]: 
  key   value_x   value_y
0   A -1.421715       NaN
1   B -1.152748 -0.878607
2   C -0.129893       NaN
3   D -0.808546 -0.106710
4   D -0.808546  0.533295
5   E       NaN -1.408811
```

UNION

UNION ALL can be performed using `concat()`.

```
In [30]: df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],
   ....:                     'rank': range(1, 4)})
   ....: 

In [31]: df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],
   ....:                     'rank': [1, 4, 5]})
   ....: 
```

```
SELECT city, rank
FROM df1
UNION ALL
SELECT city, rank
FROM df2;
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
      Chicago     1
       Boston     4
  Los Angeles     5
*/
```

```
In [32]: pd.concat([df1, df2])
Out[32]: 
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
0        Chicago     1
1         Boston     4
2    Los Angeles     5
```

SQLs UNION is similar to UNION ALL, however UNION will remove duplicate rows.

```
SELECT city, rank
FROM df1
UNION
SELECT city, rank
FROM df2;
```

(continues on next page)

(continued from previous page)

```
-- notice that there is only one Chicago record this time
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
       Boston     4
  Los Angeles     5
*/
```

In pandas, you can use `concat()` in conjunction with `drop_duplicates()`.

```
In [33]: pd.concat([df1, df2]).drop_duplicates()
Out[33]:
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
1         Boston     4
2    Los Angeles     5
```

Pandas equivalents for some SQL analytic and aggregate functions

Top N rows with offset

```
-- MySQL
SELECT * FROM tips
ORDER BY tip DESC
LIMIT 10 OFFSET 5;
```

```
In [34]: tips.nlargest(10 + 5, columns='tip').tail(10)
Out[34]:
     total_bill   tip     sex smoker   day    time  size
183       23.17  6.50    Male    Yes   Sun  Dinner     4
214       28.17  6.50  Female    Yes   Sat  Dinner     3
47        32.40  6.00    Male     No   Sun  Dinner     4
239       29.03  5.92    Male     No   Sat  Dinner     3
88        24.71  5.85    Male     No  Thur   Lunch     2
181       23.33  5.65    Male    Yes   Sun  Dinner     2
44        30.40  5.60    Male     No   Sun  Dinner     4
52        34.81  5.20  Female     No   Sun  Dinner     4
85        34.83  5.17  Female     No  Thur   Lunch     4
211       25.89  5.16    Male    Yes   Sat  Dinner     4
```

Top N rows per group

```
-- Oracle's ROW_NUMBER() analytic function
SELECT * FROM (
  SELECT
    t.*,
    ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn
```

(continues on next page)

(continued from previous page)

```
  FROM tips t
)
WHERE rn < 3
ORDER BY day, rn;
```

```
In [35]: (tips.assign(rn=tips.sort_values(['total_bill'], ascending=False)
   ....:                     .groupby(['day'])
   ....:                     .cumcount() + 1)
   ....:      .query('rn < 3')
   ....:      .sort_values(['day', 'rn']))
   ....: 
Out[35]: 
     total_bill    tip     sex smoker   day    time  size  rn
95        40.17   4.73    Male    Yes   Fri  Dinner     4   1
90        28.97   3.00    Male    Yes   Fri  Dinner     2   2
170       50.81  10.00    Male    Yes   Sat  Dinner     3   1
212       48.33   9.00    Male     No   Sat  Dinner     4   2
156       48.17   5.00    Male     No   Sun  Dinner     6   1
182       45.35   3.50    Male    Yes   Sun  Dinner     3   2
197       43.11   5.00  Female    Yes  Thur   Lunch     4   1
142       41.19   5.00    Male     No  Thur   Lunch     5   2
```

the same using *rank(method=first)* function

```
In [36]: (tips.assign(rnk=tips.groupby(['day'])['total_bill']
   ....:                      .rank(method='first', ascending=False))
   ....:      .query('rnk < 3')
   ....:      .sort_values(['day', 'rnk']))
   ....: 
Out[36]: 
     total_bill    tip     sex smoker   day    time  size  rnk
95        40.17   4.73    Male    Yes   Fri  Dinner     4  1.0
90        28.97   3.00    Male    Yes   Fri  Dinner     2  2.0
170       50.81  10.00    Male    Yes   Sat  Dinner     3  1.0
212       48.33   9.00    Male     No   Sat  Dinner     4  2.0
156       48.17   5.00    Male     No   Sun  Dinner     6  1.0
182       45.35   3.50    Male    Yes   Sun  Dinner     3  2.0
197       43.11   5.00  Female    Yes  Thur   Lunch     4  1.0
142       41.19   5.00    Male     No  Thur   Lunch     5  2.0
```

```
-- Oracle's RANK() analytic function
SELECT * FROM (
  SELECT
    t.*,
    RANK() OVER(PARTITION BY sex ORDER BY tip) AS rnk
  FROM tips t
  WHERE tip < 2
)
WHERE rnk < 3
ORDER BY sex, rnk;
```

Lets find tips with (rank < 3) per gender group for (tips < 2). Notice that when using `rank(method='min')` function *rnk_min* remains the same for the same *tip* (as Oracles RANK() function)

```
In [37]: (tips[tips['tip'] < 2]
   ....:     .assign(rnk_min=tips.groupby(['sex'])['tip']
```

(continues on next page)

(continued from previous page)

```
   ....:                             .rank(method='min'))
   ....:      .query('rnk_min < 3')
   ....:      .sort_values(['sex', 'rnk_min']))
   ....:
Out[37]:
     total_bill   tip     sex smoker  day    time  size  rnk_min
67         3.07  1.00  Female    Yes  Sat  Dinner     1      1.0
92         5.75  1.00  Female    Yes  Fri  Dinner     2      1.0
111        7.25  1.00  Female     No  Sat  Dinner     1      1.0
236       12.60  1.00    Male    Yes  Sat  Dinner     2      1.0
237       32.83  1.17    Male    Yes  Sat  Dinner     2      2.0
```

UPDATE

```
UPDATE tips
SET tip = tip*2
WHERE tip < 2;
```

```
In [38]: tips.loc[tips['tip'] < 2, 'tip'] *= 2
```

DELETE

```
DELETE FROM tips
WHERE tip > 9;
```

In pandas we select the rows that should remain, instead of deleting them

```
In [39]: tips = tips.loc[tips['tip'] <= 9]
```

{{ header }}

3.5.3 Comparison with SAS

For potential users coming from SAS this page is meant to demonstrate how different SAS operations would be performed in pandas.

If youre new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

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

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

Note: Throughout this tutorial, the pandas `DataFrame` will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the `DataFrame`. This is often used in interactive work (e.g. Jupyter notebook or terminal) - the equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

Data structures

General terminology translation

pandas	SAS
`DataFrame`	data set
column	variable
row	observation
groupby	BY-group
`NaN`	`.`

`DataFrame` / `Series`

A `DataFrame` in pandas is analogous to a SAS data set - a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set using SASs `DATA` step, can also be accomplished in pandas.

A `Series` is the data structure that represents one column of a `DataFrame`. SAS doesnt have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column in the `DATA` step.

`Index`

Every `DataFrame` and `Series` has an `Index` - which are labels on the *rows* of the data. SAS does not have an exactly analogous concept. A data sets rows are essentially unlabeled, other than an implicit integer index that can be accessed during the `DATA` step (_N_).

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled `Index` or `MultiIndex` can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the *indexing documentation* for much more on how to use an `Index` effectively.

Data input / output

Constructing a DataFrame from values

A SAS data set can be built from specified values by placing the data after a `datalines` statement and specifying the column names.

```
data df;
    input x y;
    datalines;
    1 2
    3 4
    5 6
    ;
run;
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

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

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Like SAS, pandas provides utilities for reading in data from many formats. The tips dataset, found within the pandas tests (csv) will be used in many of the following examples.

SAS provides PROC IMPORT to read csv data into a data set.

```
proc import datafile='tips.csv' dbms=csv out=tips replace;
    getnames=yes;
run;
```

The pandas method is read_csv(), which works similarly.

```
In [5]: url = ('https://raw.github.com/pandas-dev/'
   ...:        'pandas/master/pandas/tests/data/tips.csv')
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Like PROC IMPORT, read_csv can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

In addition to text/csv, pandas supports a variety of other data formats such as Excel, HDF5, and SQL databases. These are all read via a pd.read_* function. See the *IO documentation* for more details.

Exporting data

The inverse of PROC IMPORT in SAS is PROC EXPORT

```
proc export data=tips outfile='tips2.csv' dbms=csv;
run;
```

Similarly in pandas, the opposite of `read_csv` is `to_csv()`, and other data formats follow a similar api.

```
tips.to_csv('tips2.csv')
```

Data operations

Operations on columns

In the `DATA` step, arbitrary math expressions can be used on new or existing columns.

```
data tips;
    set tips;
    total_bill = total_bill - 2;
    new_bill = total_bill / 2;
run;
```

pandas provides similar vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2.0

In [10]: tips.head()
Out[10]: 
   total_bill   tip     sex smoker  day    time  size  new_bill
0       14.99  1.01  Female     No  Sun  Dinner     2     7.495
1        8.34  1.66    Male     No  Sun  Dinner     3     4.170
2       19.01  3.50    Male     No  Sun  Dinner     3     9.505
3       21.68  3.31    Male     No  Sun  Dinner     2    10.840
4       22.59  3.61  Female     No  Sun  Dinner     4    11.295
```

Filtering

Filtering in SAS is done with an `if` or `where` statement, on one or more columns.

```
data tips;
    set tips;
    if total_bill > 10;
run;

data tips;
    set tips;
    where total_bill > 10;
    /* equivalent in this case - where happens before the
       DATA step begins and can also be used in PROC statements */
run;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*

```
In [11]: tips[tips['total_bill'] > 10].head()
Out[11]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/then logic

In SAS, if/then logic can be used to create new columns.

```
data tips;
    set tips;
    format bucket $4.;

    if total_bill < 10 then bucket = 'low';
    else bucket = 'high';
run;
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [12]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [13]: tips.head()
Out[13]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date functionality

SAS provides a variety of functions to do operations on date/datetime columns.

```
data tips;
    set tips;
    format date1 date2 date1_plusmonth mmddyy10.;
    date1 = mdy(1, 15, 2013);
    date2 = mdy(2, 15, 2015);
    date1_year = year(date1);
    date2_month = month(date2);
    * shift date to beginning of next interval;
    date1_next = intnx('MONTH', date1, 1);
    * count intervals between dates;
    months_between = intck('MONTH', date1, date2);
run;
```

The equivalent pandas operations are shown below. In addition to these functions pandas supports other Time Series features not available in Base SAS (such as resampling and custom offsets) - see the *timeseries documentation* for more details.

```
In [14]: tips['date1'] = pd.Timestamp('2013-01-15')

In [15]: tips['date2'] = pd.Timestamp('2015-02-15')

In [16]: tips['date1_year'] = tips['date1'].dt.year

In [17]: tips['date2_month'] = tips['date2'].dt.month

In [18]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [19]: tips['months_between'] = (
   ....:     tips['date2'].dt.to_period('M') - tips['date1'].dt.to_period('M'))
   ....:

In [20]: tips[['date1', 'date2', 'date1_year', 'date2_month',
   ....:       'date1_next', 'months_between']].head()
   ....:
Out[20]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of columns

SAS provides keywords in the `DATA` step to select, drop, and rename columns.

```
data tips;
    set tips;
    keep sex total_bill tip;
run;

data tips;
    set tips;
    drop sex;
run;

data tips;
    set tips;
    rename total_bill=total_bill_2;
run;
```

The same operations are expressed in pandas below.

```
# keep
In [21]: tips[['sex', 'total_bill', 'tip']].head()
Out[21]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61
```

```
# drop
In [22]: tips.drop('sex', axis=1).head()
Out[22]:
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [23]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
Out[23]:
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by values

Sorting in SAS is accomplished via `PROC SORT`

```
proc sort data=tips;
    by sex total_bill;
run;
```

pandas objects have a `sort_values()` method, which takes a list of columns to sort by.

```
In [24]: tips = tips.sort_values(['sex', 'total_bill'])

In [25]: tips.head()
Out[25]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String processing

Length

SAS determines the length of a character string with the LENGTHN and LENGTHC functions. `LENGTHN` excludes trailing blanks and `LENGTHC` includes trailing blanks.

```
data _null_;
set tips;
put(LENGTHN(time));
put(LENGTHC(time));
run;
```

Python determines the length of a character string with the `len` function. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [26]: tips['time'].str.len().head()
Out[26]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64

In [27]: tips['time'].str.rstrip().str.len().head()
Out[27]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64
```

Find

SAS determines the position of a character in a string with the FINDW function. `FINDW` takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
data _null_;
set tips;
put(FINDW(sex,'ale'));
run;
```

Python determines the position of a character in a string with the `find` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [28]: tips['sex'].str.find("ale").head()
Out[28]:
67     3
92     3
111    3
145    3
135    3
Name: sex, dtype: int64
```

Substring

SAS extracts a substring from a string based on its position with the SUBSTR function.

```
data _null_;
set tips;
put(substr(sex,1,1));
run;
```

With pandas you can use [] notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [29]: tips['sex'].str[0:1].head()
Out[29]: 
67     F
92     F
111    F
145    F
135    F
Name: sex, dtype: object
```

Scan

The SAS SCAN function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
data firstlast;
input String $60.;
First_Name = scan(string, 1);
Last_Name = scan(string, -1);
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [30]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [31]: firstlast['First_Name'] = firstlast['String'].str.split(" ", expand=True)[0]

In [32]: firstlast['Last_Name'] = firstlast['String'].str.rsplit(" ", expand=True)[0]

In [33]: firstlast
Out[33]: 
       String First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Upcase, lowcase, and propcase

The SAS UPCASE LOWCASE and PROPCASE functions change the case of the argument.

```
data firstlast;
input String $60.;
string_up = UPCASE(string);
string_low = LOWCASE(string);
string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
```

(continues on next page)

(continued from previous page)

```
;;;
run;
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [34]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [35]: firstlast['string_up'] = firstlast['String'].str.upper()

In [36]: firstlast['string_low'] = firstlast['String'].str.lower()

In [37]: firstlast['string_prop'] = firstlast['String'].str.title()

In [38]: firstlast
Out[38]: 
       String   string_up  string_low string_prop
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [39]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....: 

In [40]: df1
Out[40]: 
  key     value
0   A -1.579390
1   B  0.454513
2   C  0.051246
3   D  2.043664

In [41]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....: 

In [42]: df2
Out[42]: 
  key     value
0   B -0.016440
1   D -1.413628
2   D  0.117613
3   E -1.797024
```

In SAS, data must be explicitly sorted before merging. Different types of joins are accomplished using the `in=` dummy variables to track whether a match was found in one or both input frames.

```
proc sort data=df1;
    by key;
run;

proc sort data=df2;
```

(continues on next page)

(continued from previous page)

```
    by key;
run;

data left_join inner_join right_join outer_join;
    merge df1(in=a) df2(in=b);

    if a and b then output inner_join;
    if a then output left_join;
    if b then output right_join;
    if a or b then output outer_join;
run;
```

pandas DataFrames have a `merge()` method, which provides similar functionality. Note that the data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

```
In [43]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [44]: inner_join
Out[44]: 
  key   value_x   value_y
0   B  0.454513 -0.016440
1   D  2.043664 -1.413628
2   D  2.043664  0.117613

In [45]: left_join = df1.merge(df2, on=['key'], how='left')

In [46]: left_join
Out[46]: 
  key   value_x   value_y
0   A -1.579390       NaN
1   B  0.454513 -0.016440
2   C  0.051246       NaN
3   D  2.043664 -1.413628
4   D  2.043664  0.117613

In [47]: right_join = df1.merge(df2, on=['key'], how='right')

In [48]: right_join
Out[48]: 
  key   value_x   value_y
0   B  0.454513 -0.016440
1   D  2.043664 -1.413628
2   D  2.043664  0.117613
3   E       NaN -1.797024

In [49]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [50]: outer_join
Out[50]: 
  key   value_x   value_y
0   A -1.579390       NaN
1   B  0.454513 -0.016440
2   C  0.051246       NaN
3   D  2.043664 -1.413628
4   D  2.043664  0.117613
5   E       NaN -1.797024
```

Missing data

Like SAS, pandas has a representation for missing data - which is the special float value NaN (not a number). Many of the semantics are the same, for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [51]: outer_join
Out[51]:
  key   value_x   value_y
0   A -1.579390       NaN
1   B  0.454513 -0.016440
2   C  0.051246       NaN
3   D  2.043664 -1.413628
4   D  2.043664  0.117613
5   E       NaN -1.797024

In [52]: outer_join['value_x'] + outer_join['value_y']
Out[52]:
0         NaN
1    0.438072
2         NaN
3    0.630036
4    2.161278
5         NaN
dtype: float64

In [53]: outer_join['value_x'].sum()
Out[53]: 3.0136973094137085
```

One difference is that missing data cannot be compared to its sentinel value. For example, in SAS you could do this to filter missing values.

```
data outer_join_nulls;
    set outer_join;
    if value_x = .;
run;

data outer_join_no_nulls;
    set outer_join;
    if value_x ^= .;
run;
```

Which doesnt work in pandas. Instead, the pd.isna or pd.notna functions should be used for comparisons.

```
In [54]: outer_join[pd.isna(outer_join['value_x'])]
Out[54]:
  key  value_x   value_y
5   E      NaN -1.797024

In [55]: outer_join[pd.notna(outer_join['value_x'])]
Out[55]:
  key   value_x   value_y
0   A -1.579390       NaN
1   B  0.454513 -0.016440
2   C  0.051246       NaN
3   D  2.043664 -1.413628
```

```
4   D  2.043664  0.117613
```

pandas also provides a variety of methods to work with missing data - some of which would be challenging to express in SAS. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the *missing data documentation* for more.

```
In [56]: outer_join.dropna()
Out[56]:
  key   value_x   value_y
1   B  0.454513 -0.016440
3   D  2.043664 -1.413628
4   D  2.043664  0.117613

In [57]: outer_join.fillna(method='ffill')
Out[57]:
  key   value_x   value_y
0   A -1.579390       NaN
1   B  0.454513 -0.016440
2   C  0.051246 -0.016440
3   D  2.043664 -1.413628
4   D  2.043664  0.117613
5   E  2.043664 -1.797024

In [58]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
Out[58]:
0   -1.579390
1    0.454513
2    0.051246
3    2.043664
4    2.043664
5    0.602739
Name: value_x, dtype: float64
```

GroupBy

Aggregation

SASs PROC SUMMARY can be used to group by one or more key variables and compute aggregations on numeric columns.

```
proc summary data=tips nway;
    class sex smoker;
    var total_bill tip;
    output out=tips_summed sum=;
run;
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [59]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()

In [60]: tips_summed.head()
Out[60]:
              total_bill     tip
```

(continues on next page)

(continued from previous page)

```
sex    smoker
Female No            869.68  149.77
       Yes           527.27   96.74
Male   No           1725.75  302.00
       Yes          1217.07  183.07
```

Transformation

In SAS, if the group aggregations need to be used with the original frame, it must be merged back together. For example, to subtract the mean for each observation by smoker group.

```
proc summary data=tips missing nway;
    class smoker;
    var total_bill;
    output out=smoker_means mean(total_bill)=group_bill;
run;

proc sort data=tips;
    by smoker;
run;

data tips;
    merge tips(in=a) smoker_means(in=b);
    by smoker;
    adj_total_bill = total_bill - group_bill;
    if a and b;
run;
```

pandas groupby provides a transform mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [61]: gb = tips.groupby('smoker')['total_bill']

In [62]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [63]: tips.head()
Out[63]:
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By group processing

In addition to aggregation, pandas groupby can be used to replicate most other by group processing from SAS. For example, this DATA step reads the data by sex/smoker group and filters to the first entry for each.

```
proc sort data=tips;
   by sex smoker;
run;
```

(continues on next page)

(continued from previous page)

```
data tips_first;
    set tips;
    by sex smoker;
    if FIRST.sex or FIRST.smoker then output;
run;
```

In pandas this would be written as:

```
In [64]: tips.groupby(['sex', 'smoker']).first()
Out[64]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other Considerations

Disk vs memory

pandas operates exclusively in memory, where a SAS data set exists on disk. This means that the size of data able to be loaded in pandas is limited by your machines memory, but also that the operations on that data may be faster.

If out of core processing is needed, one possibility is the dask.dataframe library (currently in development) which provides a subset of pandas functionality for an on-disk `DataFrame`

Data interop

pandas provides a `read_sas()` method that can read SAS data saved in the XPORT or SAS7BDAT binary format.

```
libname xportout xport 'transport-file.xpt';
data xportout.tips;
    set tips(rename=(total_bill=tbill));
    * xport variable names limited to 6 characters;
run;
```

```
df = pd.read_sas('transport-file.xpt')
df = pd.read_sas('binary-file.sas7bdat')
```

You can also specify the file format directly. By default, pandas will try to infer the file format based on its extension.

```
df = pd.read_sas('transport-file.xpt', format='xport')
df = pd.read_sas('binary-file.sas7bdat', format='sas7bdat')
```

XPORT is a relatively limited format and the parsing of it is not as optimized as some of the other pandas readers. An alternative way to interop data between SAS and pandas is to serialize to csv.

```
# version 0.17, 10M rows

In [8]: %time df = pd.read_sas('big.xpt')
Wall time: 14.6 s

```

(continues on next page)

(continued from previous page)

```
In [9]: %time df = pd.read_csv('big.csv')
Wall time: 4.86 s
```

{{ header }}

3.5.4 Comparison with Stata

For potential users coming from Stata this page is meant to demonstrate how different Stata operations would be performed in pandas.

If youre new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows. This means that we can refer to the libraries as pd and np, respectively, for the rest of the document.

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

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

Note: Throughout this tutorial, the pandas DataFrame will be displayed by calling df.head(), which displays the first N (default 5) rows of the DataFrame. This is often used in interactive work (e.g. Jupyter notebook or terminal) – the equivalent in Stata would be:

```
list in 1/5
```

Data structures

General terminology translation

pandas	Stata
DataFrame	data set
column	variable
row	observation
groupby	bysort
NaN	.

DataFrame / Series

A DataFrame in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

A Series is the data structure that represents one column of a DataFrame. Stata doesnt have a separate data structure for a single column, but in general, working with a Series is analogous to referencing a column of a data set in Stata.

Index

Every DataFrame and Series has an Index – labels on the *rows* of the data. Stata does not have an exactly analogous concept. In Stata, a data sets rows are essentially unlabeled, other than an implicit integer index that can be accessed with _n.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the Index and just treat the DataFrame as a collection of columns. Please see the *indexing documentation* for much more on how to use an Index effectively.

Data input / output

Constructing a DataFrame from values

A Stata data set can be built from specified values by placing the data after an input statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

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

In [4]: df
Out[4]: 
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Like Stata, pandas provides utilities for reading in data from many formats. The tips data set, found within the pandas tests (csv) will be used in many of the following examples.

Stata provides import delimited to read csv data into a data set in memory. If the tips.csv file is in the current working directory, we can import it as follows.

```
import delimited tips.csv
```

The pandas method is read_csv(), which works similarly. Additionally, it will automatically download the data set if presented with a url.

```
In [5]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/data/tips.csv')
   ...: 

In [6]: tips = pd.read_csv(url)
```

(continues on next page)

(continued from previous page)

```
In [7]: tips.head()
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Like `import delimited`, `read_csv()` can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

Pandas can also read Stata data sets in `.dta` format with the `read_stata()` function.

```
df = pd.read_stata('data.dta')
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a `pd.read_*` function. See the *IO documentation* for more details.

Exporting data

The inverse of `import delimited` in Stata is `export delimited`

```
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is `DataFrame.to_csv()`.

```
tips.to_csv('tips2.csv')
```

Pandas can also export to Stata file format with the `DataFrame.to_stata()` method.

```
tips.to_stata('tips2.dta')
```

Data operations

Operations on columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```
replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill
```

pandas provides similar vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2

In [10]: tips.head()
Out[10]:
   total_bill   tip     sex smoker  day    time  size  new_bill
0       14.99  1.01  Female     No  Sun  Dinner     2     7.495
1        8.34  1.66    Male     No  Sun  Dinner     3     4.170
2       19.01  3.50    Male     No  Sun  Dinner     3     9.505
3       21.68  3.31    Male     No  Sun  Dinner     2    10.840
4       22.59  3.61  Female     No  Sun  Dinner     4    11.295

In [11]: tips = tips.drop('new_bill', axis=1)
```

Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [12]: tips[tips['total_bill'] > 10].head()
Out[12]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/then logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [13]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [14]: tips.head()
Out[14]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the *timeseries documentation* for more details.

```
In [15]: tips['date1'] = pd.Timestamp('2013-01-15')

In [16]: tips['date2'] = pd.Timestamp('2015-02-15')

In [17]: tips['date1_year'] = tips['date1'].dt.year

In [18]: tips['date2_month'] = tips['date2'].dt.month

In [19]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [20]: tips['months_between'] = (tips['date2'].dt.to_period('M')
   ....:                           - tips['date1'].dt.to_period('M'))
   ....:

In [21]: tips[['date1', 'date2', 'date1_year', 'date2_month', 'date1_next',
   ....:        'months_between']].head()
   ....:
Out[21]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of columns

Stata provides keywords to select, drop, and rename columns.

```
keep sex total_bill tip

drop sex

rename total_bill total_bill_2
```

The same operations are expressed in pandas below. Note that in contrast to Stata, these operations do not happen in place. To make these changes persist, assign the operation back to a variable.

```
# keep
In [22]: tips[['sex', 'total_bill', 'tip']].head()
Out[22]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61

# drop
In [23]: tips.drop('sex', axis=1).head()
Out[23]:
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [24]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
Out[24]:
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas objects have a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```
In [25]: tips = tips.sort_values(['sex', 'total_bill'])

In [26]: tips.head()
Out[26]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String processing

Finding length of string

Stata determines the length of a character string with the `strlen()` and `ustrlen()` functions for ASCII and Unicode strings, respectively.

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

Python determines the length of a character string with the `len` function. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [27]: tips['time'].str.len().head()
Out[27]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64

In [28]: tips['time'].str.rstrip().str.len().head()
Out[28]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64
```

Finding position of substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
generate str_position = strpos(sex, "ale")
```

Python determines the position of a character in a string with the `find()` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [29]: tips['sex'].str.find("ale").head()
Out[29]:
67     3
92     3
111    3
145    3
135    3
Name: sex, dtype: int64
```

Extracting substring by position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [30]: tips['sex'].str[0:1].head()
Out[30]:
67     F
92     F
111    F
145    F
135    F
Name: sex, dtype: object
```

Extracting nth word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [31]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [32]: firstlast['First_Name'] = firstlast['string'].str.split(" ", expand=True)[0]

In [33]: firstlast['Last_Name'] = firstlast['string'].str.rsplit(" ", expand=True)[0]

In [34]: firstlast
Out[34]:
       string First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Changing case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end
```

(continues on next page)

(continued from previous page)

```
generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [35]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [36]: firstlast['upper'] = firstlast['string'].str.upper()

In [37]: firstlast['lower'] = firstlast['string'].str.lower()

In [38]: firstlast['title'] = firstlast['string'].str.title()

In [39]: firstlast
Out[39]: 
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [40]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....: 

In [41]: df1
Out[41]: 
  key     value
0   A -0.358468
1   B -0.697821
2   C  0.162113
3   D -0.076884

In [42]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....: 

In [43]: df2
Out[43]: 
  key     value
0   B -0.235803
1   D  0.449752
2   D -2.220693
3   E  0.186546
```

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
end
generate value = rnormal()
save df2.dta

* Now create df1 in memory
clear
input str1 key
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas DataFrames have a `DataFrame.merge()` method, which provides similar functionality. Note that different join types are accomplished via the `how` keyword.

```
In [44]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [45]: inner_join
Out[45]: 
  key   value_x   value_y
0   B -0.697821 -0.235803
1   D -0.076884  0.449752
2   D -0.076884 -2.220693

In [46]: left_join = df1.merge(df2, on=['key'], how='left')

In [47]: left_join
Out[47]: 
```

(continues on next page)

(continued from previous page)

```
  key   value_x   value_y
0   A -0.358468       NaN
1   B -0.697821 -0.235803
2   C  0.162113       NaN
3   D -0.076884  0.449752
4   D -0.076884 -2.220693

In [48]: right_join = df1.merge(df2, on=['key'], how='right')

In [49]: right_join
Out[49]: 
  key   value_x   value_y
0   B -0.697821 -0.235803
1   D -0.076884  0.449752
2   D -0.076884 -2.220693
3   E       NaN  0.186546

In [50]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [51]: outer_join
Out[51]: 
  key   value_x   value_y
0   A -0.358468       NaN
1   B -0.697821 -0.235803
2   C  0.162113       NaN
3   D -0.076884  0.449752
4   D -0.076884 -2.220693
5   E       NaN  0.186546
```

Missing data

Like Stata, pandas has a representation for missing data – the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [52]: outer_join
Out[52]: 
  key   value_x   value_y
0   A -0.358468       NaN
1   B -0.697821 -0.235803
2   C  0.162113       NaN
3   D -0.076884  0.449752
4   D -0.076884 -2.220693
5   E       NaN  0.186546

In [53]: outer_join['value_x'] + outer_join['value_y']
Out[53]: 
0         NaN
1   -0.933624
2         NaN
3    0.372867
4   -2.297577
5         NaN
dtype: float64
```

```
In [54]: outer_join['value_x'].sum()
Out[54]: -1.0479449929596008
```

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

This doesnt work in pandas. Instead, the `pd.isna()` or `pd.notna()` functions should be used for comparisons.

```
In [55]: outer_join[pd.isna(outer_join['value_x'])]
Out[55]:
  key  value_x   value_y
5   E      NaN  0.186546

In [56]: outer_join[pd.notna(outer_join['value_x'])]
Out[56]:
  key   value_x   value_y
0   A -0.358468       NaN
1   B -0.697821 -0.235803
2   C  0.162113       NaN
3   D -0.076884  0.449752
4   D -0.076884 -2.220693
```

Pandas also provides a variety of methods to work with missing data – some of which would be challenging to express in Stata. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the *missing data documentation* for more.

```
# Drop rows with any missing value
In [57]: outer_join.dropna()
Out[57]:
  key   value_x   value_y
1   B -0.697821 -0.235803
3   D -0.076884  0.449752
4   D -0.076884 -2.220693

# Fill forwards
In [58]: outer_join.fillna(method='ffill')
Out[58]:
  key   value_x   value_y
0   A -0.358468       NaN
1   B -0.697821 -0.235803
2   C  0.162113 -0.235803
3   D -0.076884  0.449752
4   D -0.076884 -2.220693
5   E -0.076884  0.186546

# Impute missing values with the mean
In [59]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
Out[59]:
0   -0.358468
1   -0.697821
```

```
2    0.162113
3   -0.076884
4   -0.076884
5   -0.209589
Name: value_x, dtype: float64
```

GroupBy

Aggregation

Statas `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [60]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()

In [61]: tips_summed.head()
Out[61]: 
               total_bill     tip
sex    smoker                    
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [62]: gb = tips.groupby('smoker')['total_bill']

In [63]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [64]: tips.head()
Out[64]: 
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [65]: tips.groupby(['sex', 'smoker']).first()
Out[65]: 
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker                                                      
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other considerations

Disk vs memory

Pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machines memory. If out of core processing is needed, one possibility is the dask.dataframe library, which provides a subset of pandas functionality for an on-disk `DataFrame`. {{ header }}

3.6 Tutorials

This is a guide to many pandas tutorials, geared mainly for new users.

3.6.1 Internal guides

pandas own *10 Minutes to pandas*.

More complex recipes are in the *Cookbook*.

A handy pandas cheat sheet.

3.6.2 Community guides

pandas Cookbook by Julia Evans

The goal of this 2015 cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails. For the table of contents, see the pandas-cookbook GitHub repository.

Learn Pandas by Hernan Rojas

A set of lesson for new pandas users: https://bitbucket.org/hrojas/learn-pandas

Practical data analysis with Python

This guide is an introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as munging data, aggregating data, visualizing data and time series.

Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

Modern pandas

Tutorial series written in 2016 by Tom Augspurger. The source may be found in the GitHub repository TomAugspurger/effective-pandas.

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization
- Timeseries

Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

Video tutorials

- Pandas From The Ground Up (2015) (2:24) GitHub repo
- Introduction Into Pandas (2016) (1:28) GitHub repo
- Pandas: .head() to .tail() (2016) (1:26) GitHub repo
- Data analysis in Python with pandas (2016-2018) GitHub repo and Jupyter Notebook
- Best practices with pandas (2018) GitHub repo and Jupyter Notebook

Various tutorials

- Wes McKinneys (pandas BDFL) blog
- Statistical analysis made easy in Python with SciPy and pandas DataFrames, by Randal Olson
- Statistical Data Analysis in Python, tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- Financial analysis in Python, by Thomas Wiecki
- Intro to pandas data structures, by Greg Reda
- Pandas and Python: Top 10, by Manish Amde

- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples

{{ header }}

CHAPTER

FOUR

USER GUIDE

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as working with missing data), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

Further information on any specific method can be obtained in the *API reference*. {{ header }}

4.1 IO tools (text, CSV, HDF5,)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	*read_csv*	*to_csv*
text	JSON	*read_json*	*to_json*
text	HTML	*read_html*	*to_html*
text	Local clipboard	*read_clipboard*	*to_clipboard*
binary	MS Excel	*read_excel*	*to_excel*
binary	OpenDocument	*read_excel*	
binary	HDF5 Format	*read_hdf*	*to_hdf*
binary	Feather Format	*read_feather*	*to_feather*
binary	Parquet Format	*read_parquet*	*to_parquet*
binary	Msgpack	*read_msgpack*	*to_msgpack*
binary	Stata	*read_stata*	*to_stata*
binary	SAS	*read_sas*	
binary	Python Pickle Format	*read_pickle*	*to_pickle*
SQL	SQL	*read_sql*	*to_sql*
SQL	Google Big Query	*read_gbq*	*to_gbq*

Here is an informal performance comparison for some of these IO methods.

Note: For examples that use the `StringIO` class, make sure you import it according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

4.1.1 CSV & text files

The workhorse function for reading text files (a.k.a. flat files) is *read_csv()*. See the *cookbook* for some advanced strategies.

Parsing options

read_csv() accepts the following common arguments:

Basic

filepath_or_buffer [various] Either a path to a file (a `str`, `pathlib.Path`, or `py._path.local.LocalPath`), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or `StringIO`).

sep [str, defaults to `','` for *read_csv()*, `\t` for *read_table()*] Delimiter to use. If sep is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Pythons builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

delimiter [str, default `None`] Alternative argument name for sep.

delim_whitespace [boolean, default False] Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

Column and index locations and names

header [int or list of ints, default `'infer'`] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names.

The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so header=0 denotes the first line of data rather than the first line of the file.

names [array-like, default `None`] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list are not allowed.

index_col [int, str, sequence of int / str, or False, default `None`] Column(s) to use as the row labels of the `DataFrame`, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: `index_col=False` can be used to force pandas to *not* use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

usecols [list-like or callable, default `None`] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```
In [1]: from io import StringIO, BytesIO

In [2]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

In [3]: pd.read_csv(StringIO(data))
Out[3]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [4]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in␣
↪['COL1', 'COL3'])
Out[4]:
  col1  col3
0    a     1
1    a     2
2    c     3
```

Using this parameter results in much faster parsing time and lower memory usage.

squeeze [boolean, default `False`] If the parsed data only contains one column then return a `Series`.

prefix [str, default `None`] Prefix to add to column numbers when no header, e.g. X for X0, X1,

mangle_dupe_cols [boolean, default `True`] Duplicate columns will be specified as X, X.1X.N, rather than XX. Passing in `False` will cause data to be overwritten if there are duplicate names in the columns.

General parsing configuration

dtype [Type name or dict of column -> type, default `None`] Data type for data or columns. E.g. `{'a': np.float64, 'b': np.int32}` (unsupported with `engine='python'`). Use *str* or *object* together with suitable `na_values` settings to preserve and not interpret dtype.

New in version 0.20.0: support for the Python parser.

engine [{`'c'`, `'python'`}] Parser engine to use. The C engine is faster while the Python engine is currently more feature-complete.

converters [dict, default `None`] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, default `None`] Values to consider as `True`.

false_values [list, default `None`] Values to consider as `False`.

skipinitialspace [boolean, default `False`] Skip spaces after delimiter.

skiprows [list-like or integer, default None] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```
In [5]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

In [6]: pd.read_csv(StringIO(data))
Out[6]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [7]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[7]:
  col1 col2  col3
0    a    b     2
```

skipfooter [int, default 0] Number of lines at bottom of file to skip (unsupported with engine=c).

nrows [int, default None] Number of rows of file to read. Useful for reading pieces of large files.

low_memory [boolean, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser)

memory_map [boolean, default False] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

NA and missing data handling

na_values [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See *na values const* below for a list of the values interpreted as NaN by default.

keep_default_na [boolean, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is True, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is True, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is False, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is False, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as False, the *keep_default_na* and *na_values* parameters will be ignored.

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.

verbose [boolean, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [boolean, default True] If True, skip over blank lines rather than interpreting as NaN values.

Datetime handling

parse_dates [boolean or list of ints or names or list of lists or dict, default False.]

- 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.
- If {'foo': [1, 3]} -> parse columns 1, 3 as date and call result foo. A fast-path exists for iso8601-formatted dates.

infer_datetime_format [boolean, default False] If True and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

keep_date_col [boolean, default False] If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser [function, default None] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion. pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirst [boolean, default False] DD/MM format dates, international and European format.

cache_dates [boolean, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

Iteration

iterator [boolean, default False] Return *TextFileReader* object for iteration or getting chunks with get_chunk().

chunksize [int, default None] Return *TextFileReader* object for iteration. See *iterating and chunking* below.

Quoting, compression, and file format

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If infer, then use gzip, bz2, zip, or xz if filepath_or_buffer is a string ending in .gz, .bz2, .zip, or .xz, respectively, and no decompression otherwise. If using zip, the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for zip and xz compression.

Changed in version 0.24.0: infer option added and set to default.

thousands [str, default None] Thousands separator.

decimal [str, default '.'] Character to recognize as decimal point. E.g. use ',' for European data.

float_precision [string, default None] Specifies which converter the C engine should use for floating-point values. The options are None for the ordinary converter, high for the high-precision converter, and round_trip for the round-trip converter.

lineterminator [str (length 1), default None] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1)] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequote [boolean, default True] When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements **inside** a field as a single quotechar element.

escapechar [str (length 1), default None] One-character string used to escape delimiter when quoting is QUOTE_NONE.

comment [str, default None] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as skip_blank_lines=True), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if comment='#', parsing #empty\na,b,c\n1,2,3 with *header=0* will result in a,b,c being treated as the header.

encoding [str, default None] Encoding to use for UTF when reading/writing (e.g. 'utf-8'). List of Python standard encodings.

dialect [str or csv.Dialect instance, default None] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

Error handling

error_bad_lines [boolean, default True] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these bad lines will dropped from the DataFrame that is returned. See *bad lines* below.

warn_bad_lines [boolean, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each bad line will be output.

Specifying column data types

You can indicate the data type for the whole DataFrame or individual columns:

```
In [8]: data = ('a,b,c,d\n'
   ...:         '1,2,3,4\n'
   ...:         '5,6,7,8\n'
   ...:         '9,10,11')
   ...:

In [9]: print(data)
a,b,c,d
1,2,3,4
5,6,7,8
```

```
9,10,11

In [10]: df = pd.read_csv(StringIO(data), dtype=object)

In [11]: df
Out[11]:
   a   b   c    d
0  1   2   3    4
1  5   6   7    8
2  9  10  11  NaN

In [12]: df['a'][0]
Out[12]: '1'

In [13]: df = pd.read_csv(StringIO(data),
   ....:                  dtype={'b': object, 'c': np.float64, 'd': 'Int64'})
   ....:

In [14]: df.dtypes
Out[14]:
a      int64
b     object
c    float64
d      Int64
dtype: object
```

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one `dtype`. If youre unfamiliar with these concepts, you can see *here* to learn more about dtypes, and *here* to learn more about `object` conversion in pandas.

For instance, you can use the `converters` argument of *`read_csv()`*:

```
In [15]: data = ("col_1\n"
   ....:         "1\n"
   ....:         "2\n"
   ....:         "'A'\n"
   ....:         "4.22")
   ....:

In [16]: df = pd.read_csv(StringIO(data), converters={'col_1': str})

In [17]: df
Out[17]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [18]: df['col_1'].apply(type).value_counts()
Out[18]:
<class 'str'>    4
Name: col_1, dtype: int64
```

Or you can use the *`to_numeric()`* function to coerce the dtypes after reading in the data,

```
In [19]: df2 = pd.read_csv(StringIO(data))

In [20]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')

In [21]: df2
Out[21]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22

In [22]: df2['col_1'].apply(type).value_counts()
Out[22]:
<class 'float'>    4
Name: col_1, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as NaN.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to NaN out the data anomalies, then *to_numeric()* is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the converters argument of *read_csv()* would certainly be worth trying.

New in version 0.20.0: support for the Python parser.

The dtype option is supported by the python engine.

Note: In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [23]: col_1 = list(range(500000)) + ['a', 'b'] + list(range(500000))

In [24]: df = pd.DataFrame({'col_1': col_1})

In [25]: df.to_csv('foo.csv')

In [26]: mixed_df = pd.read_csv('foo.csv')

In [27]: mixed_df['col_1'].apply(type).value_counts()
Out[27]:
<class 'int'>    737858
<class 'str'>    262144
Name: col_1, dtype: int64

In [28]: mixed_df['col_1'].dtype
Out[28]: dtype('O')
```

will result with *mixed_df* containing an int dtype for certain chunks of the column, and str for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a dtype of object, which is used for columns with mixed dtypes.

Specifying categorical dtype

New in version 0.19.0.

`Categorical` columns can be parsed directly by specifying `dtype='category'` or `dtype=CategoricalDtype(categories, ordered)`.

```
In [29]: data = ('col1,col2,col3\n'
   ....:         'a,b,1\n'
   ....:         'a,b,2\n'
   ....:         'c,d,3')
   ....:

In [30]: pd.read_csv(StringIO(data))
Out[30]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [31]: pd.read_csv(StringIO(data)).dtypes
Out[31]:
col1    object
col2    object
col3     int64
dtype: object

In [32]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[32]:
col1    category
col2    category
col3    category
dtype: object
```

Individual columns can be parsed as a `Categorical` using a dict specification:

```
In [33]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[33]:
col1    category
col2      object
col3       int64
dtype: object
```

New in version 0.21.0.

Specifying `dtype='category'` will result in an unordered `Categorical` whose `categories` are the unique values observed in the data. For more control on the categories and order, create a `CategoricalDtype` ahead of time, and pass that for that columns `dtype`.

```
In [34]: from pandas.api.types import CategoricalDtype

In [35]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[36]:
col1    category
col2      object
```

(continues on next page)

(continued from previous page)

```
col3        int64
dtype: object
```

When using `dtype=CategoricalDtype`, unexpected values outside of `dtype.categories` are treated as missing values.

```
In [37]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [38]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[38]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

This matches the behavior of `Categorical.set_categories()`.

Note: With `dtype='category'`, the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the *`to_numeric()`* function, or as appropriate, another converter such as *`to_datetime()`*.

When `dtype` is a `CategoricalDtype` with homogeneous `categories` (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [39]: df = pd.read_csv(StringIO(data), dtype='category')

In [40]: df.dtypes
Out[40]:
col1    category
col2    category
col3    category
dtype: object

In [41]: df['col3']
Out[41]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [42]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [43]: df['col3']
Out[43]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and using columns

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 [44]: data = ('a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [45]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [46]: pd.read_csv(StringIO(data))
Out[46]:
   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 [47]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [48]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[48]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [49]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[49]:
  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 [50]: data = ('skip this skip it\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
```

(continues on next page)

(continued from previous page)

```
   ....:         '7,8,9')
   ....:

In [51]: pd.read_csv(StringIO(data), header=1)
Out[51]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Note: Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to `header=None`.

Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```
In [52]: data = ('a,b,a\n'
   ....:         '0,1,2\n'
   ....:         '3,4,5')
   ....:

In [53]: pd.read_csv(StringIO(data))
Out[53]:
   a  b  a.1
0  0  1    2
1  3  4    5
```

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns X, , X to become X, X.1, , X.N. If `mangle_dupe_cols=False`, duplicate data can arise:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
   a  b  a
0  2  1  2
1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

Filtering columns (`usecols`)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

New in version 0.20.0: support for callable *usecols* arguments

```
In [54]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [55]: pd.read_csv(StringIO(data))
Out[55]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [56]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[56]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [57]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[57]:
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [58]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A',␣
 ↪'C'])
Out[58]:
   a  c
0  1  3
1  4  6
2  7  9
```

The `usecols` argument can also be used to specify which columns not to use in the final result:

```
In [59]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[59]:
   b    d
0  2  foo
1  5  bar
2  8  baz
```

In this case, the callable is specifying that we exclude the a and c columns from the output.

Comments and empty lines

Ignoring line comments and empty lines

If the `comment` parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

```
In [60]: data = ('\n'
   ....:         'a,b,c\n'
   ....:         '  \n'
   ....:         '# commented line\n'
   ....:            '1,2,3\n'
   ....:            '\n'
   ....:            '4,5,6')
   ....:

In [61]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [62]: pd.read_csv(StringIO(data), comment='#')
Out[62]:
   a  b  c
0  1  2  3
1  4  5  6
```

If `skip_blank_lines=False`, then `read_csv` will not ignore blank lines:

```
In [63]: data = ('a,b,c\n'
   ....:         '\n'
   ....:         '1,2,3\n'
   ....:         '\n'
   ....:         '\n'
   ....:         '4,5,6')
   ....:

In [64]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[64]:
     a    b    c
0  NaN  NaN  NaN
1  1.0  2.0  3.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  4.0  5.0  6.0
```

Warning: The presence of ignored lines might create ambiguities involving line numbers; the parameter `header` uses row numbers (ignoring commented/empty lines), while `skiprows` uses line numbers (including commented/empty lines):

```
In [65]: data = ('#comment\n'
   ....:         'a,b,c\n'
   ....:         'A,B,C\n'
   ....:         '1,2,3')
   ....:

In [66]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[66]:
   A  B  C
0  1  2  3

In [67]: data = ('A,B,C\n'
   ....:         '#comment\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3')
   ....:

In [68]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[68]:
   a  b  c
0  1  2  3
```

If both header and skiprows are specified, header will be relative to the end of skiprows. For example:

```
In [69]: data = ('# empty\n'
   ....:         '# second empty line\n'
   ....:         '# third emptyline\n'
   ....:         'X,Y,Z\n'
   ....:         '1,2,3\n'
   ....:         'A,B,C\n'
   ....:         '1,2.,4.\n'
   ....:         '5.,NaN,10.0\n')
   ....:

In [70]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0


In [71]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[71]:
     A    B     C
0  1.0  2.0   4.0
1  5.0  NaN  10.0
```

Comments

Sometimes comments or meta data may be included in a file:

```
In [72]: 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 parser includes the comments in the output:

```
In [73]: df = pd.read_csv('tmp.csv')

In [74]: df
Out[74]:
         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 [75]: df = pd.read_csv('tmp.csv', comment='#')

In [76]: df
Out[76]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z
```

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 [77]: data = (b'word,length\n'
   ....:         b'Tr\xc3\xa4umen,7\n'
   ....:         b'Gr\xc3\xbc\xc3\x9fe,5')
   ....:

In [78]: data = data.decode('utf8').encode('latin-1')

In [79]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [80]: df
Out[80]:
      word  length
0  Träumen       7
1    GrüSSe       5

In [81]: df['word'][1]
Out[81]: 'GrüSSe'
```

Some formats which encode all characters as multiple bytes, like UTF-16, wont parse correctly at all without specifying the encoding. Full list of Python standard encodings.

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 `DataFrames` row names:

```
In [82]: data = ('a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....:

In [83]: pd.read_csv(StringIO(data))
Out[83]:
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

```
In [84]: data = ('index,a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
```

(continues on next page)

(continued from previous page)

```
   ....:          '8,orange,cow,10')
   ....:

In [85]: pd.read_csv(StringIO(data), index_col=0)
Out[85]: 
            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 [86]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [87]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [88]: pd.read_csv(StringIO(data))
Out[88]: 
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [89]: pd.read_csv(StringIO(data), index_col=False)
Out[89]: 
   a       b    c
0  4   apple  bat
1  8  orange  cow
```

If a subset of data is being parsed using the `usecols` option, the `index_col` specification is based on that subset, not the original data.

```
In [90]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [91]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [92]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
Out[92]: 
     b   c
4  bat NaN
8  cow NaN
```

```
In [93]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
Out[93]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying date columns

To better facilitate working with datetime data, *read_csv()* 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 [94]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [95]: df
Out[95]:
            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 [96]: df.index
Out[96]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'],␣
 ↪dtype='datetime64[ns]', name='date', freq=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 [97]: 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 [98]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [99]: df
Out[99]:
                  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
```

(continues on next page)

(continued from previous page)

```
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 [100]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   .....:                  keep_date_col=True)
   .....:

In [101]: df
Out[101]: 
                  1_2                 1_3     0         1          2          3     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00   18:56:00  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00   19:56:00  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00   20:56:00 -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00   21:18:00 -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00   21:56:00 -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00   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 [102]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [104]: df
Out[104]: 
              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 [105]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  index_col=0)  # index is the nominal column
   .....:

In [107]: df
Out[107]: 
                                 actual     0     4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
```

(continues on next page)

(continued from previous page)

```
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: If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `to_datetime()` after `pd.read_csv`.

Note: read_csv has a fast_path for parsing datetime strings in iso8601 format, e.g 2000-01-01T00:01:02+00:00 and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

Note: When passing a dict as the *parse_dates* argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use *collections.OrderedDict* instead of a regular *dict* if this matters to you. Because of this, when using a dict for parse_dates in conjunction with the *index_col* argument, its best to specify *index_col* as a column label rather then as an index on the resulting frame.

Date parsing functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```
In [108]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  date_parser=pd.io.date_converters.parse_date_time)
   .....:

In [109]: df
Out[109]:
              nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Pandas will try to call the `date_parser` function in three different ways. If an exception is raised, the next one is tried:

1. `date_parser` is first called with one or more arrays as arguments, as defined using *parse_dates* (e.g., `date_parser(['2013', '2013'], ['1', '2'])`).
2. If #1 fails, `date_parser` is called with all the columns concatenated row-wise into a single array (e.g., `date_parser(['2013 1', '2013 2'])`).
3. If #2 fails, `date_parser` is called once for every row with one or more string arguments from the columns indicated with *parse_dates* (e.g., `date_parser('2013', '1')` for the first row, `date_parser('2013', '2')` for the second, etc.).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using `infer_datetime_format=True` (see section below).
2. If you know the format, use `pd.to_datetime()`: `date_parser=lambda x: pd.to_datetime(x, format=...)`.
3. If you have a really non-standard format, use a custom `date_parser` function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

You can explore the date parsing functionality in date_converters.py and add your own. We would love to turn this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contains functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single date rather than the entire array.

Parsing a CSV with mixed timezones

Pandas cannot natively represent a column or index with mixed timezones. If your CSV file contains columns with a mixture of timezones, the default result will be an object-dtype column with strings, even with `parse_dates`.

```
In [110]: content = """\
   .....: a
   .....: 2000-01-01T00:00:00+05:00
   .....: 2000-01-01T00:00:00+06:00"""
   .....:

In [111]: df = pd.read_csv(StringIO(content), parse_dates=['a'])

In [112]: df['a']
Out[112]:
0    2000-01-01 00:00:00+05:00
1    2000-01-01 00:00:00+06:00
Name: a, dtype: object
```

To parse the mixed-timezone values as a datetime column, pass a partially-applied *`to_datetime()`* with `utc=True` as the `date_parser`.

```
In [113]: df = pd.read_csv(StringIO(content), parse_dates=['a'],
   .....:                  date_parser=lambda col: pd.to_datetime(col, utc=True))
   .....:

In [114]: df['a']
Out[114]:
0   1999-12-31 19:00:00+00:00
1   1999-12-31 18:00:00+00:00
Name: a, dtype: datetime64[ns, UTC]
```

Inferring datetime format

If you have `parse_dates` enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting `infer_datetime_format=True`. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, `infer_datetime_format` should not have any negative consequences if enabled.

Here are some examples of datetime strings that can be guessed (All representing December 30th, 2011 at 00:00:00):

- 20111230
- 2011/12/30
- 20111230 00:00:00
- 12/30/2011 00:00:00
- 30/Dec/2011 00:00:00
- 30/December/2011 00:00:00

Note that `infer_datetime_format` is sensitive to `dayfirst`. With `dayfirst=True`, it will guess 01/12/2011 to be December 1st. With `dayfirst=False` (default) it will guess 01/12/2011 to be January 12th.

```
# Try to infer the format for the index column
In [115]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
   .....:                  infer_datetime_format=True)
   .....:

In [116]: df
Out[116]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

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 [117]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [118]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[118]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c

In [119]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[119]:
        date  value cat
0 2000-06-01      5   a
1 2000-06-02     10   b
2 2000-06-03     15   c
```

Specifying method for floating-point conversion

The parameter `float_precision` can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```
In [120]: val = '0.3066101993807095471566981359501369297504425048828125'

In [121]: data = 'a,b,c\n1,2,{0}'.format(val)

In [122]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                  float_precision=None)['c'][0] - float(val))
   .....:
Out[122]: 1.1102230246251565e-16

In [123]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                  float_precision='high')['c'][0] - float(val))
   .....:
Out[123]: 5.551115123125783e-17

In [124]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                  float_precision='round_trip')['c'][0] - float(val))
   .....:
Out[124]: 0.0
```

Thousand separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```
In [125]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [126]: df = pd.read_csv('tmp.csv', sep='|')

In [127]: df
Out[127]:
         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [128]: df.level.dtype
Out[128]: dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly:

```
In [129]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
```

```
Patient3|1,234,018|z

In [130]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [131]: df
Out[131]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [132]: df.level.dtype
Out[132]: dtype('int64')
```

NA values

To control which values are parsed as missing values (which are signified by NaN), specify a string in na_values. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0, 5] are recognized as NaN).

To completely override the default values that are recognized as missing, specify keep_default_na=False.

The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', ''].

Let us consider some examples:

```
pd.read_csv('path_to_file.csv', na_values=[5])
```

In the example above 5 and 5.0 will be recognized as NaN, in addition to the defaults. A string will first be interpreted as a numerical 5, then as a NaN.

```
pd.read_csv('path_to_file.csv', keep_default_na=False, na_values=[""])
```

Above, only an empty field will be recognized as NaN.

```
pd.read_csv('path_to_file.csv', keep_default_na=False, na_values=["NA", "0"])
```

Above, both NA and 0 as strings are NaN.

```
pd.read_csv('path_to_file.csv', na_values=["Nope"])
```

The default values, in addition to the string "Nope" are recognized as NaN.

Infinity

inf like values will be parsed as np.inf (positive infinity), and -inf as -np.inf (negative infinity). These will ignore the case of the value, meaning Inf, will also be parsed as np.inf.

Returning Series

Using the squeeze keyword, the parser will return output with a single column as a Series:

```
In [133]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [134]: output = pd.read_csv('tmp.csv', squeeze=True)

In [135]: output
Out[135]:
Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [136]: type(output)
Out[136]: pandas.core.series.Series
```

Boolean values

The common values `True`, `False`, `TRUE`, and `FALSE` are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the `true_values` and `false_values` options as follows:

```
In [137]: data = ('a,b,c\n'
   .....:         '1,Yes,2\n'
   .....:         '3,No,4')
   .....:

In [138]: print(data)
a,b,c
1,Yes,2
3,No,4

In [139]: pd.read_csv(StringIO(data))
Out[139]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [140]: pd.read_csv(StringIO(data), true_values=['Yes'], false_
↪values=['No'])
Out[140]:
   a      b  c
0  1   True  2
1  3  False  4
```

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 fields will raise an error by default:

```
In [141]: data = ('a,b,c\n'
   .....:         '1,2,3\n'
```

(continues on next page)

(continued from previous page)

```
   .....:          '4,5,6,7\n'
   .....:          '8,9,10')
   .....:

In [142]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
<ipython-input-142-6388c394e6b8> in <module>
----> 1 pd.read_csv(StringIO(data))

~/sandbox/pandas-release/pandas/pandas/io/parsers.py in parser_f(filepath_or_buffer,␣
↪sep, delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_
↪cols, dtype, engine, converters, true_values, false_values, skipinitialspace,␣
↪skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_
↪blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser,␣
↪dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal,␣
↪lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding,␣
↪dialect, error_bad_lines, warn_bad_lines, delim_whitespace, low_memory, memory_map,␣
↪float_precision)
    683         )
    684
--> 685         return _read(filepath_or_buffer, kwds)
    686
    687     parser_f.__name__ = name

~/sandbox/pandas-release/pandas/pandas/io/parsers.py in _read(filepath_or_buffer,␣
↪kwds)
    461
    462     try:
--> 463         data = parser.read(nrows)
    464     finally:
    465         parser.close()

~/sandbox/pandas-release/pandas/pandas/io/parsers.py in read(self, nrows)
   1152     def read(self, nrows=None):
   1153         nrows = _validate_integer("nrows", nrows)
-> 1154         ret = self._engine.read(nrows)
   1155
   1156         # May alter columns / col_dict

~/sandbox/pandas-release/pandas/pandas/io/parsers.py in read(self, nrows)
   2057     def read(self, nrows=None):
   2058         try:
-> 2059             data = self._reader.read(nrows)
   2060         except StopIteration:
   2061             if self._first_chunk:

~/sandbox/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.
↪TextReader.read()

~/sandbox/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.
↪TextReader._read_low_memory()

~/sandbox/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.
↪TextReader._read_rows()

~/sandbox/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.
↪TextReader._tokenize_rows()
```

(continues on next page)

(continued from previous page)

```
~/sandbox/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.
↪raise_parser_error()

ParserError: 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
```

You can also use the `usecols` parameter to eliminate extraneous column data that appear in some lines but not others:

```
In [30]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

 Out[30]:
    a  b   c
 0  1  2   3
 1  4  5   6
 2  8  9  10
```

Dialect

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 [143]: 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 [144]: import csv

In [145]: dia = csv.excel()

In [146]: dia.quoting = csv.QUOTE_NONE

In [147]: pd.read_csv(StringIO(data), dialect=dia)
Out[147]:
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [148]: data = 'a,b,c~1,2,3~4,5,6'

In [149]: pd.read_csv(StringIO(data), lineterminator='~')
Out[149]: 
   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 [150]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [151]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [152]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[152]: 
   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to do the right thing and not be fragile. Type inference is a pretty big deal. If a column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

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 [153]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [154]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [155]: pd.read_csv(StringIO(data), escapechar='\\')
Out[155]: 
                               a  b
0  hello, "Bob", nice to see you  5
```

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, and a different usage of the `delimiter` parameter:

- `colspecs`: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value infer can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.
- `widths`: A list of field widths which can be used instead of colspecs if the intervals are contiguous.

- `delimiter`: Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., ~).

Consider a typical fixed-width data file:

```
In [156]: 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 [157]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [158]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [159]: df
Out[159]:
                 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 [160]: widths = [6, 14, 13, 10]

In [161]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [162]: df
Out[162]:
        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 its ok to have extra separation between the columns in the file.

By default, `read_fwf` will try to infer the files `colspecs` by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided `delimiter` (default delimiter is whitespace).

```
In [163]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [164]: df
Out[164]:
```

(continues on next page)

(continued from previous page)

```
                 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
```

New in version 0.20.0.

`read_fwf` supports the `dtype` parameter for specifying the types of parsed columns to be different from the inferred type.

```
In [165]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[165]:
1    float64
2    float64
3    float64
dtype: object

In [166]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[166]:
0     object
1    float64
2     object
3    float64
dtype: object
```

Indexes

Files with an implicit index column

Consider a file with one less entry in the header than the number of data column:

```
In [167]: 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 [168]: pd.read_csv('foo.csv')
Out[168]:
          A  B  C
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Note that the dates werent automatically parsed. In that case you would need to do as before:

```
In [169]: df = pd.read_csv('foo.csv', parse_dates=True)
```

(continues on next page)

(continued from previous page)

```
In [170]: df.index
Out[170]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype=
↪'datetime64[ns]', freq=None)
```

Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [171]: 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` can take a list of column numbers to turn multiple columns into a `MultiIndex` for the index of the returned object:

```
In [172]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [173]: df
Out[173]: 
             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 [174]: df.loc[1978]
Out[174]: 
       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
```

Reading columns with a `MultiIndex`

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the intervening rows.

```
In [175]: from pandas.util.testing import makeCustomDataframe as mkdf

In [176]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [177]: df.to_csv('mi.csv')

In [178]: 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 [179]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
Out[179]:
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
```

`read_csv` is also able to interpret a more common format of multi-columns indices.

```
In [180]: print(open('mi2.csv').read())
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12

In [181]: pd.read_csv('mi2.csv', header=[0, 1], index_col=0)
Out[181]:
     a         b   c
     q  r  s   t   u   v
one  1  2  3   4   5   6
two  7  8  9  10  11  12
```

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

Automatically sniffing the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files, as pandas uses the `csv.Sniffer` class of the csv module. For this, you have to specify `sep=None`.

```
In [182]: print(open('tmp2.sv').read())
:0:1:2:3
0:1.1214905765122583:-1.1011663421613171:-1.2725711408453018:0.
↪8434589457722285
1:0.8739661419816901:-1.1622548707272122:0.12618578996106738:0.
↪5057848504967111
2:0.6695152369722812:0.4833977900441433:-0.4383565886430891:-0.
↪13952146077085656
3:1.6678766138462109:0.906356209978661:0.8603041052486606:-0.
↪009413710135323125
4:-0.8075485015292924:-0.7848128653629299:-1.3155155066668116:0.
↪6875244729698119
5:-0.1572352664979729:0.30339976035788174:-0.36340691002502046:-0.
↪5526511482544121
6:0.41442095212262187:0.17517103850750262:-0.5295157789486404:-0.
↪06745694327155764
7:1.058814717443789:-0.11789792502832808:-1.8534207864364352:-0.
↪7018494437516053
8:0.26239634172416604:-1.7245959745828128:0.2765803759042711:1.
↪0730241342647273
9:0.6352851164219758:-2.1785482358583024:0.3120437647651685:1.5723784501068536


In [183]: pd.read_csv('tmp2.sv', sep=None, engine='python')
Out[183]:
   Unnamed: 0         0         1         2         3
0           0  1.121491 -1.101166 -1.272571  0.843459
1           1  0.873966 -1.162255  0.126186  0.505785
2           2  0.669515  0.483398 -0.438357 -0.139521
3           3  1.667877  0.906356  0.860304 -0.009414
4           4 -0.807549 -0.784813 -1.315516  0.687524
5           5 -0.157235  0.303400 -0.363407 -0.552651
6           6  0.414421  0.175171 -0.529516 -0.067457
7           7  1.058815 -0.117898 -1.853421 -0.701849
8           8  0.262396 -1.724596  0.276580  1.073024
9           9  0.635285 -2.178548  0.312044  1.572378
```

Reading multiple files to create a single DataFrame

Its best to use *`concat()`* to combine multiple files. See the *cookbook* for an example.

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 [184]: print(open('tmp.sv').read())
|0|1|2|3
0|1.1214905765122583|-1.1011663421613171|-1.2725711408453018|0.8434589457722285
1|0.8739661419816901|-1.1622548707272122|0.12618578996106738|0.5057848504967111
2|0.6695152369722812|0.4833977900441433|-0.4383565886430891|-0.13952146077085656
3|1.6678766138462109|0.906356209978661|0.8603041052486606|-0.009413710135323125
4|-0.8075485015292924|-0.7848128653629299|-1.3155150666668116|0.6875244729698119
5|-0.1572352664979729|0.30339976035788174|-0.36340691002502046|-0.5526511482544121
6|0.41442095212262187|0.17517103850750262|-0.5295157789486404|-0.06745694327155764
7|1.058814717443789|-0.11789792502832808|-1.8534207864364352|-0.7018494437516053
8|0.26239634172416604|-1.7245959745828128|0.2765803759042711|1.0730241342647273
9|0.6352851164219758|-2.1785482358583024|0.3120437647651685|1.5723784501068536


In [185]: table = pd.read_csv('tmp.sv', sep='|')

In [186]: table
Out[186]: 
   Unnamed: 0         0         1         2         3
0           0  1.121491 -1.101166 -1.272571  0.843459
1           1  0.873966 -1.162255  0.126186  0.505785
2           2  0.669515  0.483398 -0.438357 -0.139521
3           3  1.667877  0.906356  0.860304 -0.009414
4           4 -0.807549 -0.784813 -1.315516  0.687524
5           5 -0.157235  0.303400 -0.363407 -0.552651
6           6  0.414421  0.175171 -0.529516 -0.067457
7           7  1.058815 -0.117898 -1.853421 -0.701849
8           8  0.262396 -1.724596  0.276580  1.073024
9           9  0.635285 -2.178548  0.312044  1.572378
```

By specifying a `chunksize` to `read_csv`, the return value will be an iterable object of type `TextFileReader`:

```
In [187]: reader = pd.read_csv('tmp.sv', sep='|', chunksize=4)

In [188]: reader
Out[188]: <pandas.io.parsers.TextFileReader at 0x1c334ff710>

In [189]: for chunk in reader:
   .....:     print(chunk)
   .....: 
  Unnamed: 0         0         1         2         3
0           0  1.121491 -1.101166 -1.272571  0.843459
1           1  0.873966 -1.162255  0.126186  0.505785
2           2  0.669515  0.483398 -0.438357 -0.139521
3           3  1.667877  0.906356  0.860304 -0.009414
   Unnamed: 0         0         1         2         3
4           4 -0.807549 -0.784813 -1.315516  0.687524
5           5 -0.157235  0.303400 -0.363407 -0.552651
6           6  0.414421  0.175171 -0.529516 -0.067457
7           7  1.058815 -0.117898 -1.853421 -0.701849
   Unnamed: 0         0         1         2         3
8           8  0.262396 -1.724596  0.276580  1.073024
```

```
9           9  0.635285 -2.178548  0.312044  1.572378
```

Specifying iterator=True will also return the TextFileReader object:

```
In [190]: reader = pd.read_csv('tmp.sv', sep='|', iterator=True)

In [191]: reader.get_chunk(5)
Out[191]: 
   Unnamed: 0         0         1         2         3
0           0  1.121491 -1.101166 -1.272571  0.843459
1           1  0.873966 -1.162255  0.126186  0.505785
2           2  0.669515  0.483398 -0.438357 -0.139521
3           3  1.667877  0.906356  0.860304 -0.009414
4           4 -0.807549 -0.784813 -1.315516  0.687524
```

Specifying the parser engine

Under the hood pandas uses a fast and efficient parser implemented in C as well as a Python implementation which is currently more feature-complete. Where possible pandas uses the C parser (specified as engine='c'), but may fall back to Python if C-unsupported options are specified. Currently, C-unsupported options include:

- sep other than a single character (e.g. regex separators)
- skipfooter
- sep=None with delim_whitespace=False

Specifying any of the above options will produce a ParserWarning unless the python engine is selected explicitly using engine='python'.

Reading remote files

You can pass in a URL to a CSV file:

```
df = pd.read_csv('https://download.bls.gov/pub/time.series/cu/cu.item',
                 sep='\t')
```

S3 URLs are handled as well but require installing the S3Fs library:

```
df = pd.read_csv('s3://pandas-test/tips.csv')
```

If your S3 bucket requires credentials you will need to set them as environment variables or in the ~/.aws/credentials config file, refer to the S3Fs documentation on credentials.

Writing out data

Writing to CSV format

The Series and DataFrame objects have an instance method to_csv which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- path_or_buf: A string path to the file to write or a file object. If a file object it must be opened with *newline*=
- sep : Field delimiter for the output file (default ,)
- na_rep: A string representation of a missing value (default)

- `float_format`: Format string for floating point numbers
- `columns`: Columns to write (default None)
- `header`: Whether to write out the column names (default True)
- `index`: whether to write row (index) names (default True)
- `index_label`: Column label(s) for index column(s) if desired. If None (default), and *header* and *index* are True, then the index names are used. (A sequence should be given if the `DataFrame` uses MultiIndex).
- `mode` : Python write mode, default w
- `encoding`: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- `line_terminator`: Character sequence denoting line end (default *os.linesep*)
- `quoting`: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a *float_format* then floats are converted to strings and csv.QUOTE_NONNUMERIC will treat them as non-numeric
- `quotechar`: Character used to quote fields (default)
- `doublequote`: Control quoting of `quotechar` in fields (default True)
- `escapechar`: Character used to escape `sep` and `quotechar` when appropriate (default None)
- `chunksize`: Number of rows to write at a time
- `date_format`: Format string for datetime objects

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.

4.1.2 JSON

Read and write `JSON` format files and strings.

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`, `table`}

  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` : string, type of date conversion, epoch for timestamp, iso for ISO8601.
- `double_precision` : The number of decimal places to use when encoding floating point values, default 10.
- `force_ascii` : force encoded string to be ASCII, default True.
- `date_unit` : The time unit to encode to, governs timestamp and ISO8601 precision. One of s, ms, us or ns for seconds, milliseconds, microseconds and nanoseconds respectively. Default ms.
- `default_handler` : The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.
- `lines` : If `records` orient, then will write each record per line as json.

Note `NaN`'s, `NaT`'s and `None` will be converted to `null` and `datetime` objects will be converted based on the `date_format` and `date_unit` parameters.

```
In [192]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [193]: json = dfj.to_json()

In [194]: json
Out[194]: '{"A":{"0":2.0027932898,"1":-1.1284197337,"2":-0.2671230751,"3":0.
↪0590811856,"4":0.2126018166},"B":{"0":0.009310115,"1":-1.2591311739,"2":1.
↪7549089729,"3":0.9464922966,"4":-0.5276761509}}'
```

Orient options

There are a number of different options for the format of the resulting JSON file / string. Consider the following `DataFrame` and `Series`:

```
In [195]: dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
   .....:                     columns=list('ABC'), index=list('xyz'))
   .....:

In [196]: dfjo
Out[196]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [197]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [198]: sjo
Out[198]:
x    15
y    16
z    17
Name: D, dtype: int64
```

Column oriented (the default for `DataFrame`) serializes the data as nested JSON objects with column labels acting as the primary index:

```
In [199]: dfjo.to_json(orient="columns")
Out[199]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

# Not available for Series
```

Index oriented (the default for `Series`) similar to column oriented but the index labels are now primary:

```
In [200]: dfjo.to_json(orient="index")
Out[200]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,
↪"C":9}}'

In [201]: sjo.to_json(orient="index")
Out[201]: '{"x":15,"y":16,"z":17}'
```

Record oriented serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing `DataFrame` data to plotting libraries, for example the JavaScript library `d3.js`:

```
In [202]: dfjo.to_json(orient="records")
Out[202]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [203]: sjo.to_json(orient="records")
Out[203]: '[15,16,17]'
```

Value oriented is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [204]: dfjo.to_json(orient="values")
Out[204]: '[[1,4,7],[2,5,8],[3,6,9]]'

# Not available for Series
```

Split oriented serializes to a JSON object containing separate entries for values, index and columns. Name is also included for `Series`:

```
In [205]: dfjo.to_json(orient="split")
Out[205]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,
↪5,8],[3,6,9]]}'

In [206]: sjo.to_json(orient="split")
Out[206]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

Table oriented serializes to the JSON Table Schema, allowing for the preservation of metadata including but not limited to dtypes and index names.

Note: Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the *split* option as it uses ordered containers.

Date handling

Writing in ISO date format:

```
In [207]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [208]: dfd['date'] = pd.Timestamp('20130101')

In [209]: dfd = dfd.sort_index(1, ascending=False)

In [210]: json = dfd.to_json(date_format='iso')

In [211]: json
Out[211]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":
↪"2013-01-01T00:00:00.000Z","3":"2013-01-01T00:00:00.000Z","4":"2013-01-01T00:00:00.
↪000Z"},"B":{"0":0.3903383957,"1":-0.5223681486,"2":2.0249145293,"3":2.1144885256,"4
↪":0.5337588359},"A":{"0":-1.0954121534,"1":-0.147141856,"2":0.6305826658,"3":1.
↪5730764249,"4":0.6200376615}}'
```

Writing in ISO date format, with microseconds:

```
In [212]: json = dfd.to_json(date_format='iso', date_unit='us')

In [213]: json
Out[213]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z
↪","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-
↪01T00:00:00.000000Z"},"B":{"0":0.3903383957,"1":-0.5223681486,"2":2.0249145293,"3
↪":2.1144885256,"4":0.5337588359},"A":{"0":-1.0954121534,"1":-0.147141856,"2":0.
↪6305826658,"3":1.5730764249,"4":0.6200376615}}'
```

Epoch timestamps, in seconds:

```
In [214]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [215]: json
Out[215]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4
↪":1356998400},"B":{"0":0.3903383957,"1":-0.5223681486,"2":2.0249145293,"3":2.
↪1144885256,"4":0.5337588359},"A":{"0":-1.0954121534,"1":-0.147141856,"2":0.
↪6305826658,"3":1.5730764249,"4":0.6200376615}}'
```

Writing to a file, with a date index and a date column:

```
In [216]: dfj2 = dfj.copy()

In [217]: dfj2['date'] = pd.Timestamp('20130101')

In [218]: dfj2['ints'] = list(range(5))

In [219]: dfj2['bools'] = True

In [220]: dfj2.index = pd.date_range('20130101', periods=5)

In [221]: dfj2.to_json('test.json')

In [222]: with open('test.json') as fh:
   .....:     print(fh.read())
   .....:
{"A":{"1356998400000":2.0027932898,"1357084800000":-1.1284197337,"1357171200000":-0.
↪2671230751,"1357257600000":0.0590811856,"1357344000000":0.2126018166},"B":{
↪"1356998400000":0.009310115,"1357084800000":-1.2591311739,"1357171200000":1.
↪7549089729,"1357257600000":0.9464922966,"1357344000000":-0.5276761509},"date":{
↪"1356998400000":1356998400000,"1357084800000":1356998400000,"1357171200000
↪":1356998400000,"1357257600000":1356998400000,"1357344000000":1356998400000},"ints":
↪{"1356998400000":0,"1357084800000":1,"1357171200000":2,"1357257600000":3,
↪"1357344000000":4},"bools":{"1356998400000":true,"1357084800000":true,"1357171200000
↪":true,"1357257600000":true,"1357344000000":true}}
```

Fallback behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. `np.complex`) then the `default_handler`, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a `toDict` method and call it. A `toDict` method should return a `dict` which will then be JSON serialized.
  - invoke the `default_handler` if one was provided.
  - convert the object to a `dict` by traversing its contents. However this will often fail with an `OverflowError` or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a `default_handler`. For example:

```
>>> DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json()  # raises
RuntimeError: Unhandled numpy dtype 15
```

can be dealt with by specifying a simple `default_handler`:

```
In [223]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[223]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a `DataFrame` if `typ` is not supplied or is `None`. To explicitly force `Series` parsing, pass `typ=series`

- `filepath_or_buffer` : a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, S3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json
- `typ` : type of object to recover (series or frame), default frame
- `orient` :

  Series :

  - default is `index`
  - allowed values are {`split`, `records`, `index`}

  DataFrame

  - default is `columns`
  - allowed values are {`split`, `records`, `index`, `columns`, `values`, `table`}

  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
`table`	adhering to the JSON Table Schema

- `dtype` : if True, infer dtypes, if a dict of column to dtype, then use those, if `False`, then dont infer dtypes at all, default is True, apply only to the data.
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is `True`
- `convert_dates` : a list of columns to parse for dates; If `True`, then try to parse date-like columns, default is `True`.
- `keep_default_dates` : boolean, default `True`. If parsing dates, then parse the default date-like columns.
- `numpy` : direct decoding to NumPy arrays. default is `False`; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.
- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality.
- `date_unit` : string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of s, ms, us or ns to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- `lines` : reads file as one json object per line.
- `encoding` : The encoding to use to decode py3 bytes.
- `chunksize` : when used in combination with `lines=True`, return a JsonReader which reads in `chunksize` lines per iteration.

The parser will raise one of ValueError/TypeError/AssertionError if the JSON is not parseable.

If a non-default orient was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see *Orient Options* for an overview.

Data conversion

The default of convert_axes=True, dtype=True, and convert_dates=True will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to dtype. convert_axes should only be set to False if you need to preserve string-like numbers (e.g. 1, 2) in an axes.

Note: Large integer values may be converted to dates if convert_dates=True and the data and / or column labels appear date-like. The exact threshold depends on the date_unit specified. date-like means that the column label meets one of the following criteria:

- it ends with '_at'
- it ends with '_time'
- it begins with 'timestamp'
- it is 'modified'
- it is 'date'

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 [224]: pd.read_json(json)
Out[224]:
        date         B         A
0 2013-01-01  0.390338 -1.095412
1 2013-01-01 -0.522368 -0.147142
2 2013-01-01  2.024915  0.630583
3 2013-01-01  2.114489  1.573076
4 2013-01-01  0.533759  0.620038
```

Reading from a file:

```
In [225]: pd.read_json('test.json')
Out[225]:
                   A         B       date  ints  bools
2013-01-01  2.002793  0.009310 2013-01-01     0   True
2013-01-02 -1.128420 -1.259131 2013-01-01     1   True
2013-01-03 -0.267123  1.754909 2013-01-01     2   True
```

(continues on next page)

(continued from previous page)

```
2013-01-04  0.059081  0.946492 2013-01-01     3   True
2013-01-05  0.212602 -0.527676 2013-01-01     4   True
```

Dont convert any data (but still convert axes and dates):

```
In [226]: pd.read_json('test.json', dtype=object).dtypes
Out[226]:
A        object
B        object
date     object
ints     object
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [227]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[227]:
A               float32
B               float64
date     datetime64[ns]
ints              int64
bools              int8
dtype: object
```

Preserve string indices:

```
In [228]: si = pd.DataFrame(np.zeros((4, 4)), columns=list(range(4)),
   .....:                   index=[str(i) for i in range(4)])
   .....:

In [229]: si
Out[229]:
     0    1    2    3
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0

In [230]: si.index
Out[230]: Index(['0', '1', '2', '3'], dtype='object')

In [231]: si.columns
Out[231]: Int64Index([0, 1, 2, 3], dtype='int64')

In [232]: json = si.to_json()

In [233]: sij = pd.read_json(json, convert_axes=False)

In [234]: sij
Out[234]:
   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 [235]: sij.index
Out[235]: Index(['0', '1', '2', '3'], dtype='object')

In [236]: sij.columns
Out[236]: Index(['0', '1', '2', '3'], dtype='object')
```

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [237]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as milliseconds -> Won't Work
In [238]: dfju = pd.read_json(json, date_unit='ms')

In [239]: dfju
Out[239]:
                            A         B                 date  ints  bools
1356998400000000000  2.002793  0.009310  1356998400000000000     0   True
1357084800000000000 -1.128420 -1.259131  1356998400000000000     1   True
1357171200000000000 -0.267123  1.754909  1356998400000000000     2   True
1357257600000000000  0.059081  0.946492  1356998400000000000     3   True
1357344000000000000  0.212602 -0.527676  1356998400000000000     4   True

# Let pandas detect the correct precision
In [240]: dfju = pd.read_json(json)

In [241]: dfju
Out[241]:
                   A         B       date  ints  bools
2013-01-01  2.002793  0.009310 2013-01-01     0   True
2013-01-02 -1.128420 -1.259131 2013-01-01     1   True
2013-01-03 -0.267123  1.754909 2013-01-01     2   True
2013-01-04  0.059081  0.946492 2013-01-01     3   True
2013-01-05  0.212602 -0.527676 2013-01-01     4   True

# Or specify that all timestamps are in nanoseconds
In [242]: dfju = pd.read_json(json, date_unit='ns')

In [243]: dfju
Out[243]:
                   A         B       date  ints  bools
2013-01-01  2.002793  0.009310 2013-01-01     0   True
2013-01-02 -1.128420 -1.259131 2013-01-01     1   True
2013-01-03 -0.267123  1.754909 2013-01-01     2   True
2013-01-04  0.059081  0.946492 2013-01-01     3   True
2013-01-05  0.212602 -0.527676 2013-01-01     4   True
```

The Numpy parameter

Note: This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

If `numpy=True` is passed to `read_json` an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```
In [244]: randfloats = np.random.uniform(-100, 1000, 10000)

In [245]: randfloats.shape = (1000, 10)

In [246]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [247]: jsonfloats = dffloats.to_json()
```

```
In [248]: %timeit pd.read_json(jsonfloats)
9.45 ms +- 266 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [249]: %timeit pd.read_json(jsonfloats, numpy=True)
7.21 ms +- 356 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [250]: jsonfloats = dffloats.head(100).to_json()
```

```
In [251]: %timeit pd.read_json(jsonfloats)
7.05 ms +- 307 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [252]: %timeit pd.read_json(jsonfloats, numpy=True)
5.68 ms +- 136 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Warning: Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:

- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A ValueError may be raised, or incorrect output may be produced if this condition is not satisfied.
- labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using to_json but may not be the case if the JSON is from another source.

Normalization

pandas provides a utility function to take a dict or list of dicts and *normalize* this semi-structured data into a flat table.

```
In [253]: from pandas.io.json import json_normalize

In [254]: data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
   .....:          {'name': {'given': 'Mose', 'family': 'Regner'}},
   .....:          {'id': 2, 'name': 'Faye Raker'}]
   .....:

In [255]: json_normalize(data)
Out[255]:
    id name.first name.last name.given name.family        name
0  1.0     Coleen      Volk        NaN         NaN         NaN
```

(continues on next page)

(continued from previous page)

```
1  NaN        NaN       NaN       Mose      Regner         NaN
2  2.0        NaN       NaN        NaN         NaN  Faye Raker
```

```
In [256]: data = [{'state': 'Florida',
   .....:          'shortname': 'FL',
   .....:          'info': {'governor': 'Rick Scott'},
   .....:          'counties': [{'name': 'Dade', 'population': 12345},
   .....:                       {'name': 'Broward', 'population': 40000},
   .....:                       {'name': 'Palm Beach', 'population': 60000}]},
   .....:         {'state': 'Ohio',
   .....:          'shortname': 'OH',
   .....:          'info': {'governor': 'John Kasich'},
   .....:          'counties': [{'name': 'Summit', 'population': 1234},
   .....:                       {'name': 'Cuyahoga', 'population': 1337}]}]
   .....:

In [257]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor
↪']])
Out[257]: 
         name  population    state shortname info.governor
0        Dade       12345  Florida        FL    Rick Scott
1     Broward       40000  Florida        FL    Rick Scott
2  Palm Beach       60000  Florida        FL    Rick Scott
3      Summit        1234     Ohio        OH   John Kasich
4    Cuyahoga        1337     Ohio        OH   John Kasich
```

The max_level parameter provides more control over which level to end normalization. With max_level=1 the following snippet normalizes until 1st nesting level of the provided dict.

```
In [258]: data = [{'CreatedBy': {'Name': 'User001'},
   .....:          'Lookup': {'TextField': 'Some text',
   .....:                     'UserField': {'Id': 'ID001',
   .....:                                   'Name': 'Name001'}},
   .....:          'Image': {'a': 'b'}
   .....:          }]
   .....:

In [259]: json_normalize(data, max_level=1)
Out[259]: 
  CreatedBy.Name Lookup.TextField                    Lookup.UserField Image.a
0        User001        Some text  {'Id': 'ID001', 'Name': 'Name001'}       b
```

Line delimited json

New in version 0.19.0.

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

New in version 0.21.0.

For line-delimited json files, pandas can also return an iterator which reads in `chunksize` lines at a time. This can be useful for large files or to read from a stream.

```
In [260]: jsonl = '''
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: '''
   .....:

In [261]: df = pd.read_json(jsonl, lines=True)

In [262]: df
Out[262]:
   a  b
0  1  2
1  3  4

In [263]: df.to_json(orient='records', lines=True)
Out[263]: '{"a":1,"b":2}\n{"a":3,"b":4}'

# reader is an iterator that returns `chunksize` lines each iteration
In [264]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [265]: reader
Out[265]: <pandas.io.json._json.JsonReader at 0x1c336c2c10>

In [266]: for chunk in reader:
   .....:     print(chunk)
   .....:
Empty DataFrame
Columns: []
Index: []
   a  b
0  1  2
   a  b
1  3  4
```

Table schema

New in version 0.20.0.

Table Schema is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the orient `table` to build a JSON string with two fields, `schema` and `data`.

```
In [267]: df = pd.DataFrame({'A': [1, 2, 3],
   .....:                    'B': ['a', 'b', 'c'],
   .....:                    'C': pd.date_range('2016-01-01', freq='d',␣
↪periods=3)},
   .....:                   index=pd.Index(range(3), name='idx'))
   .....:

In [268]: df
Out[268]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03
```

```
In [269]: df.to_json(orient='table', date_format="iso")
Out[269]: '{"schema": {"fields":[{"name":"idx","type":"integer"},
↪{"name":"A","type":"integer"},{"name":"B","type":"string"},{"name":"C",
↪"type":"datetime"}],"primaryKey":["idx"],"pandas_version":"0.20.0"},␣
↪"data": [{"idx":0,"A":1,"B":"a","C":"2016-01-01T00:00:00.000Z"},{"idx":1,
↪"A":2,"B":"b","C":"2016-01-02T00:00:00.000Z"},{"idx":2,"A":3,"B":"c",
↪"C":"2016-01-03T00:00:00.000Z"}]}'
```

The schema field contains the fields key, which itself contains a list of column name to type pairs, including the Index or MultiIndex (see below for a list of types). The schema field also contains a primaryKey field if the (Multi)index is unique.

The second field, data, contains the serialized data with the records orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:

Pandas type	Table Schema type
int64	integer
float64	number
bool	boolean
datetime64[ns]	datetime
timedelta64[ns]	duration
categorical	any
object	str

A few notes on the generated table schema:

- The schema object contains a pandas_version field. This contains the version of pandas dialect of the schema, and will be incremented with each revision.
- All dates are converted to UTC when serializing. Even timezone naive values, which are treated as UTC with an offset of 0.

```
In [270]: from pandas.io.json import build_table_schema

In [271]: s = pd.Series(pd.date_range('2016', periods=4))

In [272]: build_table_schema(s)
Out[272]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- datetimes with a timezone (before serializing), include an additional field tz with the time zone name (e.g. 'US/Central').

```
In [273]: s_tz = pd.Series(pd.date_range('2016', periods=12,
   .....:                                tz='US/Central'))
   .....:

In [274]: build_table_schema(s_tz)
Out[274]:
{'fields': [{'name': 'index', 'type': 'integer'},
```

(continues on next page)

(continued from previous page)

```
  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain and additional field `freq` with the periods frequency, e.g. `'A-DEC'`.

```
In [275]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC',
   .....:                                            periods=4))
   .....:

In [276]: build_table_schema(s_per)
Out[276]:
{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Categoricals use the `any` type and an `enum` constraint listing the set of possible values. Additionally, an `ordered` field is included:

```
In [277]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [278]: build_table_schema(s_cat)
Out[278]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values',
   'type': 'any',
   'constraints': {'enum': ['a', 'b']},
   'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- A `primaryKey` field, containing an array of labels, is included *if the index is unique*:

```
In [279]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [280]: build_table_schema(s_dupe)
Out[280]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0'}
```

- The `primaryKey` behavior is the same with MultiIndexes, but in this case the `primaryKey` is an array:

```
In [281]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
   .....:                                                          (0, 1)]))
   .....:

In [282]: build_table_schema(s_multi)
Out[282]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '0.20.0'}
```

- The default naming roughly follows these rules:
  - For series, the `object.name` is used. If thats none, then the name is `values`
  - For `DataFrames`, the stringified version of the column name is used
  - For `Index` (not `MultiIndex`), `index.name` is used, with a fallback to `index` if that is None.
  - For `MultiIndex`, `mi.names` is used. If any level has no name, then `level_<i>` is used.

New in version 0.23.0.

`read_json` also accepts `orient='table'` as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner.

```
In [283]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
   .....:                    'bar': ['a', 'b', 'c', 'd'],
   .....:                    'baz': pd.date_range('2018-01-01', freq='d', periods=4),
   .....:                    'qux': pd.Categorical(['a', 'b', 'c', 'c'])
   .....:                    }, index=pd.Index(range(4), name='idx'))
   .....:

In [284]: df
Out[284]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

In [285]: df.dtypes
Out[285]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object

In [286]: df.to_json('test.json', orient='table')

In [287]: new_df = pd.read_json('test.json', orient='table')

In [288]: new_df
Out[288]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

In [289]: new_df.dtypes
Out[289]:
foo             int64
bar            object
baz    datetime64[ns]
```

```
qux            category
dtype: object
```

Please note that the literal string index as the name of an `Index` is not round-trippable, nor are any names beginning with `'level_'` within a `MultiIndex`. These are used by default in `DataFrame.to_json()` to indicate missing values and the subsequent read cannot distinguish the intent.

```
In [290]: df.index.name = 'index'

In [291]: df.to_json('test.json', orient='table')

In [292]: new_df = pd.read_json('test.json', orient='table')

In [293]: print(new_df.index.name)
None
```

4.1.3 HTML

Reading HTML content

Warning: We **highly encourage** you to read the *HTML Table Parsing gotchas* below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level `read_html()` function can accept an HTML string/file/URL and will parse HTML tables into list of pandas `DataFrames`. Lets 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 [294]: url = 'https://www.fdic.gov/bank/individual/failed/banklist.html'

In [295]: dfs = pd.read_html(url)

In [296]: dfs
Out[296]:
[                               Bank Name        City  ...       Closing Date      ␣
↪Updated Date
 0       City National Bank of New Jersey      Newark  ...   November 1, 2019  ␣
↪November 1, 2019
 1                          Resolute Bank      Maumee  ...   October 25, 2019  ␣
↪October 25, 2019
 2                  Louisa Community Bank      Louisa  ...   October 25, 2019  ␣
↪October 28, 2019
 3                   The Enloe State Bank      Cooper  ...       May 31, 2019   ␣
↪August 22, 2019
 4    Washington Federal Bank for Savings     Chicago  ...  December 15, 2017     ␣
↪July 24, 2019
 ..                                   ...         ...  ...                ...        ␣
↪       ...
 554                  Superior Bank, FSB    Hinsdale  ...      July 27, 2001   ␣
↪August 19, 2014
```

(continues on next page)

(continued from previous page)

```
555                    Malta National Bank       Malta  ...        May 3, 2001 ␣
↪November 18, 2002
556       First Alliance Bank & Trust Co.  Manchester  ...   February 2, 2001 ␣
↪February 18, 2003
557     National State Bank of Metropolis  Metropolis  ...  December 14, 2000    ␣
↪March 17, 2005
558                      Bank of Honolulu    Honolulu  ...   October 13, 2000    ␣
↪March 17, 2005

[559 rows x 7 columns]]
```

Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string:

```
In [297]: with open(file_path, 'r') as f:
   .....:     dfs = pd.read_html(f.read())
   .....:

In [298]: dfs
Out[298]: 
[                                    Bank Name          City  ...       Closing Date ␣
↪      Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  ...       May 31, 2013 ␣
↪      May 31, 2013
 1                        Central Arizona Bank    Scottsdale  ...       May 14, 2013 ␣
↪      May 20, 2013
 2                                Sunrise Bank      Valdosta  ...       May 10, 2013 ␣
↪      May 21, 2013
 3                       Pisgah Community Bank     Asheville  ...       May 10, 2013 ␣
↪      May 14, 2013
 4                         Douglas County Bank  Douglasville  ...     April 26, 2013 ␣
↪      May 16, 2013
 ..                                        ...           ...  ...                ... ␣
↪               ...
 500                        Superior Bank, FSB      Hinsdale  ...      July 27, 2001 ␣
↪      June 5, 2012
 501                       Malta National Bank         Malta  ...        May 3, 2001 ␣
↪November 18, 2002
 502           First Alliance Bank & Trust Co.    Manchester  ...   February 2, 2001 ␣
↪February 18, 2003
 503         National State Bank of Metropolis    Metropolis  ...  December 14, 2000 ␣
↪    March 17, 2005
 504                          Bank of Honolulu      Honolulu  ...   October 13, 2000 ␣
↪    March 17, 2005

[505 rows x 7 columns]]
```

You can even pass in an instance of `StringIO` if you so desire:

```
In [299]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:
```

(continues on next page)

(continued from previous page)

```
In [300]: dfs = pd.read_html(sio)

In [301]: dfs
Out[301]:
[                                    Bank Name          City  ...       Closing Date ␣
↪      Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  ...       May 31, 2013 ␣
↪      May 31, 2013
 1                        Central Arizona Bank    Scottsdale  ...       May 14, 2013 ␣
↪      May 20, 2013
 2                                Sunrise Bank      Valdosta  ...       May 10, 2013 ␣
↪      May 21, 2013
 3                       Pisgah Community Bank     Asheville  ...       May 10, 2013 ␣
↪      May 14, 2013
 4                         Douglas County Bank  Douglasville  ...     April 26, 2013 ␣
↪      May 16, 2013
 ..                                        ...           ...  ...                ... ␣
↪               ...
 500                        Superior Bank, FSB      Hinsdale  ...      July 27, 2001 ␣
↪      June 5, 2012
 501                       Malta National Bank         Malta  ...        May 3, 2001 ␣
↪November 18, 2002
 502           First Alliance Bank & Trust Co.    Manchester  ...   February 2, 2001 ␣
↪February 18, 2003
 503         National State Bank of Metropolis    Metropolis  ...  December 14, 2000 ␣
↪    March 17, 2005
 504                          Bank of Honolulu      Honolulu  ...   October 13, 2000 ␣
↪    March 17, 2005

 [505 rows x 7 columns]]
```

Note: The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesnt 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 = pd.read_html(url, match=match)
```

Specify a header row (by default `<th>` or `<td>` elements located within a `<thead>` are used to form the column index, if multiple rows are contained within `<thead>` then a MultiIndex is created); if specified, the header row is taken from the data minus the parsed header elements (`<th>` elements).

```
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```
dfs = pd.read_html(url, index_col=0)
```

Specify a number of rows to skip:

```
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (`xrange` (Python 2 only) works as well):

```
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```
dfs1 = pd.read_html(url, attrs={'id': 'table'})
dfs2 = pd.read_html(url, attrs={'class': 'sortable'})
print(np.array_equal(dfs1[0], dfs2[0]))  # Should be True
```

Specify values that should be converted to NaN:

```
dfs = pd.read_html(url, na_values=['No Acquirer'])
```

New in version 0.19.

Specify whether to keep the default set of NaN values:

```
dfs = pd.read_html(url, keep_default_na=False)
```

New in version 0.19.

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```
url_mcc = 'https://en.wikipedia.org/wiki/Mobile_country_code'
dfs = pd.read_html(url_mcc, match='Telekom Albania', header=0,
                   converters={'MNC': str})
```

New in version 0.19.

Use some combination of the above:

```
dfs = pd.read_html(url, match='Metcalf Bank', index_col=0)
```

Read in pandas `to_html` output (with some loss of floating point precision):

```
df = pd.DataFrame(np.random.randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = pd.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. You may use:

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])
```

Or you could pass `flavor='lxml'` without a list:

```
dfs = pd.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 = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])
```

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 brevitys sake. See to_html() for the full set of options.

```
In [302]: df = pd.DataFrame(np.random.randn(2, 2))

In [303]: df
Out[303]:
          0         1
0 -1.050304  1.131622
1 -0.692581 -1.174172

In [304]: 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>-1.050304</td>
      <td>1.131622</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.692581</td>
      <td>-1.174172</td>
    </tr>
  </tbody>
</table>
```

HTML:

The columns argument will limit the columns shown:

```
In [305]: 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>
```

(continues on next page)

(continued from previous page)

```
      <td>-1.050304</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.692581</td>
    </tr>
  </tbody>
</table>
```

HTML:

`float_format` takes a Python callable to control the precision of floating point values:

```
In [306]: 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>-1.0503044154</td>
      <td>1.1316218324</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.6925807265</td>
      <td>-1.1741715747</td>
    </tr>
  </tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off:

```
In [307]: 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>-1.050304</td>
      <td>1.131622</td>
    </tr>
    <tr>
      <td>1</td>
```

(continues on next page)

(continued from previous page)

```
      <td>-0.692581</td>
      <td>-1.174172</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 [308]: 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>-1.050304</td>
      <td>1.131622</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.692581</td>
      <td>-1.174172</td>
    </tr>
  </tbody>
</table>
```

The `render_links` argument provides the ability to add hyperlinks to cells that contain URLs.

New in version 0.24.

```
In [309]: url_df = pd.DataFrame({
   .....:     'name': ['Python', 'Pandas'],
   .....:     'url': ['https://www.python.org/', 'http://pandas.pydata.org']})
   .....:

In [310]: print(url_df.to_html(render_links=True))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>url</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Python</td>
      <td><a href="https://www.python.org/" target="_blank">https://www.python.org/</
↪a></td>
```

(continues on next page)

(continued from previous page)

```
    </tr>
    <tr>
      <th>1</th>
      <td>Pandas</td>
      <td><a href="http://pandas.pydata.org" target="_blank">http://pandas.pydata.org
↪</a></td>
    </tr>
  </tbody>
</table>
```

HTML:

Finally, the `escape` argument allows you to control whether the <, > and & characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

```
In [311]: df = pd.DataFrame({'a': list('&<>'), 'b': np.random.randn(3)})
```

Escaped:

```
In [312]: 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>1.254800</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&lt;</td>
      <td>1.131996</td>
    </tr>
    <tr>
      <th>2</th>
      <td>&gt;</td>
      <td>-1.311021</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```
In [313]: 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>
```

(continues on next page)

(continued from previous page)

```
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&</td>
      <td>1.254800</td>
    </tr>
    <tr>
      <th>1</th>
      <td><</td>
      <td>1.131996</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>-1.311021</td>
    </tr>
  </tbody>
</table>
```

Note: Some browsers may not show a difference in the rendering of the previous two HTML tables.

HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function `read_html`.

Issues with lxml

- Benefits
  - **lxml** is very fast.
  - **lxml** requires Cython to install correctly.
- Drawbacks
  - **lxml** does *not* make any guarantees about the results of its parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the **lxml** backend, but **this backend will use html5lib** if **lxml** fails to parse
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if **lxml** fails.

Issues with BeautifulSoup4 using lxml as a backend

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

Issues with BeautifulSoup4 using html5lib as a backend

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is correct, since the process of fixing markup does not have a single definition.

  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.

- Drawbacks

  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

4.1.4 Excel files

The `read_excel()` method can read Excel 2003 (`.xls`) files using the `xlrd` Python module. Excel 2007+ (`.xlsx`) files can be read using either `xlrd` or `openpyxl`. The `to_excel()` instance method is used for saving a `DataFrame` to Excel. Generally the semantics are similar to working with *csv* data. See the *cookbook* for some advanced strategies.

Reading Excel files

In the most basic use-case, `read_excel` takes a path to an Excel file, and the `sheet_name` indicating which sheet to parse.

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', sheet_name='Sheet1')
```

`ExcelFile` class

To facilitate working with multiple sheets from the same file, the `ExcelFile` class can be used to wrap the file and can be passed into `read_excel` There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```
xlsx = pd.ExcelFile('path_to_file.xls')
df = pd.read_excel(xlsx, 'Sheet1')
```

The `ExcelFile` class can also be used as a context manager.

```
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

The `sheet_names` property will generate a list of the sheet names in the file.

The primary use-case for an `ExcelFile` is parsing multiple sheets with different parameters:

```
data = {}
# For when Sheet1's format differs from Sheet2
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to `read_excel` with no loss in performance.

```python
# using the ExcelFile class
data = {}
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=None,
                                   na_values=['NA'])

# equivalent using the read_excel function
data = pd.read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'],
                     index_col=None, na_values=['NA'])
```

`ExcelFile` can also be called with a `xlrd.book.Book` object as a parameter. This allows the user to control how the excel file is read. For example, sheets can be loaded on demand by calling `xlrd.open_workbook()` with `on_demand=True`.

```python
import xlrd
xlrd_book = xlrd.open_workbook('path_to_file.xls', on_demand=True)
with pd.ExcelFile(xlrd_book) as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

Specifying sheets

Note: The second argument is `sheet_name`, not to be confused with `ExcelFile.sheet_names`.

Note: An ExcelFiles attribute `sheet_names` provides access to a list of sheets.

- The arguments `sheet_name` allows specifying the sheet or sheets to read.
- The default value for `sheet_name` is 0, indicating to read the first sheet
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.
- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Pass a `None` to return a dictionary of all available sheets.

```python
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```python
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

Using all default values:

```python
# Returns a DataFrame
pd.read_excel('path_to_file.xls')
```

Using None to get all sheets:

```
# Returns a dictionary of DataFrames
pd.read_excel('path_to_file.xls', sheet_name=None)
```

Using a list to get multiple sheets:

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

read_excel can read more than one sheet, by setting sheet_name to either a list of sheet names, a list of sheet positions, or None to read all sheets. Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

Reading a MultiIndex

read_excel can read a MultiIndex index, by passing a list of columns to index_col and a MultiIndex column by passing a list of rows to header. If either the index or columns have serialized level names those will be read in as well by specifying the rows/columns that make up the levels.

For example, to read in a MultiIndex index without names:

```
In [314]: df = pd.DataFrame({'a': [1, 2, 3, 4], 'b': [5, 6, 7, 8]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b'], ['c', 'd
↪']]))
   .....:

In [315]: df.to_excel('path_to_file.xlsx')

In [316]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [317]: df
Out[317]:
     a  b
a c  1  5
  d  2  6
b c  3  7
  d  4  8
```

If the index has level names, they will parsed as well, using the same parameters.

```
In [318]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [319]: df.to_excel('path_to_file.xlsx')

In [320]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [321]: df
Out[321]:
           a  b
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

If the source file has both MultiIndex index and columns, lists specifying each should be passed to index_col and header:

```
In [322]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']],
   .....:                                          names=['c1', 'c2'])
   .....:

In [323]: df.to_excel('path_to_file.xlsx')

In [324]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1], header=[0, 1])

In [325]: df
Out[325]:
c1           a
c2           b  d
lvl1 lvl2
a    c       1  5
     d       2  6
b    c       3  7
     d       4  8
```

Parsing specific columns

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. `read_excel` takes a `usecols` keyword to allow you to specify a subset of columns to parse.

Deprecated since version 0.24.0.

Passing in an integer for `usecols` has been deprecated. Please pass in a list of ints from 0 to `usecols` inclusive instead.

If `usecols` is an integer, then it is assumed to indicate the last column to be parsed.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=2)
```

You can also specify a comma-delimited set of Excel columns and ranges as a string:

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols='A,C:E')
```

If `usecols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=[0, 2, 3])
```

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`.

New in version 0.24.

If `usecols` is a list of strings, it is assumed that each string corresponds to a column name provided either by the user in `names` or inferred from the document header row(s). Those strings define which columns will be parsed:

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=['foo', 'bar'])
```

Element order is ignored, so `usecols=['baz', 'joe']` is the same as `['joe', 'baz']`.

New in version 0.24.

If `usecols` is callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=lambda x: x.isalpha())
```

Parsing dates

Datetime-like values are normally automatically converted to the appropriate dtype when reading the excel file. But if you have a column of strings that *look* like dates (but are not actually formatted as dates in excel), you can use the `parse_dates` keyword to parse those strings to datetimes:

```
pd.read_excel('path_to_file.xls', 'Sheet1', parse_dates=['date_strings'])
```

Cell converters

It is possible to transform the contents of Excel cells via the `converters` option. For instance, to convert a column to boolean:

```
pd.read_excel('path_to_file.xls', 'Sheet1', converters={'MyBools': bool})
```

This options handles missing values and treats exceptions in the converters as missing data. Transformations are applied cell by cell rather than to the column as a whole, so the array dtype is not guaranteed. For instance, a column of integers with missing values cannot be transformed to an array with integer dtype, because NaN is strictly a float. You can manually mask missing data to recover integer dtype:

```
def cfun(x):
    return int(x) if x else -1


pd.read_excel('path_to_file.xls', 'Sheet1', converters={'MyInts': cfun})
```

Dtype specifications

New in version 0.20.

As an alternative to converters, the type for an entire column can be specified using the *dtype* keyword, which takes a dictionary mapping column names to types. To interpret data with no type inference, use the type `str` or `object`.

```
pd.read_excel('path_to_file.xls', dtype={'MyInts': 'int64', 'MyText': str})
```

Writing Excel files

Writing Excel files to disk

To write a `DataFrame` object to a sheet of an Excel file, you can use the `to_excel` instance method. The arguments are largely the same as `to_csv` described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the `DataFrame` should be written. For example:

```
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Files with a `.xls` extension will be written using `xlwt` and those with a `.xlsx` extension will be written using `xlsxwriter` (if available) or `openpyxl`.

The `DataFrame` will be written in a way that tries to mimic the REPL output. The `index_label` will be placed in the second row instead of the first. You can place it in the first row by setting the `merge_cells` option in `to_excel()` to `False`:

```
df.to_excel('path_to_file.xlsx', index_label='label', merge_cells=False)
```

In order to write separate DataFrames to separate sheets in a single Excel file, one can pass an ExcelWriter.

```
with pd.ExcelWriter('path_to_file.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

Note: Wringing a little more performance out of `read_excel` Internally, Excel stores all numeric data as floats. Because this can produce unexpected behavior when reading in data, pandas defaults to trying to convert integers to floats if it doesnt lose information (`1.0 --> 1`). You can pass `convert_float=False` to disable this behavior, which may give a slight performance improvement.

Writing Excel files to memory

Pandas supports writing Excel files to buffer-like objects such as `StringIO` or `BytesIO` using `ExcelWriter`.

```
# Safe import for either Python 2.x or 3.x
try:
    from io import BytesIO
except ImportError:
    from cStringIO import StringIO as BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter(bio, engine='xlsxwriter')
df.to_excel(writer, sheet_name='Sheet1')

# Save the workbook
writer.save()

# Seek to the beginning and read to copy the workbook to a variable in memory
bio.seek(0)
workbook = bio.read()
```

Note: `engine` is optional but recommended. Setting the engine determines the version of workbook produced. Setting `engine='xlrd'` will produce an Excel 2003-format workbook (xls). Using either `'openpyxl'` or `'xlsxwriter'` will produce an Excel 2007-format workbook (xlsx). If omitted, an Excel 2007-formatted workbook is produced.

Excel writer engines

Pandas chooses an Excel writer via two methods:

1. the `engine` keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the XlsxWriter for `.xlsx`, openpyxl for `.xlsm`, and xlwt for `.xls` files. If you have multiple engines installed, you can set the default engine through *setting the config options* `io.excel.xlsx.writer` and `io.excel.xls.writer`. pandas will fall back on openpyxl for `.xlsx` files if Xlsxwriter is not available.

To specify which writer you want to use, you can pass an engine keyword argument to to_excel and to ExcelWriter. The built-in engines are:

- openpyxl: version 2.4 or higher is required
- xlsxwriter
- xlwt

```
# By setting the 'engine' in the DataFrame 'to_excel()' methods.
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='xlsxwriter')

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter('path_to_file.xlsx', engine='xlsxwriter')

# Or via pandas configuration.
from pandas import options                                       # noqa: E402
options.io.excel.xlsx.writer = 'xlsxwriter'

df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Style and formatting

The look and feel of Excel worksheets created from pandas can be modified using the following parameters on the DataFrames to_excel method.

- float_format : Format string for floating point numbers (default None).
- freeze_panes : A tuple of two integers representing the bottommost row and rightmost column to freeze. Each of these parameters is one-based, so (1, 1) will freeze the first row and first column (default None).

Using the Xlsxwriter engine provides many options for controlling the format of an Excel worksheet created with the to_excel method. Excellent examples can be found in the Xlsxwriter documentation here: https://xlsxwriter.readthedocs.io/working_with_pandas.html

4.1.5 OpenDocument Spreadsheets

New in version 0.25.

The *read_excel()* method can also read OpenDocument spreadsheets using the odfpy module. The semantics and features for reading OpenDocument spreadsheets match what can be done for *Excel files* using engine='odf'.

```
# Returns a DataFrame
pd.read_excel('path_to_file.ods', engine='odf')
```

Note: Currently pandas only supports *reading* OpenDocument spreadsheets. Writing is not implemented.

4.1.6 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_csv 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()
>>> 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.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [4, 5, 6],
...                    'C': ['p', 'q', 'r']},
...                   index=['x', 'y', 'z'])
>>> df
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
>>> df.to_clipboard()
>>> pd.read_clipboard()
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

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 PyQt5, PyQt4 or qtpy) on Linux to use these methods.

4.1.7 Pickling

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

```
In [326]: df
Out[326]:
c1          a
c2          b  d
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8

In [327]: 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 [328]: pd.read_pickle('foo.pkl')
Out[328]:
c1          a
c2          b  d
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8
```

Warning: Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3/library/pickle.html

Warning: *read_pickle()* is only guaranteed backwards compatible back to pandas version 0.20.3

Compressed pickle files

New in version 0.20.0.

read_pickle(), *DataFrame.to_pickle()* and *Series.to_pickle()* can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz` are supported for reading and writing. The `zip` file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If infer, then use `gzip`, `bz2`, `zip`, or `xz` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, or `'.xz'`, respectively.

```
In [329]: df = pd.DataFrame({
   .....:     'A': np.random.randn(1000),
   .....:     'B': 'foo',
   .....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
   .....:

In [330]: df
Out[330]:
            A    B                   C
0   -0.053113  foo 2013-01-01 00:00:00
1    0.348832  foo 2013-01-01 00:00:01
2   -0.162729  foo 2013-01-01 00:00:02
3   -1.269943  foo 2013-01-01 00:00:03
4   -0.481824  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -1.001718  foo 2013-01-01 00:16:35
996 -0.471336  foo 2013-01-01 00:16:36
997 -0.071712  foo 2013-01-01 00:16:37
998  0.578273  foo 2013-01-01 00:16:38
999  0.595708  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type:

```
In [331]: df.to_pickle("data.pkl.compress", compression="gzip")

In [332]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [333]: rt
Out[333]:
            A    B                   C
0   -0.053113  foo 2013-01-01 00:00:00
1    0.348832  foo 2013-01-01 00:00:01
2   -0.162729  foo 2013-01-01 00:00:02
3   -1.269943  foo 2013-01-01 00:00:03
4   -0.481824  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -1.001718  foo 2013-01-01 00:16:35
996 -0.471336  foo 2013-01-01 00:16:36
997 -0.071712  foo 2013-01-01 00:16:37
998  0.578273  foo 2013-01-01 00:16:38
999  0.595708  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension:

```
In [334]: df.to_pickle("data.pkl.xz", compression="infer")

In [335]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [336]: rt
Out[336]:
            A    B                   C
0   -0.053113  foo 2013-01-01 00:00:00
1    0.348832  foo 2013-01-01 00:00:01
2   -0.162729  foo 2013-01-01 00:00:02
3   -1.269943  foo 2013-01-01 00:00:03
4   -0.481824  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -1.001718  foo 2013-01-01 00:16:35
996 -0.471336  foo 2013-01-01 00:16:36
997 -0.071712  foo 2013-01-01 00:16:37
998  0.578273  foo 2013-01-01 00:16:38
999  0.595708  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

The default is to infer:

```
In [337]: df.to_pickle("data.pkl.gz")

In [338]: rt = pd.read_pickle("data.pkl.gz")

In [339]: rt
Out[339]:
            A    B                   C
0   -0.053113  foo 2013-01-01 00:00:00
1    0.348832  foo 2013-01-01 00:00:01
2   -0.162729  foo 2013-01-01 00:00:02
3   -1.269943  foo 2013-01-01 00:00:03
```

(continues on next page)

(continued from previous page)

```
4   -0.481824  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -1.001718  foo 2013-01-01 00:16:35
996 -0.471336  foo 2013-01-01 00:16:36
997 -0.071712  foo 2013-01-01 00:16:37
998  0.578273  foo 2013-01-01 00:16:38
999  0.595708  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [340]: df["A"].to_pickle("s1.pkl.bz2")

In [341]: rt = pd.read_pickle("s1.pkl.bz2")

In [342]: rt
Out[342]:
0      -0.053113
1       0.348832
2      -0.162729
3      -1.269943
4      -0.481824
         ...
995    -1.001718
996    -0.471336
997    -0.071712
998     0.578273
999     0.595708
Name: A, Length: 1000, dtype: float64
```

4.1.8 msgpack

pandas supports the `msgpack` format for object serialization. This is a lightweight portable binary format, similar to binary JSON, that is highly space efficient, and provides good performance both on the writing (serialization), and reading (deserialization).

> **Warning:** The msgpack format is deprecated as of 0.25 and will be removed in a future version. It is recommended to use pyarrow for on-the-wire transmission of pandas objects.

> **Warning:** *read_msgpack()* is only guaranteed backwards compatible back to pandas version 0.20.3

```
In [343]: df = pd.DataFrame(np.random.rand(5, 2), columns=list('AB'))

In [344]: df.to_msgpack('foo.msg')

In [345]: pd.read_msgpack('foo.msg')
Out[345]:
          A         B
0  0.541029  0.554672
1  0.150831  0.503287
2  0.834267  0.881894
```

(continues on next page)

(continued from previous page)

```
3  0.706066  0.726912
4  0.639300  0.067928

In [346]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [347]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1, 2, 3]), s)

In [348]: pd.read_msgpack('foo.msg')
Out[348]: 
[          A         B
 0  0.541029  0.554672
 1  0.150831  0.503287
 2  0.834267  0.881894
 3  0.706066  0.726912
 4  0.639300  0.067928, 'foo', array([1, 2, 3]), 2013-01-01    0.753932
 2013-01-02    0.676180
 2013-01-03    0.924728
 2013-01-04    0.338661
 2013-01-05    0.592241
 Freq: D, dtype: float64]
```

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

```
In [349]: for o in pd.read_msgpack('foo.msg', iterator=True):
   .....:     print(o)
   .....:
          A         B
0  0.541029  0.554672
1  0.150831  0.503287
2  0.834267  0.881894
3  0.706066  0.726912
4  0.639300  0.067928
foo
[1 2 3]
2013-01-01    0.753932
2013-01-02    0.676180
2013-01-03    0.924728
2013-01-04    0.338661
2013-01-05    0.592241
Freq: D, dtype: float64
```

You can pass `append=True` to the writer to append to an existing pack:

```
In [350]: df.to_msgpack('foo.msg', append=True)

In [351]: pd.read_msgpack('foo.msg')
Out[351]: 
[          A         B
 0  0.541029  0.554672
 1  0.150831  0.503287
 2  0.834267  0.881894
 3  0.706066  0.726912
 4  0.639300  0.067928, 'foo', array([1, 2, 3]), 2013-01-01    0.753932
 2013-01-02    0.676180
 2013-01-03    0.924728
```

(continues on next page)

(continued from previous page)

```
 2013-01-04    0.338661
 2013-01-05    0.592241
 Freq: D, dtype: float64,           A         B
 0  0.541029  0.554672
 1  0.150831  0.503287
 2  0.834267  0.881894
 3  0.706066  0.726912
 4  0.639300  0.067928]
```

Unlike other io methods, `to_msgpack` is available on both a per-object basis, `df.to_msgpack()` and using the top-level `pd.to_msgpack(...)` where you can pack arbitrary collections of Python lists, dicts, scalars, while intermixing pandas objects.

```
In [352]: pd.to_msgpack('foo2.msg', {'dict': [{'df': df}, {'string': 'foo'},
   .....:                                     {'scalar': 1.}, {'s': s}]})
   .....:

In [353]: pd.read_msgpack('foo2.msg')
Out[353]:
{'dict': ({'df':           A         B
   0  0.541029  0.554672
   1  0.150831  0.503287
   2  0.834267  0.881894
   3  0.706066  0.726912
   4  0.639300  0.067928},
  {'string': 'foo'},
  {'scalar': 1.0},
  {'s': 2013-01-01    0.753932
   2013-01-02    0.676180
   2013-01-03    0.924728
   2013-01-04    0.338661
   2013-01-05    0.592241
   Freq: D, dtype: float64})}
```

Read/write API

Msgpacks can also be read from and written to strings.

```
In [354]: df.to_msgpack()
Out[354]: b"\x84\xa3typ\xadblock_
↪manager\xa5klass\xa9DataFrame\xa4axes\x92\x86\xa3typ\xa5index\xa5klass\xa5Index\xa4name\xc0\xa5dtyp
↪index\xa5klass\xaaRangeIndex\xa4name\xc0\xa5start\x00\xa4stop\x05\xa4step\x01\xa6blocks\x91\x86\xa
↪`)\x0c3kN\xc3?\xac\xa1:JQ\xb2\xea?\x8c|\xa87\x17\x98\xe6?\xf3H\x83*&u\xe4?\xd4S\xff
↪{\xe0\xbf\xe1?\xd3'2\xea\xed\x1a\xe0?6\x00'gy8\xec?S\x98/\xe7\xdcB\xe7?
↪`\xdbr\xed\xbac\xb1?
↪\xa5shape\x92\x02\x05\xa5dtype\xa7float64\xa5klass\xaaFloatBlock\xa8compress\xc0"
```

Furthermore you can concatenate the strings to produce a list of the original objects.

```
In [355]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[355]:
[          A         B
 0  0.541029  0.554672
 1  0.150831  0.503287
 2  0.834267  0.881894
```

(continues on next page)

(continued from previous page)

```
3  0.706066  0.726912
4  0.639300  0.067928, 2013-01-01    0.753932
2013-01-02    0.676180
2013-01-03    0.924728
2013-01-04    0.338661
2013-01-05    0.592241
Freq: D, dtype: float64]
```

4.1.9 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the *cookbook* for some advanced strategies

> **Warning:** pandas requires `PyTables` >= 3.0.0. There is a indexing bug in `PyTables` < 3.2 which may appear when querying stores using an index. If you see a subset of results being returned, upgrade to `PyTables` >= 3.2. Stores created previously will need to be rewritten using the updated version.

```
In [356]: store = pd.HDFStore('store.h5')

In [357]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Objects can be written to the file just like adding key-value pairs to a dict:

```
In [358]: index = pd.date_range('1/1/2000', periods=8)

In [359]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [360]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....:

# store.put('s', s) is an equivalent method
In [361]: store['s'] = s

In [362]: store['df'] = df

In [363]: store
Out[363]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

In a current or later Python session, you can retrieve stored objects:

```
# store.get('df') is an equivalent method
In [364]: store['df']
Out[364]:
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
```

```
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640

# dotted (attribute) access provides get as well
In [365]: store.df
Out[365]:
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640
```

Deletion of the object specified by the key:

```
# store.remove('df') is an equivalent method
In [366]: del store['df']

In [367]: store
Out[367]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [368]: store.close()

In [369]: store
Out[369]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [370]: store.is_open
Out[370]: False

# Working with, and automatically closing the store using a context manager
In [371]: with pd.HDFStore('store.h5') as store:
   .....:     store.keys()
   .....:
```

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.

```
In [372]: df_tl = pd.DataFrame({'A': list(range(5)), 'B': list(range(5))})

In [373]: df_tl.to_hdf('store_tl.h5', 'table', append=True)

In [374]: pd.read_hdf('store_tl.h5', 'table', where=['index>2'])
```

(continues on next page)

(continued from previous page)

```
Out[374]:
   A  B
3  3  3
4  4  4
```

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting `dropna=True`.

```
In [375]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
   .....:                                 'col2': [1, np.nan, np.nan]})
   .....:

In [376]: df_with_missing
Out[376]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [377]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w')
   .....:

In [378]: pd.read_hdf('file.h5', 'df_with_missing')
Out[378]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [379]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w', dropna=True)
   .....:

In [380]: pd.read_hdf('file.h5', 'df_with_missing')
Out[380]:
   col1  col2
0   0.0   1.0
2   2.0   NaN
```

Fixed format

The examples above show storing using `put`, which write the HDF5 to `PyTables` in a fixed array format, called the `fixed` format. These types of stores are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The `fixed` format stores offer very fast writing and slightly faster reading than `table` stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

Warning: A `fixed` format will raise a `TypeError` if you try to retrieve using a `where`:

```
>>> pd.DataFrame(np.random.randn(10, 2)).to_hdf('test_fixed.h5', 'df')
>>> pd.read_hdf('test_fixed.h5', 'df', where='index>5')
TypeError: cannot pass a where specification when reading a fixed format.
           this store must be selected in its entirety
```

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 and query type operations are supported. This format is specified by format='table' or format='t' to append or put or to_hdf.

This format can be set as an option as well pd.set_option('io.hdf.default_format','table') to enable put/append/to_hdf to by default store in the table format.

```
In [381]: store = pd.HDFStore('store.h5')

In [382]: df1 = df[0:4]

In [383]: df2 = df[4:]

# append data (creates a table automatically)
In [384]: store.append('df', df1)

In [385]: store.append('df', df2)

In [386]: store
Out[386]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [387]: store.select('df')
Out[387]:
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640

# the type of stored data
In [388]: store.root.df._v_attrs.pandas_type
Out[388]: 'frame_table'
```

Note: You can also create a table by passing format='table' or format='t' to a put operation.

Hierarchical keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. foo/bar/bah), which will generate a hierarchy of sub-stores (or Groups in PyTables parlance). Keys can be specified with out the leading / and are **always** absolute (e.g. foo refers to /foo). Removal operations can remove everything in the sub-store and **below**, so be *careful*.

```
In [389]: store.put('foo/bar/bah', df)
In [390]: store.append('food/orange', df)

In [391]: store.append('food/apple', df)

In [392]: store
Out[392]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [393]: store.keys()
Out[393]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [394]: store.remove('food')

In [395]: store
Out[395]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

You can walk through the group hierarchy using the `walk` method which will yield a tuple for each group key along with the relative keys of its contents.

New in version 0.24.0.

```
In [396]: for (path, subgroups, subkeys) in store.walk():
   .....:     for subgroup in subgroups:
   .....:         print('GROUP: {}/{}'.format(path, subgroup))
   .....:     for subkey in subkeys:
   .....:         key = '/'.join([path, subkey])
   .....:         print('KEY: {}'.format(key))
   .....:         print(store.get(key))
   .....:
GROUP: /foo
KEY: /df
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640
GROUP: /foo/bar
KEY: /foo/bar/bah
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640
```

Warning: Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

```
In [8]: store.foo.bar.bah
AttributeError: 'HDFStore' object has no attribute 'foo'

# you can directly access the actual PyTables node but using the root node
In [9]: store.root.foo.bar.bah
Out[9]:
/foo/bar/bah (Group) ''
  children := ['block0_items' (Array), 'block0_values' (Array), 'axis0' (Array),
↪'axis1' (Array)]
```

Instead, use explicit string based keys:

```
In [397]: store['foo/bar/bah']
Out[397]:
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640
```

Storing types

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 attempts at appending longer strings will raise a `ValueError`.

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 [398]: df_mixed = pd.DataFrame({'A': np.random.randn(8),
   .....:                          'B': np.random.randn(8),
   .....:                          'C': np.array(np.random.randn(8),␣
↪dtype='float32'),
   .....:                          'string': 'string',
   .....:                          'int': 1,
   .....:                          'bool': True,
   .....:                          'datetime64': pd.Timestamp('20010102')},
   .....:                         index=list(range(8)))
   .....:

In [399]: df_mixed.loc[df_mixed.index[3:5],
   .....:              ['A', 'B', 'string', 'datetime64']] = np.nan
   .....:

In [400]: store.append('df_mixed', df_mixed, min_itemsize={'values': 50})
In [401]: df_mixed1 = store.select('df_mixed')

In [402]: df_mixed1
Out[402]:
          A         B         C  string  int  bool datetime64
0  0.894171 -1.452159 -0.105646  string    1  True 2001-01-02
1 -1.539066  1.018959  0.028593  string    1  True 2001-01-02
2 -0.114019 -0.087476  0.693070  string    1  True 2001-01-02
3       NaN       NaN -0.646571     NaN    1  True        NaT
4       NaN       NaN -0.174558     NaN    1  True        NaT
5 -2.110838 -1.234633 -1.257271  string    1  True 2001-01-02
6 -0.704558  0.463419 -0.917264  string    1  True 2001-01-02
7 -0.929182  0.841053  0.414183  string    1  True 2001-01-02

In [403]: df_mixed1.dtypes.value_counts()
Out[403]:
float64           2
int64             1
datetime64[ns]    1
object            1
bool              1
float32           1
dtype: int64

# we have provided a minimum string column size
In [404]: store.root.df_mixed.table
Out[404]:
/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=b'', pos=6)}
  byteorder := 'little'
  chunkshape := (689,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

Storing MultiIndex DataFrames

Storing MultiIndex `DataFrames` as tables is very similar to storing/selecting from homogeneous index `DataFrames`.

```
In [405]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   .....:                               ['one', 'two', 'three']],
   .....:                       codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   .....:                              [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:                       names=['foo', 'bar'])
   .....:

In [406]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
   .....:                       columns=['A', 'B', 'C'])
   .....:

In [407]: df_mi
Out[407]:
                  A         B         C
foo bar
foo one    0.072648 -0.851494  0.140402
    two   -0.568937  0.439050  2.531582
    three  0.539277 -1.398668  0.740635
bar one   -1.892064 -0.830925  1.775692
    two    2.183350 -1.565258  1.016985
baz two   -0.476773 -0.566776 -0.665680
    three  0.935387  0.551846  0.786999
qux one    0.481318  0.001118 -0.005084
    two    0.238900 -1.888197 -0.943224
    three -0.761786 -0.706338 -0.594234

In [408]: store.append('df_mi', df_mi)

In [409]: store.select('df_mi')
Out[409]:
                  A         B         C
foo bar
foo one    0.072648 -0.851494  0.140402
    two   -0.568937  0.439050  2.531582
    three  0.539277 -1.398668  0.740635
bar one   -1.892064 -0.830925  1.775692
    two    2.183350 -1.565258  1.016985
baz two   -0.476773 -0.566776 -0.665680
    three  0.935387  0.551846  0.786999
qux one    0.481318  0.001118 -0.005084
    two    0.238900 -1.888197 -0.943224
    three -0.761786 -0.706338 -0.594234

# the levels are automatically included as data columns
In [410]: store.select('df_mi', 'foo=bar')
Out[410]:
                A         B         C
foo bar
bar one -1.892064 -0.830925  1.775692
    two  2.183350 -1.565258  1.016985
```

Querying

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, as a boolean expression.

- `index` and `columns` are supported indexers of a `DataFrames`.
- if `data_columns` are specified, these can be used as additional indexers.

Valid comparison operators are:

`=, ==, !=, >, >=, <, <=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `(` and `)` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

Note:

- = will be automatically expanded to the comparison operator ==
- ~ is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`
- `"columns in ['A', 'D']"`
- `'columns = A'`
- `'columns == A'`
- `"~(columns = ['A', 'B'])"`
- `'index > df.index[3] & string = "bar"'`
- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The `indexers` are on the left-hand side of the sub-expression:

`columns`, `major_axis`, `ts`

The right-hand side of the sub-expression (after a comparison operator) can be:

- functions that will be evaluated, e.g. `Timestamp('2012-02-01')`
- strings, e.g. `"bar"`
- date-like, e.g. `20130101`, or `"20130101"`
- lists, e.g. `"['A', 'B']"`
- variables that are defined in the local names space, e.g. `date`

Note: Passing a string to a query by interpolating it into the query expression is not recommended. Simply assign the string of interest to a variable and use that variable in an expression. For example, do this

```
string = "HolyMoly'"
store.select('df', 'index == string')
```

instead of this

```
string = "HolyMoly'"
store.select('df', 'index == %s' % string)
```

The latter will **not** work and will raise a `SyntaxError`.Note that theres a single quote followed by a double quote in the `string` variable.

If you *must* interpolate, use the `'%r'` format specifier

```
store.select('df', 'index == %r' % string)
```

which will quote `string`.

Here are some examples:

```
In [411]: dfq = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'),
   .....:                    index=pd.date_range('20130101', periods=10))
   .....:

In [412]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [413]: store.select('dfq', "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[413]:
                   A         B
2013-01-05  0.141437 -0.757467
2013-01-06  0.428370  0.281642
2013-01-07 -0.122746  0.909453
2013-01-08  0.319459  0.828792
2013-01-09 -0.446442  0.030712
2013-01-10 -0.627425  0.599256
```

Use and inline column reference

```
In [414]: store.select('dfq', where="A>0 or C>0")
Out[414]:
                   A         B         C         D
2013-01-03  0.099259 -0.456319  0.079027  1.367963
2013-01-04  1.339489  1.086947 -0.918047  0.552759
2013-01-05  0.141437 -0.757467 -1.536944 -0.641387
2013-01-06  0.428370  0.281642  2.297828 -1.406053
2013-01-07 -0.122746  0.909453  1.434186  2.485565
2013-01-08  0.319459  0.828792 -0.221890  0.693093
2013-01-09 -0.446442  0.030712  1.330275  1.509187
```

The `columns` keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a `'columns=list_of_columns_to_filter'`:

```
In [415]: store.select('df', "columns=['A', 'B']")
Out[415]: 
                   A         B
2000-01-01  0.263806  1.913465
2000-01-02  0.283334  1.798001
2000-01-03  0.799684 -0.733715
2000-01-04  0.131478  0.100995
2000-01-05  1.891112 -1.410251
2000-01-06 -0.274852 -0.667027
2000-01-07  0.621607 -1.300199
2000-01-08 -0.999591 -0.320658
```

`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.

Note: `select` will raise a `ValueError` if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is **not** a data_column.

`select` will raise a `SyntaxError` if the query expression is not valid.

Using timedelta64[ns]

You can store and query using the `timedelta64[ns]` type. Terms can be specified in the format: `<float>(<unit>)`, where float may be signed (and fractional), and unit can be `D,s,ms,us,ns` for the timedelta. Heres an example:

```
In [416]: from datetime import timedelta

In [417]: dftd = pd.DataFrame({'A': pd.Timestamp('20130101'),
   .....:                      'B': [pd.Timestamp('20130101') + timedelta(days=i,
   .....:                                                                seconds=10)
   .....:                            for i in range(10)]})
   .....: 

In [418]: dftd['C'] = dftd['A'] - dftd['B']

In [419]: dftd
Out[419]: 
           A                   B                  C
0 2013-01-01 2013-01-01 00:00:10  -1 days +23:59:50
1 2013-01-01 2013-01-02 00:00:10  -2 days +23:59:50
2 2013-01-01 2013-01-03 00:00:10  -3 days +23:59:50
3 2013-01-01 2013-01-04 00:00:10  -4 days +23:59:50
4 2013-01-01 2013-01-05 00:00:10  -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10  -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10  -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10  -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10  -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50

In [420]: store.append('dftd', dftd, data_columns=True)

In [421]: store.select('dftd', "C<'-3.5D'")
Out[421]: 
```

(continues on next page)

(continued from previous page)

```
           A                   B                 C
4 2013-01-01 2013-01-05 00:00:10  -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10  -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10  -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10  -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10  -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50
```

Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`.

Note: Indexes are automagically created 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 [422]: i = store.root.df.table.cols.index.index

In [423]: i.optlevel, i.kind
Out[423]: (6, 'medium')

# change an index by passing new parameters
In [424]: store.create_table_index('df', optlevel=9, kind='full')

In [425]: i = store.root.df.table.cols.index.index

In [426]: i.optlevel, i.kind
Out[426]: (9, 'full')
```

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```
In [427]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [428]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [429]: st = pd.HDFStore('appends.h5', mode='w')

In [430]: st.append('df', df_1, data_columns=['B'], index=False)

In [431]: st.append('df', df_2, data_columns=['B'], index=False)

In [432]: st.get_storer('df').table
Out[432]: 
/df/table (Table(20,)) ''
  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)}
  byteorder := 'little'
  chunkshape := (2730,)
```

Then create the index when finished appending.

```
In [433]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [434]: st.get_storer('df').table
Out[434]:
/df/table (Table(20,)) ''
  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)}
  byteorder := 'little'
  chunkshape := (2730,)
  autoindex := True
  colindexes := {
    "B": Index(9, full, shuffle, zlib(1)).is_csi=True}

In [435]: st.close()
```

See here for how to create a completely-sorted-index (CSI) on an existing store.

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 [436]: df_dc = df.copy()

In [437]: df_dc['string'] = 'foo'

In [438]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan

In [439]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'

In [440]: df_dc['string2'] = 'cool'

In [441]: df_dc.loc[df_dc.index[1:3], ['B', 'C']] = 1.0

In [442]: df_dc
Out[442]:
                   A         B         C string string2
2000-01-01  0.263806  1.913465 -0.274536    foo    cool
2000-01-02  0.283334  1.000000  1.000000    foo    cool
2000-01-03  0.799684  1.000000  1.000000    foo    cool
2000-01-04  0.131478  0.100995 -0.764260    foo    cool
2000-01-05  1.891112 -1.410251  0.752883    NaN    cool
2000-01-06 -0.274852 -0.667027 -0.688782    NaN    cool
2000-01-07  0.621607 -1.300199  0.050119    foo    cool
2000-01-08 -0.999591 -0.320658 -1.922640    bar    cool

# on-disk operations
In [443]: store.append('df_dc', df_dc, data_columns=['B', 'C', 'string',␣
↪'string2'])
```

```
In [444]: store.select('df_dc', where='B > 0')
Out[444]:
                   A         B         C string string2
2000-01-01  0.263806  1.913465 -0.274536    foo    cool
2000-01-02  0.283334  1.000000  1.000000    foo    cool
2000-01-03  0.799684  1.000000  1.000000    foo    cool
2000-01-04  0.131478  0.100995 -0.764260    foo    cool

# getting creative
In [445]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
Out[445]:
                   A    B    C string string2
2000-01-02  0.283334  1.0  1.0    foo    cool
2000-01-03  0.799684  1.0  1.0    foo    cool

# this is in-memory version of this type of selection
In [446]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[446]:
                   A    B    C string string2
2000-01-02  0.283334  1.0  1.0    foo    cool
2000-01-03  0.799684  1.0  1.0    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [447]: store.root.df_dc.table
Out[447]:
/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=b'', pos=4),
  "string2": StringCol(itemsize=4, shape=(), dflt=b'', pos=5)}
  byteorder := 'little'
  chunkshape := (1680,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

There is some performance degradation by making lots of columns into *data columns*, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!).

Iterator

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 [448]: for df in store.select('df', chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01  0.263806  1.913465 -0.274536
2000-01-02  0.283334  1.798001 -0.053258
2000-01-03  0.799684 -0.733715  1.205089
                   A         B         C
2000-01-04  0.131478  0.100995 -0.764260
2000-01-05  1.891112 -1.410251  0.752883
2000-01-06 -0.274852 -0.667027 -0.688782
                   A         B         C
2000-01-07  0.621607 -1.300199  0.050119
2000-01-08 -0.999591 -0.320658 -1.922640
```

Note: You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in pd.read_hdf('store.h5', 'df', chunksize=3):
    print(df)
```

Note, that the chunksize keyword applies to the **source** rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```
In [449]: dfeq = pd.DataFrame({'number': np.arange(1, 11)})

In [450]: dfeq
Out[450]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
9      10

In [451]: store.append('dfeq', dfeq, data_columns=['number'])

In [452]: def chunks(l, n):
   .....:     return [l[i:i + n] for i in range(0, len(l), n)]
   .....:

In [453]: evens = [2, 4, 6, 8, 10]

In [454]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [455]: for c in chunks(coordinates, 2):
   .....:     print(store.select('dfeq', where=c))
   .....:
```

(continues on next page)

(continued from previous page)

```
   number
1       2
3       4
   number
5       6
7       8
   number
9      10
```

Advanced queries

Select a single column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

```
In [456]: store.select_column('df_dc', 'index')
Out[456]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
3   2000-01-04
4   2000-01-05
5   2000-01-06
6   2000-01-07
7   2000-01-08
Name: index, dtype: datetime64[ns]

In [457]: store.select_column('df_dc', 'string')
Out[457]:
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
Name: string, dtype: object
```

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [458]: df_coord = pd.DataFrame(np.random.randn(1000, 2),
   .....:                         index=pd.date_range('20000101',␣
↪periods=1000))
   .....:

In [459]: store.append('df_coord', df_coord)
In [460]: c = store.select_as_coordinates('df_coord', 'index > 20020101')

In [461]: c
Out[461]:
Int64Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
            ...
            990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
           dtype='int64', length=268)

In [462]: store.select('df_coord', where=c)
Out[462]:
                   0         1
2002-01-02 -0.479047 -1.957543
2002-01-03  1.254799  1.315447
2002-01-04 -0.233731 -1.630779
2002-01-05 -1.098851 -1.299909
2002-01-06  0.667606  0.143470
...              ...       ...
2002-09-22 -1.066208 -0.894186
2002-09-23  2.079421 -0.266075
2002-09-24  1.496228 -0.668430
2002-09-25  0.402749 -0.187683
2002-09-26  0.350581  1.152795

[268 rows x 2 columns]
```

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this `mask` would be a resulting `index` from an indexing operation. This example selects the months of a datetimeindex which are 5.

```
In [463]: df_mask = pd.DataFrame(np.random.randn(1000, 2),
   .....:                        index=pd.date_range('20000101', periods=1000))
   .....:

In [464]: store.append('df_mask', df_mask)

In [465]: c = store.select_column('df_mask', 'index')

In [466]: where = c[pd.DatetimeIndex(c).month == 5].index

In [467]: store.select('df_mask', where=where)
Out[467]:
                   0         1
2000-05-01  0.673172  0.388669
2000-05-02  0.761613 -0.368644
2000-05-03  0.435121 -0.062180
2000-05-04  1.677448 -0.019684
2000-05-05  0.562315  1.464591
...              ...       ...
2002-05-27  0.437853  1.315529
2002-05-28  0.931011  0.532589
2002-05-29  0.349402 -0.138098
```

(continues on next page)

(continued from previous page)

```
2002-05-30  0.752838  0.254404
2002-05-31  0.013580  1.078755

[93 rows x 2 columns]
```

Storer object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [468]: store.get_storer('df_dc').nrows
Out[468]: 8
```

Multiple table queries

The methods `append_to_multiple` and `select_as_multiple` 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 tables index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

The `append_to_multiple` method splits a given single DataFrame into multiple tables according to `d`, a dictionary that maps the table names to a list of columns you want in that table. If *None* is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument `selector` defines which table is the selector table (which you can make queries from). The argument `dropna` will drop rows from the input `DataFrame` to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely `np.NaN`, that row will be dropped from all tables.

If `dropna` is False, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. Remember that entirely `np.Nan` rows are not written to the HDFStore, so if you choose to call `dropna=False`, some tables may have more rows than others, and therefore `select_as_multiple` may not work or it may return unexpected results.

```
In [469]: df_mt = pd.DataFrame(np.random.randn(8, 6),
   .....:                      index=pd.date_range('1/1/2000', periods=8),
   .....:                      columns=['A', 'B', 'C', 'D', 'E', 'F'])
   .....:

In [470]: df_mt['foo'] = 'bar'

In [471]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [472]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None},
   .....:                          df_mt, selector='df1_mt')
   .....:

In [473]: store
Out[473]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

```
# individual tables were created
In [474]: store.select('df1_mt')
Out[474]:
                   A         B
2000-01-01  1.222083 -1.048640
2000-01-02       NaN       NaN
2000-01-03 -0.403428 -0.359149
2000-01-04 -0.445969  0.331799
2000-01-05 -0.358375 -0.713913
2000-01-06  0.767579 -1.518142
2000-01-07 -0.125142  0.285073
2000-01-08  1.014284 -0.305314

In [475]: store.select('df2_mt')
Out[475]:
                   C         D         E         F  foo
2000-01-01  0.519195 -1.005739 -1.126444 -0.331994  bar
2000-01-02 -0.220187 -2.576683 -1.531751  1.545878  bar
2000-01-03  0.164228 -0.920410  0.184463 -0.179357  bar
2000-01-04 -1.493748  0.016206 -1.594605 -0.077495  bar
2000-01-05 -0.818602 -0.271994  1.188345  0.345087  bar
2000-01-06 -1.514329 -1.344535 -1.243543  0.231915  bar
2000-01-07 -0.702845  2.420812  0.309805 -1.101996  bar
2000-01-08  0.441337 -0.846243 -0.984939 -1.159608  bar

# as a multiple
In [476]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                           selector='df1_mt')
   .....:
Out[476]:
Empty DataFrame
Columns: [A, B, C, D, E, F, foo]
Index: []
```

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. To get optimal performance, its 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`. Heres 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.

> **Warning:** 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 *repack and clean* the file, use *ptrepack*.

Notes & caveats

Compression

`PyTables` allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

`complevel` specifies if and how hard data is to be compressed. `complevel=0` and `complevel=None` disables compression and `0<complevel<10` enables compression.

`complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:

- zlib: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
- lzo: Fast compression and decompression.
- bzip2: Good compression rates.
- blosc: Fast compression and decompression.

New in version 0.20.2: Support for alternative blosc compressors:

- blosc:blosclz This is the default compressor for `blosc`
- blosc:lz4: A compact, very popular and fast compressor.
- blosc:lz4hc: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
- blosc:snappy: A popular compressor used in many places.
- blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
- blosc:zstd: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

Note: If the library specified with the `complib` option is missing on your platform, compression defaults to `zlib` without further ado.

Enable compression for all objects within the file:

```
store_compressed = pd.HDFStore('store_compressed.h5', complevel=9,
                               complib='blosc:blosclz')
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```
store.append('df', df, complib='zlib', complevel=5)
```

ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

Caveats

Warning: `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 (GH2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- Once a `table` is created columns (DataFrame) are fixed; only exactly the same columns can be appended
- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

Warning: `PyTables` will show a `NaturalNameWarning` if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a `where` clause and are generally a bad idea.

DataTypes

HDFStore will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating : float64, float32, float16	np.nan
integer : int64, int32, int8, uint64,uint32, uint8	
boolean	
datetime64[ns]	NaT
timedelta64[ns]	NaT
categorical : see the section below	
object : strings	np.nan

unicode columns are not supported, and **WILL FAIL**.

Categorical data

You can write data that contains category dtypes to a HDFStore. Queries work the same as if it was an object array. However, the category dtyped data is stored in a more efficient manner.

```
In [477]: dfcat = pd.DataFrame({'A': pd.Series(list('aabbcdba')).
↪astype('category'),
   .....:                       'B': np.random.randn(8)})
   .....:

In [478]: dfcat
Out[478]:
   A         B
0  a -0.182915
1  a -1.740077
2  b -0.227312
3  b -0.351706
4  c -0.484542
5  d  1.150367
6  b -1.223195
7  a  1.022514

In [479]: dfcat.dtypes
Out[479]:
A    category
B     float64
dtype: object

In [480]: cstore = pd.HDFStore('cats.h5', mode='w')

In [481]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [482]: result = cstore.select('dfcat', where="A in ['b', 'c']")

In [483]: result
Out[483]:
   A         B
```

```
2  b -0.227312
3  b -0.351706
4  c -0.484542
6  b -1.223195

In [484]: result.dtypes
Out[484]:
A    category
B     float64
dtype: object
```

String columns

min_itemsize

The underlying implementation of `HDFStore` uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specify the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data_columns* to have this min_itemsize.

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 [485]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'}, index=list(range(5)))

In [486]: dfs
Out[486]:
     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 [487]: store.append('dfs', dfs, min_itemsize=30)

In [488]: store.get_storer('dfs').table
Out[488]:
/dfs/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
```

(continues on next page)

(continued from previous page)

```
    "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 [489]: store.append('dfs2', dfs, min_itemsize={'A': 30})

In [490]: store.get_storer('dfs2').table
Out[490]:
/dfs2/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
  "A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

nan_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [491]: dfss = pd.DataFrame({'A': ['foo', 'bar', 'nan']})

In [492]: dfss
Out[492]:
     A
0  foo
1  bar
2  nan

In [493]: store.append('dfss', dfss)

In [494]: store.select('dfss')
Out[494]:
     A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [495]: store.append('dfss2', dfss, nan_rep='_nan_')

In [496]: store.select('dfss2')
Out[496]:
     A
0  foo
1  bar
2  nan
```

External compatibility

HDFStore writes table format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, HDFStore can read native PyTables format tables.

It is possible to write an HDFStore object that can easily be imported into R using the rhdf5 library (Package website). Create a table format store like this:

```
In [497]: df_for_r = pd.DataFrame({"first": np.random.rand(100),
   .....:                           "second": np.random.rand(100),
   .....:                           "class": np.random.randint(0, 2, (100, ))},
   .....:                          index=range(100))
   .....:

In [498]: df_for_r.head()
Out[498]:
      first    second  class
0  0.253517  0.473526      0
1  0.232906  0.331008      0
2  0.069221  0.532945      1
3  0.290835  0.069538      1
4  0.912722  0.346792      0

In [499]: store_export = pd.HDFStore('export.h5')

In [500]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [501]: store_export
Out[501]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
```

In R this file can be read into a data.frame object using the rhdf5 library. The following example function reads the corresponding column names and data values from the values and assembles them into a data.frame:

```
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.

library(rhdf5)

loadhdf5data <- function(h5File) {

listing <- h5ls(h5File)
# Find all data nodes, values are stored in *_values and corresponding column
# titles in *_items
data_nodes <- grep("_values", listing$name)
name_nodes <- grep("_items", listing$name)
data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
columns = list()
for (idx in seq(data_paths)) {
  # NOTE: matrices returned by h5read have to be transposed to obtain
  # required Fortran order!
  data <- data.frame(t(h5read(h5File, data_paths[idx])))
  names <- t(h5read(h5File, name_paths[idx]))
  entry <- data.frame(data)
  colnames(entry) <- names
  columns <- append(columns, entry)
```

(continues on next page)

(continued from previous page)

```
}

data <- data.frame(columns)

return(data)
}
```

Now you can import the DataFrame into R:

```
> data = loadhdf5data("transfer.hdf5")
> head(data)
         first    second class
1 0.4170220047 0.3266449     0
2 0.7203244934 0.5270581     0
3 0.0001143748 0.8859421     1
4 0.3023325726 0.3572698     1
5 0.1467558908 0.9085352     1
6 0.0923385948 0.6233601     1
```

Note: The R function lists the entire HDF5 files contents and assembles the data.frame object from all matching nodes, so use this only as a starting point if you have stored multiple DataFrame objects to a single HDF5 file.

Performance

- tables format come with a writing performance penalty as compared to fixed stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass chunksize=<int> to append, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass expectedrows=<int> to the first append, to set the TOTAL number of expected rows that PyTables will expected. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A PerformanceWarning will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See Here for more information and some solutions.

4.1.10 Feather

New in version 0.20.0.

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize DataFrames, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.

- The format will NOT write an `Index`, or `MultiIndex` for the `DataFrame` and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation.

```
In [502]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.Categorical(list('abc')),
   .....:                    'g': pd.date_range('20130101', periods=3),
   .....:                    'h': pd.date_range('20130101', periods=3, tz='US/
↪Eastern'),
   .....:                    'i': pd.date_range('20130101', periods=3,␣
↪freq='ns')})
   .....:

In [503]: df
Out[503]:
   a  b  c    d      e  f          g                         h                    ␣
↪           i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01␣
↪00:00:00.000000000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01␣
↪00:00:00.000000001
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01␣
↪00:00:00.000000002

In [504]: df.dtypes
Out[504]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                      category
g                datetime64[ns]
h    datetime64[ns, US/Eastern]
i                datetime64[ns]
dtype: object
```

Write to a feather file.

```
In [505]: df.to_feather('example.feather')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-505-b832ec9cf6be> in <module>
----> 1 df.to_feather('example.feather')
```

(continues on next page)

(continued from previous page)

```
~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_feather(self, fname)
   2150         from pandas.io.feather_format import to_feather
   2151 
-> 2152         to_feather(self, fname)
   2153 
   2154     def to_parquet(

~/sandbox/pandas-release/pandas/pandas/io/feather_format.py in to_feather(df, path)
     21 
     22     """
---> 23     import_optional_dependency("pyarrow")
     24     from pyarrow import feather
     25 

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra)) from␣
↪None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'.  Use pip or conda to install␣
↪pyarrow.
```

Read from a feather file.

```
In [506]: result = pd.read_feather('example.feather')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-506-9efa72931ac9> in <module>
----> 1 result = pd.read_feather('example.feather')

~/sandbox/pandas-release/pandas/pandas/util/_decorators.py in wrapper(*args,␣
↪**kwargs)
    206                 else:
    207                     kwargs[new_arg_name] = new_arg_value
--> 208             return func(*args, **kwargs)
    209 
    210         return wrapper

~/sandbox/pandas-release/pandas/pandas/io/feather_format.py in read_
↪feather(path, columns, use_threads)
    106     type of object stored in file
    107     """
--> 108     pyarrow = import_optional_dependency("pyarrow")
    109     from pyarrow import feather
    110 

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
```

```
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'.  Use pip or conda to␣
↪install pyarrow.

In [507]: result
Out[507]:
   A         B
2  b -0.227312
3  b -0.351706
4  c -0.484542
6  b -1.223195

# we preserve dtypes
In [508]: result.dtypes
Out[508]:
A    category
B     float64
dtype: object
```

4.1.11 Parquet

New in version 0.21.0.

Apache Parquet provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize `DataFrame` s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- The `pyarrow` engine always writes the index to the output, but `fastparquet` only writes non-default indexes. This extra column can cause problems for non-Pandas consumers that are not expecting it. You can force including or omitting indexes with the `index` argument, regardless of the underlying engine.
- Index level names, if specified, must be strings.
- Categorical dtypes can be serialized to parquet, but will de-serialize as `object` dtype.
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

You can specify an `engine` to direct the serialization. This can be one of `pyarrow`, or `fastparquet`, or `auto`. If the engine is NOT specified, then the `pd.options.io.parquet.engine` option is checked; if this is also `auto`, then `pyarrow` is tried, and falling back to `fastparquet`.

See the documentation for pyarrow and fastparquet.

Note: These engines are very similar and should read/write nearly identical parquet format files. Currently `pyarrow` does not support timedelta data, `fastparquet>=0.1.4` supports timezone aware datetimes. These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [509]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.date_range('20130101', periods=3),
   .....:                    'g': pd.date_range('20130101', periods=3, tz='US/
↪Eastern')})
   .....:

In [510]: df
Out[510]:
   a  b  c    d      e          f                         g
0  a  1  3  4.0   True 2013-01-01 2013-01-01 00:00:00-05:00
1  b  2  4  5.0  False 2013-01-02 2013-01-02 00:00:00-05:00
2  c  3  5  6.0   True 2013-01-03 2013-01-03 00:00:00-05:00

In [511]: df.dtypes
Out[511]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Write to a parquet file.

```
In [512]: df.to_parquet('example_pa.parquet', engine='pyarrow')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-512-96e1a42ae5ed> in <module>
----> 1 df.to_parquet('example_pa.parquet', engine='pyarrow')

~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_parquet(self,␣
↪fname, engine, compression, index, partition_cols, **kwargs)
   2235             index=index,
   2236             partition_cols=partition_cols,
-> 2237             **kwargs
   2238         )
   2239

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in to_parquet(df, path,␣
↪engine, compression, index, partition_cols, **kwargs)
    245         Additional keyword arguments passed to the engine
    246     """
--> 247     impl = get_engine(engine)
    248     return impl.write(
    249         df,

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     40
     41     if engine == "pyarrow":
```

```
---> 42         return PyArrowImpl()
     43     elif engine == "fastparquet":
     44         return FastParquetImpl()

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
     76     def __init__(self):
     77         pyarrow = import_optional_dependency(
---> 78             "pyarrow", extra="pyarrow is required for parquet support.
↪"
     79         )
     80         import pyarrow.parquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'. pyarrow is required for␣
↪parquet support. Use pip or conda to install pyarrow.

In [513]: df.to_parquet('example_fp.parquet', engine='fastparquet')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-513-8ba55b61d48c> in <module>
----> 1 df.to_parquet('example_fp.parquet', engine='fastparquet')

~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_parquet(self,␣
↪fname, engine, compression, index, partition_cols, **kwargs)
   2235             index=index,
   2236             partition_cols=partition_cols,
-> 2237             **kwargs
   2238         )
   2239

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in to_parquet(df, path,␣
↪engine, compression, index, partition_cols, **kwargs)
    245         Additional keyword arguments passed to the engine
    246     """
--> 247     impl = get_engine(engine)
    248     return impl.write(
    249         df,

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     42         return PyArrowImpl()
     43     elif engine == "fastparquet":
---> 44         return FastParquetImpl()
     45
     46

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
```

```
    139         # we need to import on first use
    140         fastparquet = import_optional_dependency(
--> 141             "fastparquet", extra="fastparquet is required for parquet␣
↪support."
    142         )
    143         self.api = fastparquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'fastparquet'. fastparquet is␣
↪required for parquet support. Use pip or conda to install fastparquet.
```

Read from a parquet file.

```
In [514]: result = pd.read_parquet('example_fp.parquet', engine='fastparquet')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-514-d4bce7d5df53> in <module>
----> 1 result = pd.read_parquet('example_fp.parquet', engine='fastparquet')

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in read_parquet(path,␣
↪engine, columns, **kwargs)
    293     """
    294 
--> 295     impl = get_engine(engine)
    296     return impl.read(path, columns=columns, **kwargs)

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     42         return PyArrowImpl()
     43     elif engine == "fastparquet":
---> 44         return FastParquetImpl()
     45 
     46 

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
    139         # we need to import on first use
    140         fastparquet = import_optional_dependency(
--> 141             "fastparquet", extra="fastparquet is required for parquet␣
↪support."
    142         )
    143         self.api = fastparquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
```

```
     94         else:
     95             return None

ImportError: Missing optional dependency 'fastparquet'. fastparquet is␣
↪required for parquet support. Use pip or conda to install fastparquet.

In [515]: result = pd.read_parquet('example_pa.parquet', engine='pyarrow')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-515-f9990387c867> in <module>
----> 1 result = pd.read_parquet('example_pa.parquet', engine='pyarrow')

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in read_parquet(path,␣
↪engine, columns, **kwargs)
    293     """
    294 
--> 295     impl = get_engine(engine)
    296     return impl.read(path, columns=columns, **kwargs)

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     40 
     41     if engine == "pyarrow":
---> 42         return PyArrowImpl()
     43     elif engine == "fastparquet":
     44         return FastParquetImpl()

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
     76     def __init__(self):
     77         pyarrow = import_optional_dependency(
---> 78             "pyarrow", extra="pyarrow is required for parquet support.
↪"
     79         )
     80         import pyarrow.parquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'. pyarrow is required for␣
↪parquet support. Use pip or conda to install pyarrow.

In [516]: result.dtypes
Out[516]: 
A    category
B     float64
dtype: object
```

Read only certain columns of a parquet file.

```
In [517]: result = pd.read_parquet('example_fp.parquet',
   .....:                          engine='fastparquet', columns=['a', 'b'])
   .....:
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-517-e4f7add80b89> in <module>
      1 result = pd.read_parquet('example_fp.parquet',
----> 2                          engine='fastparquet', columns=['a', 'b'])

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in read_parquet(path,␣
↪engine, columns, **kwargs)
    293     """
    294 
--> 295     impl = get_engine(engine)
    296     return impl.read(path, columns=columns, **kwargs)

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     42         return PyArrowImpl()
     43     elif engine == "fastparquet":
---> 44         return FastParquetImpl()
     45 
     46 

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
    139         # we need to import on first use
    140         fastparquet = import_optional_dependency(
--> 141             "fastparquet", extra="fastparquet is required for parquet␣
↪support."
    142         )
    143         self.api = fastparquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'fastparquet'. fastparquet is␣
↪required for parquet support. Use pip or conda to install fastparquet.

In [518]: result = pd.read_parquet('example_pa.parquet',
   .....:                          engine='pyarrow', columns=['a', 'b'])
   .....:
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-518-be22c54a7770> in <module>
      1 result = pd.read_parquet('example_pa.parquet',
----> 2                          engine='pyarrow', columns=['a', 'b'])

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in read_parquet(path,␣
↪engine, columns, **kwargs)
    293     """
    294 
--> 295     impl = get_engine(engine)
    296     return impl.read(path, columns=columns, **kwargs)

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     40 
     41     if engine == "pyarrow":
---> 42         return PyArrowImpl()
     43     elif engine == "fastparquet":
     44         return FastParquetImpl()

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
     76     def __init__(self):
     77         pyarrow = import_optional_dependency(
---> 78             "pyarrow", extra="pyarrow is required for parquet support.
↪"
     79         )
     80         import pyarrow.parquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra))␣
↪from None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'. pyarrow is required for␣
↪parquet support. Use pip or conda to install pyarrow.

In [519]: result.dtypes
Out[519]: 
A    category
B     float64
dtype: object
```

Handling indexes

Serializing a `DataFrame` to parquet may include the implicit index as one or more columns in the output file. Thus, this code:

```
In [520]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})

In [521]: df.to_parquet('test.parquet', engine='pyarrow')
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-521-bebaa85a69fa> in <module>
----> 1 df.to_parquet('test.parquet', engine='pyarrow')

~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_parquet(self, fname,␣
↪engine, compression, index, partition_cols, **kwargs)
   2235             index=index,
   2236             partition_cols=partition_cols,
```

(continues on next page)

(continued from previous page)

```
-> 2237                 **kwargs
   2238             )
   2239

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in to_parquet(df, path, engine,␣
↪compression, index, partition_cols, **kwargs)
    245         Additional keyword arguments passed to the engine
    246     """
--> 247     impl = get_engine(engine)
    248     return impl.write(
    249         df,

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     40
     41     if engine == "pyarrow":
---> 42         return PyArrowImpl()
     43     elif engine == "fastparquet":
     44         return FastParquetImpl()

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
     76     def __init__(self):
     77         pyarrow = import_optional_dependency(
---> 78             "pyarrow", extra="pyarrow is required for parquet support."
     79         )
     80         import pyarrow.parquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra)) from␣
↪None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'. pyarrow is required for parquet␣
↪support. Use pip or conda to install pyarrow.
```

creates a parquet file with *three* columns if you use `pyarrow` for serialization: `a`, `b`, and `__index_level_0__`. If youre using `fastparquet`, the index may or may not be written to the file.

This unexpected extra column causes some databases like Amazon Redshift to reject the file, because that column doesnt exist in the target table.

If you want to omit a dataframes indexes when writing, pass `index=False` to *to_parquet()*:

```
In [522]: df.to_parquet('test.parquet', index=False)
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-522-92139e5cd2c3> in <module>
----> 1 df.to_parquet('test.parquet', index=False)

~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_parquet(self, fname,␣
↪engine, compression, index, partition_cols, **kwargs)
   2235                 index=index,
   2236                 partition_cols=partition_cols,
-> 2237                 **kwargs
```

(continues on next page)

(continued from previous page)

```
   2238         )
   2239

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in to_parquet(df, path, engine,␣
↪compression, index, partition_cols, **kwargs)
    245         Additional keyword arguments passed to the engine
    246     """
--> 247     impl = get_engine(engine)
    248     return impl.write(
    249         df,

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     30
     31         raise ImportError(
---> 32             "Unable to find a usable engine; "
     33             "tried using: 'pyarrow', 'fastparquet'.\n"
     34             "pyarrow or fastparquet is required for parquet "

ImportError: Unable to find a usable engine; tried using: 'pyarrow', 'fastparquet'.
pyarrow or fastparquet is required for parquet support
```

This creates a parquet file with just the two expected columns, `a` and `b`. If your `DataFrame` has a custom index, you wont get it back when you load this file into a `DataFrame`.

Passing `index=True` will *always* write the index, even if thats not the underlying engines default behavior.

Partitioning Parquet files

New in version 0.24.0.

Parquet supports partitioning of data based on the values of one or more columns.

```
In [523]: df = pd.DataFrame({'a': [0, 0, 1, 1], 'b': [0, 1, 0, 1]})

In [524]: df.to_parquet(fname='test', engine='pyarrow',
   .....:               partition_cols=['a'], compression=None)
   .....:
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-524-a65c39059f84> in <module>
      1 df.to_parquet(fname='test', engine='pyarrow',
----> 2               partition_cols=['a'], compression=None)

~/sandbox/pandas-release/pandas/pandas/core/frame.py in to_parquet(self, fname,␣
↪engine, compression, index, partition_cols, **kwargs)
   2235             index=index,
   2236             partition_cols=partition_cols,
-> 2237             **kwargs
   2238         )
   2239

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in to_parquet(df, path, engine,␣
↪compression, index, partition_cols, **kwargs)
    245         Additional keyword arguments passed to the engine
    246     """
--> 247     impl = get_engine(engine)
```

(continues on next page)

(continued from previous page)

```
    248     return impl.write(
    249         df,

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in get_engine(engine)
     40
     41     if engine == "pyarrow":
---> 42         return PyArrowImpl()
     43     elif engine == "fastparquet":
     44         return FastParquetImpl()

~/sandbox/pandas-release/pandas/pandas/io/parquet.py in __init__(self)
     76     def __init__(self):
     77         pyarrow = import_optional_dependency(
---> 78             "pyarrow", extra="pyarrow is required for parquet support."
     79         )
     80         import pyarrow.parquet

~/sandbox/pandas-release/pandas/pandas/compat/_optional.py in import_optional_
↪dependency(name, extra, raise_on_missing, on_version)
     91     except ImportError:
     92         if raise_on_missing:
---> 93             raise ImportError(message.format(name=name, extra=extra)) from␣
↪None
     94         else:
     95             return None

ImportError: Missing optional dependency 'pyarrow'. pyarrow is required for parquet␣
↪support. Use pip or conda to install pyarrow.
```

The *fname* specifies the parent directory to which data will be saved. The *partition_cols* are the column names by which the dataset will be partitioned. Columns are partitioned in the order they are given. The partition splits are determined by the unique values in the partition columns. The above example creates a partitioned dataset that may look like:

```
test
 a=0
    0bac803e32dc42ae83fddfd029cbdebc.parquet
     ...
 a=1
     e6ab24a4f45147b49b54a662f0c412a3.parquet
     ...
```

4.1.12 SQL queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Pythons standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some *cookbook examples* for some advanced strategies.

The key functions are:

read_sql_table(table_name, con[, schema,])	Read SQL database table into a DataFrame.
read_sql_query(sql, con[, index_col,])	Read SQL query into a DataFrame.
read_sql(sql, con[, index_col,])	Read SQL query or database table into a DataFrame.
DataFrame.to_sql(self, name, con[, schema,])	Write records stored in a DataFrame to a SQL database.

pandas.read_sql_table

pandas.**read_sql_table**(*table_name*, *con*, *schema=None*, *index_col=None*, *coerce_float=True*, *parse_dates=None*, *columns=None*, *chunksize=None*)

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters

table_name [str] Name of SQL table in database.

con [SQLAlchemy connectable or str] A database URI could be provided as as str. SQLite DBAPI connection mode not supported.

schema [str, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col [str or list of str, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates [list or dict, default None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *pandas.to_datetime()* Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default None] List of column names to select from SQL table.

chunksize [int, default None] If specified, returns an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame A SQL table is returned as two-dimensional data structure with labeled axes.

See also:

read_sql_query Read SQL query into a DataFrame.

read_sql Read SQL query or database table into a DataFrame.

Notes

Any datetime values with time zone information will be converted to UTC.

Examples

```
>>> pd.read_sql_table('table_name', 'postgres:///db_name')  # doctest:+SKIP
```

pandas.read_sql_query

`pandas.`**`read_sql_query`**`(`*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *chunksize=None*`)`

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an *index_col* parameter to use one of the columns as the index, otherwise default integer index will be used.

Parameters

sql [string SQL query or SQLAlchemy Selectable (select or text object)] SQL query to be executed.

con [SQLAlchemy connectable(engine/connection), database string URI,] or sqlite3 DBAPI2 connection Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [boolean, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249s paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={name : value}

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *`pandas.to_datetime()`* Especially useful with databases without native Datetime support, such as SQLite.

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_table` Read SQL database table into a DataFrame.

read_sql

Notes

Any datetime values with time zone information parsed via the *parse_dates* parameter will be converted to UTC.

pandas.read_sql

`pandas.read_sql`(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *columns=None*, *chunksize=None*)

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around `read_sql_table` and `read_sql_query` (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to `read_sql_query`, while a database table name will be routed to `read_sql_table`. Note that the delegated function might have more specific notes about their functionality not listed here.

Parameters

sql [string or SQLAlchemy Selectable (select or text object)] SQL query to be executed or a table name.

con [SQLAlchemy connectable (engine/connection) or database string URI] or DBAPI2 connection (fallback mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [boolean, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249s paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={name : value}

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *`pandas.to_datetime()`* Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default: None] List of column names to select from SQL table (only used when reading a table).

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

read_sql_table Read SQL database table into a DataFrame.

read_sql_query Read SQL query into a DataFrame.

pandas.DataFrame.to_sql

DataFrame.**to_sql**(*self*, *name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [?] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{fail, replace, append}, default fail] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

method [{None, multi, callable}, default None] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- multi: Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section *insert method*.

New in version 0.24.0.

Raises

ValueError When the table already exists and *if_exists* is fail (the default).

See also:

read_sql Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

New in version 0.24.0.

References

[?], [?]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
```

(continues on next page)

(continued from previous page)

```
0   1.0
1   NaN
2   2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

Note: The function *read_sql()* is a convenience wrapper around *read_sql_table()* and *read_sql_query()* (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the SQlite SQL database engine. You can use a temporary SQLite database where data are stored in memory.

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on `create_engine()` and the URI formatting, see the examples below and the SQLAlchemy documentation

```
In [525]: from sqlalchemy import create_engine

# Create your engine.
In [526]: engine = create_engine('sqlite:///:memory:')
```

If you want to manage your own connections you can pass one of those instead:

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

Writing DataFrames

Assuming the following data is in a `DataFrame` `data`, we can insert it into the database using *to_sql()*.

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

```
In [527]: data
Out[527]: 
   id        Date Col_1  Col_2  Col_3
0  26 2010-10-18     X  27.50   True
1  42 2010-10-19     Y -12.50  False
2  63 2010-10-20     Z   5.73   True

In [528]: data.to_sql('data', engine)
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the chunksize parameter when calling to_sql. For example, the following writes data to the database in batches of 1000 rows at a time:

```
In [529]: data.to_sql('data_chunked', engine, chunksize=1000)
```

SQL data types

to_sql() will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype object, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the dtype argument. This argument needs a dictionary mapping column names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy String type instead of the default Text type for string columns:

```
In [530]: from sqlalchemy.types import String

In [531]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

Note: Due to the limited support for timedeltas in the different database flavors, columns with type timedelta64 will be written as integer values as nanoseconds to the database and a warning will be raised.

Note: Columns of category dtype will be converted to the dense representation as you would get with np.asarray(categorical) (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does **not** generate a categorical.

Datetime data types

Using SQLAlchemy, *to_sql()* is capable of writing datetime data that is timezone naive or timezone aware. However, the resulting data stored in the database ultimately depends on the supported data type for datetime data of the database system being used.

The following table lists supported data types for datetime data for some common databases. Other database dialects may have different data types for datetime data.

Database	SQL Datetime Types	Timezone Support
SQLite	TEXT	No
MySQL	TIMESTAMP or DATETIME	No
PostgreSQL	TIMESTAMP or TIMESTAMP WITH TIME ZONE	Yes

When writing timezone aware data to databases that do not support timezones, the data will be written as timezone naive timestamps that are in local time with respect to the timezone.

read_sql_table() is also capable of reading datetime data that is timezone aware or naive. When reading TIMESTAMP WITH TIME ZONE types, pandas will convert the data to UTC.

Insertion method

New in version 0.24.0.

The parameter method controls the SQL insertion clause used. Possible values are:

- None: Uses standard SQL INSERT clause (one per row).
- 'multi': Pass multiple values in a single INSERT clause. It uses a *special* SQL syntax not supported by all backends. This usually provides better performance for analytic databases like *Presto* and *Redshift*, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documention.
- callable with signature (pd_table, conn, keys, data_iter): This can be used to implement a more performant insertion method based on specific backend dialect features.

Example of a callable using PostgreSQL COPY clause:

```
# Alternative to_sql() *method* for DBs that support COPY FROM
import csv
from io import StringIO

def psql_insert_copy(table, conn, keys, data_iter):
    # gets a DBAPI connection that can provide a cursor
    dbapi_conn = conn.connection
    with dbapi_conn.cursor() as cur:
        s_buf = StringIO()
        writer = csv.writer(s_buf)
        writer.writerows(data_iter)
        s_buf.seek(0)

        columns = ', '.join('"{}"'.format(k) for k in keys)
        if table.schema:
            table_name = '{}.{}'.format(table.schema, table.name)
        else:
            table_name = table.name

        sql = 'COPY {} ({}) FROM STDIN WITH CSV'.format(
            table_name, columns)
        cur.copy_expert(sql=sql, file=s_buf)
```

Reading tables

read_sql_table() will read a database table given the table name and optionally a subset of columns to read.

Note: In order to use *read_sql_table()*, you **must** have the SQLAlchemy optional dependency installed.

```
In [532]: pd.read_sql_table('data', engine)
Out[532]: 
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

You can also specify the name of the column as the DataFrame index, and specify a subset of columns to be read.

```
In [533]: pd.read_sql_table('data', engine, index_col='id')
Out[533]:
    index       Date Col_1  Col_2  Col_3
id
26      0 2010-10-18     X  27.50   True
42      1 2010-10-19     Y -12.50  False
63      2 2010-10-20     Z   5.73   True

In [534]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[534]:
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

And you can explicitly force columns to be parsed as dates:

```
In [535]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[535]:
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to *pandas.to_datetime()*:

```
pd.read_sql_table('data', engine, parse_dates={'Date': '%Y-%m-%d'})
pd.read_sql_table('data', engine,
                  parse_dates={'Date': {'format': '%Y-%m-%d %H:%M:%S'}})
```

You can check if a table exists using `has_table()`

Schema support

Reading from and writing to different schemas is supported through the `schema` keyword in the *read_sql_table()* and *to_sql()* functions. Note however that this depends on the database flavor (sqlite does not have schemas). For example:

```
df.to_sql('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')
```

Querying

You can query using raw SQL in the *read_sql_query()* function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```
In [536]: pd.read_sql_query('SELECT * FROM data', engine)
Out[536]:
   index  id                          Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X  27.50      1
1      1  42  2010-10-19 00:00:00.000000     Y -12.50      0
2      2  63  2010-10-20 00:00:00.000000     Z   5.73      1
```

Of course, you can specify a more complex query.

```
In [537]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;",␣
↪engine)
Out[537]:
   id Col_1  Col_2
0  42     Y  -12.5
```

The *read_sql_query()* function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [538]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [539]: df.to_sql('data_chunks', engine, index=False)
```

```
In [540]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks",
   .....:                                 engine, chunksize=5):
   .....:     print(chunk)
   .....:
          a         b         c
0  1.305880  1.962862 -0.365194
1  0.193323  2.602824 -0.896993
2 -2.389181 -1.261600  0.868444
3  0.393496  0.609451 -0.845067
4 -0.322214 -0.447158  0.284659
          a         b         c
0  0.201085 -0.048657  0.720448
1 -0.137165 -0.684949 -0.519145
2  1.271559 -0.861960  0.105051
3 -0.541846 -0.103767 -1.095443
4 -1.361663  1.684795 -0.891315
          a         b         c
0 -0.539063 -0.544872  1.480449
1  0.613611  0.709609 -0.599927
2  0.100530  0.380645 -1.272573
3 -0.868754  0.111400 -1.851400
4 -0.034434  0.323798 -0.247849
          a         b         c
0  1.900594 -0.746429 -0.317105
1 -0.095588 -0.781659  0.205437
2 -0.407292 -0.007602 -0.257895
3  2.356896 -0.650727  0.813023
4  1.094871  0.784225 -0.217173
```

You can also run a plain query without creating a `DataFrame` with `execute()`. This is useful for queries that dont return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
engine = create_engine('sqlite:////absolute/path/to/foo.db')
```

For more information see the examples the SQLAlchemy documentation

Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [541]: import sqlalchemy as sa

In [542]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'),
   .....:             engine, params={'col1': 'X'})
   .....:
Out[542]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X   27.5      1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [543]: metadata = sa.MetaData()

In [544]: data_table = sa.Table('data', metadata,
   .....:                       sa.Column('index', sa.Integer),
   .....:                       sa.Column('Date', sa.DateTime),
   .....:                       sa.Column('Col_1', sa.String),
   .....:                       sa.Column('Col_2', sa.Float),
   .....:                       sa.Column('Col_3', sa.Boolean),
   .....:                       )
   .....:

In [545]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 is True),␣
↪engine)
Out[545]:
Empty DataFrame
Columns: [index, Date, Col_1, Col_2, Col_3]
Index: []
```

You can combine SQLAlchemy expressions with parameters passed to *`read_sql()`* using `sqlalchemy.bindparam()`

```
In [546]: import datetime as dt

In [547]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date
↪'))

In [548]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[548]:
   index        Date Col_1  Col_2  Col_3
0      1 2010-10-19     Y -12.50  False
1      2 2010-10-20     Z   5.73   True
```

Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```
import sqlite3
con = sqlite3.connect(':memory:')
```

And then issue the following queries:

```
data.to_sql('data', con)
pd.read_sql_query("SELECT * FROM data", con)
```

4.1.13 Google BigQuery

Warning: Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package `pandas-gbq`. You can `pip install pandas-gbq` to get it.

The `pandas-gbq` package provides functionality to read/write from Google BigQuery.

pandas integrates with this external package. if `pandas-gbq` is installed, you can use the pandas methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from `pandas-gbq`.

Full documentation can be found here.

4.1.14 Stata format

Writing to stata format

The method `to_stata()` will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

```
In [549]: df = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [550]: df.to_stata('stata.dta')
```

Stata data files have limited data type support; only strings with 244 or fewer characters, `int8`, `int16`, `int32`, `float32` and `float64` can be stored in `.dta` files. Additionally, *Stata* reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in Stata for a particular data type will retype the variable to the next larger size. For example, `int8` values are restricted to lie between -127 and 100 in Stata, and so variables with values above 100 will trigger a conversion to `int16`. `nan` values in floating points data types are stored as the basic missing data type (`.` in *Stata*).

Note: It is not possible to export missing data values for integer data types.

The *Stata* writer gracefully handles other data types including `int64`, `bool`, `uint8`, `uint16`, `uint32` by casting to the smallest supported type that can represent the data. For example, data with a type of `uint8` will be cast to `int8` if all values are less than 100 (the upper bound for non-missing `int8` data in *Stata*), or, if values are outside of this range, the variable is cast to `int16`.

Warning: Conversion from `int64` to `float64` may result in a loss of precision if `int64` values are larger than 2**53.

Warning: `StataWriter` and `to_stata()` only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write *Stata* dta files with strings longer than 244 characters raises a `ValueError`.

Reading from Stata format

The top-level function `read_stata` will read a dta file and return either a `DataFrame` or a `StataReader` that can be used to read the file incrementally.

```
In [551]: pd.read_stata('stata.dta')
Out[551]:
   index         A         B
0      0 -2.802620 -1.031153
1      1 -0.471722 -1.004288
2      2 -0.809833  1.537958
3      3 -0.833349  2.502008
4      4 -1.016559 -0.583782
5      5 -0.369422  0.146956
6      6 -0.815559 -1.032447
7      7  0.676818 -0.410341
8      8  1.171674  2.227913
9      9  0.764637  1.540163
```

Specifying a `chunksize` yields a `StataReader` instance that can be used to read `chunksize` lines from the file at a time. The `StataReader` object can be used as an iterator.

```
In [552]: reader = pd.read_stata('stata.dta', chunksize=3)

In [553]: for df in reader:
   .....:     print(df.shape)
   .....:
(3, 3)
(3, 3)
(3, 3)
(1, 3)
```

For more fine-grained control, use `iterator=True` and specify `chunksize` with each call to `read()`.

```
In [554]: reader = pd.read_stata('stata.dta', iterator=True)

In [555]: chunk1 = reader.read(5)

In [556]: chunk2 = reader.read(5)
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have `object` data type.

Note: *`read_stata()`* and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

Note: Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

Categorical data

`Categorical` data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a `Categorical` and information about *whether* the variable is ordered is lost when exporting.

Warning: *Stata* only supports string value labels, and so `str` is called on the categories when exporting data. Exporting `Categorical` variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as `Categorical` variables using the keyword argument `convert_categoricals` (`True` by default). The keyword argument `order_categoricals` (`True` by default) determines whether imported `Categorical` variables are ordered.

Note: When importing categorical data, the values of the variables in the *Stata* data file are not preserved since `Categorical` variables always use integer data types between `-1` and `n-1` where `n` is the number of categories. If the original values in the *Stata* data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original *Stata* data values and the category codes of imported Categorical variables: missing values are assigned code `-1`, and the smallest original value is assigned `0`, the second smallest is assigned `1` and so on until the largest original value is assigned the code `n-1`.

Note: *Stata* supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a `Categorical` with string categories for the values that are labeled and numeric categories for values with no label.

4.1.15 SAS formats

The top-level function *read_sas()* can read (but not write) SAS *xport* (.XPT) and (since *v0.18.0*) *SAS7BDAT* (.sas7bdat) format files.

SAS files only contain two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates. By default the whole file is read and returned as a `DataFrame`.

Specify a `chunksize` or use `iterator=True` to obtain reader objects (`XportReader` or `SAS7BDATReader`) for incrementally reading the file. The reader objects also have attributes that contain additional information about the file and its variables.

Read a SAS7BDAT file:

```
df = pd.read_sas('sas_data.sas7bdat')
```

Obtain an iterator and read an XPORT file 100,000 lines at a time:

```
def do_something(chunk):
    pass

rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
for chunk in rdr:
    do_something(chunk)
```

The specification for the xport file format is available from the SAS web site.

No official documentation is available for the SAS7BDAT format.

4.1.16 Other file formats

pandas itself only supports IO with a limited set of file formats that map cleanly to its tabular data model. For reading and writing other file formats into and from pandas, we recommend these packages from the broader community.

netCDF

xarray provides data structures inspired by the pandas `DataFrame` for working with multi-dimensional datasets, with a focus on the netCDF file format and easy conversion to and from pandas.

4.1.17 Performance considerations

This is an informal comparison of various IO methods, using pandas 0.20.3. Timings are machine dependent and small differences should be ignored.

```
In [1]: sz = 1000000
In [2]: df = pd.DataFrame({'A': np.random.randn(sz), 'B': [1] * sz})

In [3]: df.info()
```

(continues on next page)

(continued from previous page)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A    1000000 non-null float64
B    1000000 non-null int64
dtypes: float64(1), int64(1)
memory usage: 15.3 MB
```

Given the next test set:

```
from numpy.random import randn

sz = 1000000
df = pd.DataFrame({'A': randn(sz), 'B': [1] * sz})


def test_sql_write(df):
    if os.path.exists('test.sql'):
        os.remove('test.sql')
    sql_db = sqlite3.connect('test.sql')
    df.to_sql(name='test_table', con=sql_db)
    sql_db.close()


def test_sql_read():
    sql_db = sqlite3.connect('test.sql')
    pd.read_sql_query("select * from test_table", sql_db)
    sql_db.close()


def test_hdf_fixed_write(df):
    df.to_hdf('test_fixed.hdf', 'test', mode='w')


def test_hdf_fixed_read():
    pd.read_hdf('test_fixed.hdf', 'test')


def test_hdf_fixed_write_compress(df):
    df.to_hdf('test_fixed_compress.hdf', 'test', mode='w', complib='blosc')


def test_hdf_fixed_read_compress():
    pd.read_hdf('test_fixed_compress.hdf', 'test')


def test_hdf_table_write(df):
    df.to_hdf('test_table.hdf', 'test', mode='w', format='table')


def test_hdf_table_read():
    pd.read_hdf('test_table.hdf', 'test')


def test_hdf_table_write_compress(df):
    df.to_hdf('test_table_compress.hdf', 'test', mode='w',
              complib='blosc', format='table')
```

(continues on next page)

(continued from previous page)

```


def test_hdf_table_read_compress():
    pd.read_hdf('test_table_compress.hdf', 'test')


def test_csv_write(df):
    df.to_csv('test.csv', mode='w')


def test_csv_read():
    pd.read_csv('test.csv', index_col=0)


def test_feather_write(df):
    df.to_feather('test.feather')


def test_feather_read():
    pd.read_feather('test.feather')


def test_pickle_write(df):
    df.to_pickle('test.pkl')


def test_pickle_read():
    pd.read_pickle('test.pkl')


def test_pickle_write_compress(df):
    df.to_pickle('test.pkl.compress', compression='xz')


def test_pickle_read_compress():
    pd.read_pickle('test.pkl.compress', compression='xz')
```

When writing, the top-three functions in terms of speed are are `test_pickle_write`, `test_feather_write` and `test_hdf_fixed_write_compress`.

```
In [14]: %timeit test_sql_write(df)
2.37 s ± 36.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [15]: %timeit test_hdf_fixed_write(df)
194 ms ± 65.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [26]: %timeit test_hdf_fixed_write_compress(df)
119 ms ± 2.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [16]: %timeit test_hdf_table_write(df)
623 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [27]: %timeit test_hdf_table_write_compress(df)
563 ms ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [17]: %timeit test_csv_write(df)
```

(continues on next page)

(continued from previous page)

```
3.13 s ± 49.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [30]: %timeit test_feather_write(df)
103 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [31]: %timeit test_pickle_write(df)
109 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [32]: %timeit test_pickle_write_compress(df)
3.33 s ± 55.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

When reading, the top three are `test_feather_read`, `test_pickle_read` and `test_hdf_fixed_read`.

```
In [18]: %timeit test_sql_read()
1.35 s ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %timeit test_hdf_fixed_read()
14.3 ms ± 438 ţs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [28]: %timeit test_hdf_fixed_read_compress()
23.5 ms ± 672 ţs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [20]: %timeit test_hdf_table_read()
35.4 ms ± 314 ţs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [29]: %timeit test_hdf_table_read_compress()
42.6 ms ± 2.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [22]: %timeit test_csv_read()
516 ms ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [33]: %timeit test_feather_read()
4.06 ms ± 115 ţs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [34]: %timeit test_pickle_read()
6.5 ms ± 172 ţs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [35]: %timeit test_pickle_read_compress()
588 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Space on disk (in bytes)

```
34816000 Aug 21 18:00 test.sql
24009240 Aug 21 18:00 test_fixed.hdf
 7919610 Aug 21 18:00 test_fixed_compress.hdf
24458892 Aug 21 18:00 test_table.hdf
 8657116 Aug 21 18:00 test_table_compress.hdf
28520770 Aug 21 18:00 test.csv
16000248 Aug 21 18:00 test.feather
16000848 Aug 21 18:00 test.pkl
 7554108 Aug 21 18:00 test.pkl.compress
```

{{ header }}

4.2 Indexing and selecting data

The axis labeling information in pandas objects serves many purposes:

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

In this section, we will focus on the final point: namely, how to slice, dice, and generally get and set subsets of pandas objects. The primary focus will be on Series and DataFrame as they have received more development attention in this area.

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 theres 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 isnt known in advance, directly using standard operators has some optimization limits. For production code, we recommended that you take advantage of the optimized pandas data access methods exposed in this chapter.

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Warning: Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see *here*.

See the *MultiIndex / Advanced Indexing* for `MultiIndex` and more advanced indexing documentation.

See the *cookbook* for some advanced strategies.

4.2.1 Different choices for indexing

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

- `.loc` is primarily label based, but may also be used with a boolean array. `.loc` will raise `KeyError` when the items are not found. Allowed inputs are:
  - A single label, e.g. `5` or `'a'` (Note that `5` is interpreted as a *label* of the index. This use is **not** an integer position along the index.).
  - A list or array of labels `['a', 'b', 'c']`.
  - A slice object with labels `'a':'f'` (Note that contrary to usual python slices, **both** the start and the stop are included, when present in the index! See *Slicing with labels* and *Endpoints are inclusive*.)
  - A boolean array
  - A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

    New in version 0.18.1.

See more at *Selection by Label*.

- `.iloc` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array. `.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy *slice* semantics). Allowed inputs are:
  - An integer e.g. `5`.
  - A list or array of integers `[4, 3, 0]`.
  - A slice object with ints `1:7`.
  - A boolean array.
  - A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

    New in version 0.18.1.

  See more at *Selection by Position*, *Advanced Indexing* and *Advanced Hierarchical*.
- `.loc`, `.iloc`, and also `[]` indexing can accept a `callable` as indexer. See more at *Selection By Callable*.

Getting values from an object with multi-axes selection uses the following notation (using `.loc` as an example, but the following applies to `.iloc` 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 equivalent to `p.loc['a', :, :]`.

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`

4.2.2 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. The following table shows return type values when indexing pandas objects with `[]`:

Object Type	Selection	Return Value Type
Series	`series[label]`	scalar value
DataFrame	`frame[colname]`	`Series` corresponding to colname

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

```
In [1]: dates = pd.date_range('1/1/2000', periods=8)

In [2]: df = pd.DataFrame(np.random.randn(8, 4),
   ...:                   index=dates, columns=['A', 'B', 'C', 'D'])
   ...:

In [3]: df
Out[3]:
                   A         B         C         D
2000-01-01 -1.157426 -0.096491  0.999344 -1.482012
2000-01-02  0.189291  0.926828 -0.029095  1.776600
2000-01-03 -1.334294  2.085399 -0.633036  0.208208
2000-01-04 -1.723333 -0.355486 -0.143959  0.177635
2000-01-05  1.071746 -0.516876 -0.382709  0.888600
```

(continues on next page)

(continued from previous page)

```
2000-01-06 -0.156260 -0.720254 -0.837161 -0.426902
2000-01-07 -0.354174  0.510804  0.156535  0.294767
2000-01-08 -1.448608 -1.191084 -0.128338 -0.687717
```

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 [4]: s = df['A']

In [5]: s[dates[5]]
Out[5]: -0.15625969875302725
```

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 [6]: df
Out[6]: 
                   A         B         C         D
2000-01-01 -1.157426 -0.096491  0.999344 -1.482012
2000-01-02  0.189291  0.926828 -0.029095  1.776600
2000-01-03 -1.334294  2.085399 -0.633036  0.208208
2000-01-04 -1.723333 -0.355486 -0.143959  0.177635
2000-01-05  1.071746 -0.516876 -0.382709  0.888600
2000-01-06 -0.156260 -0.720254 -0.837161 -0.426902
2000-01-07 -0.354174  0.510804  0.156535  0.294767
2000-01-08 -1.448608 -1.191084 -0.128338 -0.687717

In [7]: df[['B', 'A']] = df[['A', 'B']]

In [8]: df
Out[8]: 
                   A         B         C         D
2000-01-01 -0.096491 -1.157426  0.999344 -1.482012
2000-01-02  0.926828  0.189291 -0.029095  1.776600
2000-01-03  2.085399 -1.334294 -0.633036  0.208208
2000-01-04 -0.355486 -1.723333 -0.143959  0.177635
2000-01-05 -0.516876  1.071746 -0.382709  0.888600
2000-01-06 -0.720254 -0.156260 -0.837161 -0.426902
2000-01-07  0.510804 -0.354174  0.156535  0.294767
2000-01-08 -1.191084 -1.448608 -0.128338 -0.687717
```

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

Warning: pandas aligns all AXES when setting `Series` and `DataFrame` from `.loc`, and `.iloc`.

This will **not** modify `df` because the column alignment is before value assignment.

```
In [9]: df[['A', 'B']]
Out[9]:
                   A         B
2000-01-01 -0.096491 -1.157426
2000-01-02  0.926828  0.189291
2000-01-03  2.085399 -1.334294
2000-01-04 -0.355486 -1.723333
2000-01-05 -0.516876  1.071746
2000-01-06 -0.720254 -0.156260
2000-01-07  0.510804 -0.354174
2000-01-08 -1.191084 -1.448608

In [10]: df.loc[:, ['B', 'A']] = df[['A', 'B']]

In [11]: df[['A', 'B']]
Out[11]:
                   A         B
2000-01-01 -0.096491 -1.157426
2000-01-02  0.926828  0.189291
2000-01-03  2.085399 -1.334294
2000-01-04 -0.355486 -1.723333
2000-01-05 -0.516876  1.071746
2000-01-06 -0.720254 -0.156260
2000-01-07  0.510804 -0.354174
2000-01-08 -1.191084 -1.448608
```

The correct way to swap column values is by using raw values:

```
In [12]: df.loc[:, ['B', 'A']] = df[['A', 'B']].to_numpy()

In [13]: df[['A', 'B']]
Out[13]:
                   A         B
2000-01-01 -1.157426 -0.096491
2000-01-02  0.189291  0.926828
2000-01-03 -1.334294  2.085399
2000-01-04 -1.723333 -0.355486
2000-01-05  1.071746 -0.516876
2000-01-06 -0.156260 -0.720254
2000-01-07 -0.354174  0.510804
2000-01-08 -1.448608 -1.191084
```

4.2.3 Attribute access

You may access an index on a `Series` or column on a `DataFrame` directly as an attribute:

```
In [14]: sa = pd.Series([1, 2, 3], index=list('abc'))

In [15]: dfa = df.copy()
```

```
In [16]: sa.b
Out[16]: 2

In [17]: dfa.A
Out[17]:
```

```
2000-01-01   -1.157426
2000-01-02    0.189291
2000-01-03   -1.334294
2000-01-04   -1.723333
2000-01-05    1.071746
2000-01-06   -0.156260
2000-01-07   -0.354174
2000-01-08   -1.448608
Freq: D, Name: A, dtype: float64
```

```
In [18]: sa.a = 5

In [19]: sa
Out[19]:
a    5
b    2
c    3
dtype: int64

In [20]: dfa.A = list(range(len(dfa.index)))  # ok if A already exists

In [21]: dfa
Out[21]:
            A         B         C         D
2000-01-01  0 -0.096491  0.999344 -1.482012
2000-01-02  1  0.926828 -0.029095  1.776600
2000-01-03  2  2.085399 -0.633036  0.208208
2000-01-04  3 -0.355486 -0.143959  0.177635
2000-01-05  4 -0.516876 -0.382709  0.888600
2000-01-06  5 -0.720254 -0.837161 -0.426902
2000-01-07  6  0.510804  0.156535  0.294767
2000-01-08  7 -1.191084 -0.128338 -0.687717

In [22]: dfa['A'] = list(range(len(dfa.index)))  # use this form to create a new␣
↪column

In [23]: dfa
Out[23]:
            A         B         C         D
2000-01-01  0 -0.096491  0.999344 -1.482012
2000-01-02  1  0.926828 -0.029095  1.776600
2000-01-03  2  2.085399 -0.633036  0.208208
2000-01-04  3 -0.355486 -0.143959  0.177635
2000-01-05  4 -0.516876 -0.382709  0.888600
2000-01-06  5 -0.720254 -0.837161 -0.426902
2000-01-07  6  0.510804  0.156535  0.294767
2000-01-08  7 -1.191084 -0.128338 -0.687717
```

Warning:

- You can use this access only if the index element is a valid Python identifier, e.g. `s.1` is not allowed. See here for an explanation of valid identifiers.
- The attribute will not be available if it conflicts with an existing method name, e.g. `s.min` is not allowed.
- Similarly, the attribute will not be available if it conflicts with any of the following list: `index`, `major_axis`, `minor_axis`, `items`.

- In any of these cases, standard indexing will still work, e.g. `s['1']`, `s['min']`, and `s['index']` will access the corresponding element or column.

If you are using the IPython environment, you may also use tab-completion to see these accessible attributes.

You can also assign a `dict` to a row of a `DataFrame`:

```
In [24]: x = pd.DataFrame({'x': [1, 2, 3], 'y': [3, 4, 5]})

In [25]: x.iloc[1] = {'x': 9, 'y': 99}

In [26]: x
Out[26]:
   x   y
0  1   3
1  9  99
2  3   5
```

You can use attribute access to modify an existing element of a Series or column of a DataFrame, but be careful; if you try to use attribute access to create a new column, it creates a new attribute rather than a new column. In 0.21.0 and later, this will raise a `UserWarning`:

```
In [1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In [2]: df.two = [4, 5, 6]
UserWarning: Pandas doesn't allow Series to be assigned into nonexistent columns -␣
↪see https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute_access
In [3]: df
Out[3]:
   one
0  1.0
1  2.0
2  3.0
```

4.2.4 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 [27]: s[:5]
Out[27]:
2000-01-01   -1.157426
2000-01-02    0.189291
2000-01-03   -1.334294
2000-01-04   -1.723333
2000-01-05    1.071746
Freq: D, Name: A, dtype: float64

In [28]: s[::2]
Out[28]:
2000-01-01   -1.157426
2000-01-03   -1.334294
2000-01-05    1.071746
2000-01-07   -0.354174
```

```
Freq: 2D, Name: A, dtype: float64

In [29]: s[::-1]
Out[29]:
2000-01-08   -1.448608
2000-01-07   -0.354174
2000-01-06   -0.156260
2000-01-05    1.071746
2000-01-04   -1.723333
2000-01-03   -1.334294
2000-01-02    0.189291
2000-01-01   -1.157426
Freq: -1D, Name: A, dtype: float64
```

Note that setting works as well:

```
In [30]: s2 = s.copy()

In [31]: s2[:5] = 0

In [32]: s2
Out[32]:
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   -0.156260
2000-01-07   -0.354174
2000-01-08   -1.448608
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 [33]: df[:3]
Out[33]:
                   A         B         C         D
2000-01-01 -1.157426 -0.096491  0.999344 -1.482012
2000-01-02  0.189291  0.926828 -0.029095  1.776600
2000-01-03 -1.334294  2.085399 -0.633036  0.208208

In [34]: df[::-1]
Out[34]:
                   A         B         C         D
2000-01-08 -1.448608 -1.191084 -0.128338 -0.687717
2000-01-07 -0.354174  0.510804  0.156535  0.294767
2000-01-06 -0.156260 -0.720254 -0.837161 -0.426902
2000-01-05  1.071746 -0.516876 -0.382709  0.888600
2000-01-04 -1.723333 -0.355486 -0.143959  0.177635
2000-01-03 -1.334294  2.085399 -0.633036  0.208208
2000-01-02  0.189291  0.926828 -0.029095  1.776600
2000-01-01 -1.157426 -0.096491  0.999344 -1.482012
```

4.2.5 Selection by label

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Warning:

> `.loc` is strict when you present slicers that are not compatible (or convertible) with the index type. For example using integers in a `DatetimeIndex`. These will raise a `TypeError`.

```
In [35]: dfl = pd.DataFrame(np.random.randn(5, 4),
   ....:                    columns=list('ABCD'),
   ....:                    index=pd.date_range('20130101', periods=5))
   ....:

In [36]: dfl
Out[36]:
                   A         B         C         D
2013-01-01 -2.100928  1.025928 -0.007973 -1.336035
2013-01-02  0.915382 -0.130655  0.022627 -1.425459
2013-01-03  0.271939  0.169543  0.692513 -1.231139
2013-01-04  1.692870  0.783855 -0.721626  1.698994
2013-01-05 -0.349882  1.586451  1.454199  0.149458
```

```
In [4]: dfl.loc[2:3]
TypeError: cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'>␣
↪with these indexers [2] of <type 'int'>
```

String likes in slicing *can* be convertible to the type of the index and lead to natural slicing.

```
In [37]: dfl.loc['20130102':'20130104']
Out[37]:
                   A         B         C         D
2013-01-02  0.915382 -0.130655  0.022627 -1.425459
2013-01-03  0.271939  0.169543  0.692513 -1.231139
2013-01-04  1.692870  0.783855 -0.721626  1.698994
```

Warning: Starting in 0.21.0, pandas will show a `FutureWarning` if indexing with a list with missing labels. In the future this will raise a `KeyError`. See *list-like Using loc with missing keys in a list is Deprecated*.

pandas provides a suite of methods in order to have **purely label based indexing**. This is a strict inclusion based protocol. Every label asked for must be in the index, or a `KeyError` will be raised. When slicing, both the start bound **AND** the stop bound are *included*, if present in the index. 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, when present in the index! See *Slicing with labels*.

- A boolean array.
- A `callable`, see *Selection By Callable*.

```
In [38]: s1 = pd.Series(np.random.randn(6), index=list('abcdef'))

In [39]: s1
Out[39]: 
a    1.018393
b   -0.565651
c   -1.590951
d   -0.651777
e    0.456407
f    1.134054
dtype: float64

In [40]: s1.loc['c':]
Out[40]: 
c   -1.590951
d   -0.651777
e    0.456407
f    1.134054
dtype: float64

In [41]: s1.loc['b']
Out[41]: -0.5656513106035832
```

Note that setting works as well:

```
In [42]: s1.loc['c':] = 0

In [43]: s1
Out[43]: 
a    1.018393
b   -0.565651
c    0.000000
d    0.000000
e    0.000000
f    0.000000
dtype: float64
```

With a DataFrame:

```
In [44]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list('abcdef'),
   ....:                    columns=list('ABCD'))
   ....: 

In [45]: df1
Out[45]: 
          A         B         C         D
a  1.695493  1.632303 -1.726092  0.486227
b -0.625187  0.386616 -0.048112 -2.598355
c -0.871135 -0.209156  0.004590  0.449006
d  0.573428  0.697186 -2.442512 -1.423556
e -0.304997  0.672794  0.954090  1.323584
f  0.015720 -0.815293  0.164562  0.576599
```

```
In [46]: df1.loc[['a', 'b', 'd'], :]
Out[46]:
          A         B         C         D
a  1.695493  1.632303 -1.726092  0.486227
b -0.625187  0.386616 -0.048112 -2.598355
d  0.573428  0.697186 -2.442512 -1.423556
```

Accessing via label slices:

```
In [47]: df1.loc['d':, 'A':'C']
Out[47]:
          A         B         C
d  0.573428  0.697186 -2.442512
e -0.304997  0.672794  0.954090
f  0.015720 -0.815293  0.164562
```

For getting a cross section using a label (equivalent to `df.xs('a')`):

```
In [48]: df1.loc['a']
Out[48]:
A    1.695493
B    1.632303
C   -1.726092
D    0.486227
Name: a, dtype: float64
```

For getting values with a boolean array:

```
In [49]: df1.loc['a'] > 0
Out[49]:
A     True
B     True
C    False
D     True
Name: a, dtype: bool

In [50]: df1.loc[:, df1.loc['a'] > 0]
Out[50]:
          A         B         D
a  1.695493  1.632303  0.486227
b -0.625187  0.386616 -2.598355
c -0.871135 -0.209156  0.449006
d  0.573428  0.697186 -1.423556
e -0.304997  0.672794  1.323584
f  0.015720 -0.815293  0.576599
```

For getting a value explicitly (equivalent to deprecated `df.get_value('a','A')`):

```
# this is also equivalent to ``df1.at['a','A']``
In [51]: df1.loc['a', 'A']
Out[51]: 1.6954931738440173
```

Slicing with labels

When using `.loc` with slices, if both the start and the stop labels are present in the index, then elements *located* between the two (including them) are returned:

```
In [52]: s = pd.Series(list('abcde'), index=[0, 3, 2, 5, 4])

In [53]: s.loc[3:5]
Out[53]:
3    b
2    c
5    d
dtype: object
```

If at least one of the two is absent, but the index is sorted, and can be compared against start and stop labels, then slicing will still work as expected, by selecting labels which *rank* between the two:

```
In [54]: s.sort_index()
Out[54]:
0    a
2    c
3    b
4    e
5    d
dtype: object

In [55]: s.sort_index().loc[1:6]
Out[55]:
2    c
3    b
4    e
5    d
dtype: object
```

However, if at least one of the two is absent *and* the index is not sorted, an error will be raised (since doing otherwise would be computationally expensive, as well as potentially ambiguous for mixed type indexes). For instance, in the above example, `s.loc[1:6]` would raise `KeyError`.

For the rationale behind this behavior, see *Endpoints are inclusive*.

4.2.6 Selection by position

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Pandas provides a suite of methods in order to get **purely integer based indexing**. The semantics follow closely Python and NumPy slicing. These are `0-based` indexing. When slicing, the start bound is *included*, while the upper bound is *excluded*. Trying to use a non-integer, even a **valid** label will raise an `IndexError`.

The `.iloc` attribute is the primary access method. The following are valid inputs:

- An integer e.g. `5`.
- A list or array of integers `[4, 3, 0]`.
- A slice object with ints `1:7`.

- A boolean array.
- A `callable`, see *Selection By Callable*.

```
In [56]: s1 = pd.Series(np.random.randn(5), index=list(range(0, 10, 2)))

In [57]: s1
Out[57]:
0   -0.813837
2    1.020094
4   -0.538755
6   -0.273898
8    1.374350
dtype: float64

In [58]: s1.iloc[:3]
Out[58]:
0   -0.813837
2    1.020094
4   -0.538755
dtype: float64

In [59]: s1.iloc[3]
Out[59]: -0.2738980637291465
```

Note that setting works as well:

```
In [60]: s1.iloc[:3] = 0

In [61]: s1
Out[61]:
0    0.000000
2    0.000000
4    0.000000
6   -0.273898
8    1.374350
dtype: float64
```

With a DataFrame:

```
In [62]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list(range(0, 12, 2)),
   ....:                    columns=list(range(0, 8, 2)))
   ....:

In [63]: df1
Out[63]:
           0         2         4         6
0  -0.769208 -0.094955 -0.339642  1.131238
2  -1.165074  0.191823 -0.424832  0.641310
4  -1.389117  0.367491  2.164790  1.126079
6   1.550817  0.826973 -0.677486  2.087563
8   0.117134 -0.855723  0.082120  1.276149
10  0.270969  1.210188 -0.988631 -1.253327
```

Select via integer slicing:

```
In [64]: df1.iloc[:3]
```

```
Out[64]:
          0         2         4         6
0 -0.769208 -0.094955 -0.339642  1.131238
2 -1.165074  0.191823 -0.424832  0.641310
4 -1.389117  0.367491  2.164790  1.126079

In [65]: df1.iloc[1:5, 2:4]
Out[65]:
          4         6
2 -0.424832  0.641310
4  2.164790  1.126079
6 -0.677486  2.087563
8  0.082120  1.276149
```

Select via integer list:

```
In [66]: df1.iloc[[1, 3, 5], [1, 3]]
Out[66]:
           2         6
2   0.191823  0.641310
6   0.826973  2.087563
10  1.210188 -1.253327
```

```
In [67]: df1.iloc[1:3, :]
Out[67]:
          0         2         4         6
2 -1.165074  0.191823 -0.424832  0.641310
4 -1.389117  0.367491  2.164790  1.126079
```

```
In [68]: df1.iloc[:, 1:3]
Out[68]:
           2         4
0  -0.094955 -0.339642
2   0.191823 -0.424832
4   0.367491  2.164790
6   0.826973 -0.677486
8  -0.855723  0.082120
10  1.210188 -0.988631
```

```
# this is also equivalent to ``df1.iat[1,1]``
In [69]: df1.iloc[1, 1]
Out[69]: 0.1918230563556888
```

For getting a cross section using an integer position (equiv to `df.xs(1)`):

```
In [70]: df1.iloc[1]
Out[70]:
0   -1.165074
2    0.191823
4   -0.424832
6    0.641310
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/Numpy.

```
# these are allowed in python/numpy.
In [71]: x = list('abcdef')
In [72]: x
Out[72]: ['a', 'b', 'c', 'd', 'e', 'f']

In [73]: x[4:10]
Out[73]: ['e', 'f']

In [74]: x[8:10]
Out[74]: []

In [75]: s = pd.Series(x)

In [76]: s
Out[76]:
0    a
1    b
2    c
3    d
4    e
5    f
dtype: object

In [77]: s.iloc[4:10]
Out[77]:
4    e
5    f
dtype: object

In [78]: s.iloc[8:10]
Out[78]: Series([], dtype: object)
```

Note that using slices that go out of bounds can result in an empty axis (e.g. an empty DataFrame being returned).

```
In [79]: dfl = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [80]: dfl
Out[80]:
          A         B
0 -1.141824  0.487982
1  1.110710  1.775785
2 -0.158929  1.256688
3  0.480722  0.545781
4 -1.214729  0.259405

In [81]: dfl.iloc[:, 2:3]
Out[81]:
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [82]: dfl.iloc[:, 1:3]
Out[82]:
          B
0  0.487982
1  1.775785
```

```
2  1.256688
3  0.545781
4  0.259405

In [83]: dfl.iloc[4:6]
Out[83]:
          A         B
4 -1.214729  0.259405
```

A single indexer that is out of bounds will raise an `IndexError`. A list of indexers where any element is out of bounds will raise an `IndexError`.

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

>>> dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

4.2.7 Selection by callable

New in version 0.18.1.

`.loc`, `.iloc`, and also `[]` indexing can accept a `callable` as indexer. The `callable` must be a function with one argument (the calling Series or DataFrame) that returns valid output for indexing.

```
In [84]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list('abcdef'),
   ....:                    columns=list('ABCD'))
   ....:

In [85]: df1
Out[85]:
          A         B         C         D
a -1.898204  0.933280  0.410757 -1.209116
b  1.072207 -2.076376 -0.032087 -1.179905
c  0.819041  0.169362  0.395066  1.793339
d  0.620358  0.687095  0.924752 -0.953211
e  0.272744 -0.264613 -0.299304  0.828769
f  1.384847  1.408420 -0.599304  1.455457

In [86]: df1.loc[lambda df: df.A > 0, :]
Out[86]:
          A         B         C         D
b  1.072207 -2.076376 -0.032087 -1.179905
c  0.819041  0.169362  0.395066  1.793339
d  0.620358  0.687095  0.924752 -0.953211
e  0.272744 -0.264613 -0.299304  0.828769
f  1.384847  1.408420 -0.599304  1.455457

In [87]: df1.loc[:, lambda df: ['A', 'B']]
Out[87]:
          A         B
a -1.898204  0.933280
b  1.072207 -2.076376
c  0.819041  0.169362
```

```
d  0.620358  0.687095
e  0.272744 -0.264613
f  1.384847  1.408420

In [88]: df1.iloc[:, lambda df: [0, 1]]
Out[88]: 
          A         B
a -1.898204  0.933280
b  1.072207 -2.076376
c  0.819041  0.169362
d  0.620358  0.687095
e  0.272744 -0.264613
f  1.384847  1.408420

In [89]: df1[lambda df: df.columns[0]]
Out[89]: 
a   -1.898204
b    1.072207
c    0.819041
d    0.620358
e    0.272744
f    1.384847
Name: A, dtype: float64
```

You can use callable indexing in `Series`.

```
In [90]: df1.A.loc[lambda s: s > 0]
Out[90]: 
b    1.072207
c    0.819041
d    0.620358
e    0.272744
f    1.384847
Name: A, dtype: float64
```

Using these methods / indexers, you can chain data selection operations without using a temporary variable.

```
In [91]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [92]: (bb.groupby(['year', 'team']).sum()
   ....:    .loc[lambda df: df.r > 100])
   ....: 
Out[92]: 
           stint    g    ab    r    h  X2b  X3b  hr    rbi    sb   cs   bb     so  ␣
↪ibb   hbp    sh    sf  gidp
year team                                                                            ␣
↪
2007 CIN       6  379   745  101  203   35    2  36  125.0  10.0  1.0  105  127.0  14.
↪0   1.0   1.0  15.0  18.0
     DET       5  301  1062  162  283   54    4  37  144.0  24.0  7.0   97  176.0   3.
↪0  10.0   4.0   8.0  28.0
     HOU       4  311   926  109  218   47    6  14   77.0  10.0  4.0   60  212.0   3.
↪0   9.0  16.0   6.0  17.0
     LAN      11  413  1021  153  293   61    3  36  154.0   7.0  5.0  114  141.0   8.
↪0   9.0   3.0   8.0  29.0
     NYN      13  622  1854  240  509  101    3  61  243.0  22.0  4.0  174  310.0  24.
↪0  23.0  18.0  15.0  48.0
```

(continues on next page)

(continued from previous page)

```
     SFN        5   482   1305   198   337    67     6   40   171.0   26.0   7.0   235   188.0   51.
↪0   8.0   16.0    6.0   41.0
     TEX        2   198    729   115   200    40     4   28   115.0   21.0   4.0    73   140.0    4.
↪0   5.0    2.0    8.0   16.0
     TOR        4   459   1408   187   378    96     2   58   223.0    4.0   2.0   190   265.0   16.
↪0  12.0    4.0   16.0   38.0
```

4.2.8 IX indexer is deprecated

Warning: Starting in 0.20.0, the `.ix` indexer is deprecated, in favor of the more strict `.iloc` and `.loc` indexers.

`.ix` offers a lot of magic on the inference of what the user wants to do. To wit, `.ix` can decide to index *positionally* OR via *labels* depending on the data type of the index. This has caused quite a bit of user confusion over the years.

The recommended methods of indexing are:

- `.loc` if you want to *label* index.
- `.iloc` if you want to *positionally* index.

```
In [93]: dfd = pd.DataFrame({'A': [1, 2, 3],
   ....:                     'B': [4, 5, 6]},
   ....:                    index=list('abc'))
   ....:

In [94]: dfd
Out[94]:
   A  B
a  1  4
b  2  5
c  3  6
```

Previous behavior, where you wish to get the 0th and the 2nd elements from the index in the A column.

```
In [3]: dfd.ix[[0, 2], 'A']
Out[3]:
a    1
c    3
Name: A, dtype: int64
```

Using `.loc`. Here we will select the appropriate indexes from the index, then use *label* indexing.

```
In [95]: dfd.loc[dfd.index[[0, 2]], 'A']
Out[95]:
a    1
c    3
Name: A, dtype: int64
```

This can also be expressed using `.iloc`, by explicitly getting locations on the indexers, and using *positional* indexing to select things.

```
In [96]: dfd.iloc[[0, 2], dfd.columns.get_loc('A')]
Out[96]:
```

(continues on next page)

(continued from previous page)

```
a    1
c    3
Name: A, dtype: int64
```

For getting *multiple* indexers, using `.get_indexer`:

```
In [97]: dfd.iloc[[0, 2], dfd.columns.get_indexer(['A', 'B'])]
Out[97]: 
   A  B
a  1  4
c  3  6
```

4.2.9 Indexing with list with missing labels is deprecated

> **Warning:** Starting in 0.21.0, using `.loc` or `[]` with a list with one or more missing labels, is deprecated, in favor of `.reindex`.

In prior versions, using `.loc[list-of-labels]` would work as long as *at least 1* of the keys was found (otherwise it would raise a `KeyError`). This behavior is deprecated and will show a warning message pointing to this section. The recommended alternative is to use `.reindex()`.

For example.

```
In [98]: s = pd.Series([1, 2, 3])

In [99]: s
Out[99]: 
0    1
1    2
2    3
dtype: int64
```

Selection with all keys found is unchanged.

```
In [100]: s.loc[[1, 2]]
Out[100]: 
1    2
2    3
dtype: int64
```

Previous behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Current behavior

```
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc with any non-matching elements will raise
```

(continues on next page)

(continued from previous page)

```
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-
↪listlike

Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Reindexing

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`. See also the section on *reindexing*.

```
In [101]: s.reindex([1, 2, 3])
Out[101]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Alternatively, if you want to select only *valid* keys, the following is idiomatic and efficient; it is guaranteed to preserve the dtype of the selection.

```
In [102]: labels = [1, 2, 3]

In [103]: s.loc[s.index.intersection(labels)]
Out[103]:
1    2
2    3
dtype: int64
```

Having a duplicated index will raise for a `.reindex()`:

```
In [104]: s = pd.Series(np.arange(4), index=['a', 'a', 'b', 'c'])

In [105]: labels = ['c', 'd']
```

```
In [17]: s.reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

Generally, you can intersect the desired labels with the current axis, and then reindex.

```
In [106]: s.loc[s.index.intersection(labels)].reindex(labels)
Out[106]:
c    3.0
d    NaN
dtype: float64
```

However, this would *still* raise if your resulting index is duplicated.

```
In [41]: labels = ['a', 'd']

In [42]: s.loc[s.index.intersection(labels)].reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

4.2.10 Selecting random samples

A random selection of rows or columns from a Series or DataFrame with the `sample()` method. The method will sample rows by default, and accepts a specific number of rows/columns to return, or a fraction of rows.

```
In [107]: s = pd.Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1 row.
In [108]: s.sample()
Out[108]:
0    0
dtype: int64

# One may specify either a number of rows:
In [109]: s.sample(n=3)
Out[109]:
2    2
1    1
5    5
dtype: int64

# Or a fraction of the rows:
In [110]: s.sample(frac=0.5)
Out[110]:
0    0
2    2
3    3
dtype: int64
```

By default, `sample` will return each row at most once, but one can also sample with replacement using the `replace` option:

```
In [111]: s = pd.Series([0, 1, 2, 3, 4, 5])

# Without replacement (default):
In [112]: s.sample(n=6, replace=False)
Out[112]:
3    3
2    2
4    4
0    0
5    5
1    1
dtype: int64

# With replacement:
In [113]: s.sample(n=6, replace=True)
Out[113]:
0    0
```

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

By default, each row has an equal probability of being selected, but if you want rows to have different probabilities, you can pass the `sample` function sampling weights as `weights`. These weights can be a list, a NumPy array, or a Series, but they must be of the same length as the object you are sampling. Missing values will be treated as a weight of zero, and inf values are not allowed. If weights do not sum to 1, they will be re-normalized by dividing all weights by the sum of the weights. For example:

```
In [114]: s = pd.Series([0, 1, 2, 3, 4, 5])

In [115]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [116]: s.sample(n=3, weights=example_weights)
Out[116]:
5    5
4    4
3    3
dtype: int64

# Weights will be re-normalized automatically
In [117]: example_weights2 = [0.5, 0, 0, 0, 0, 0]

In [118]: s.sample(n=1, weights=example_weights2)
Out[118]:
0    0
dtype: int64
```

When applied to a DataFrame, you can use a column of the DataFrame as sampling weights (provided you are sampling rows and not columns) by simply passing the name of the column as a string.

```
In [119]: df2 = pd.DataFrame({'col1': [9, 8, 7, 6],
   .....:                     'weight_column': [0.5, 0.4, 0.1, 0]})
   .....:

In [120]: df2.sample(n=3, weights='weight_column')
Out[120]:
   col1  weight_column
0     9            0.5
2     7            0.1
1     8            0.4
```

`sample` also allows users to sample columns instead of rows using the `axis` argument.

```
In [121]: df3 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

In [122]: df3.sample(n=1, axis=1)
Out[122]:
   col1
0     1
1     2
2     3
```

Finally, one can also set a seed for `samples` random number generator using the `random_state` argument, which will accept either an integer (as a seed) or a NumPy RandomState object.

```
In [123]: df4 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

# With a given seed, the sample will always draw the same rows.
In [124]: df4.sample(n=2, random_state=2)
Out[124]:
   col1  col2
2     3     4
1     2     3

In [125]: df4.sample(n=2, random_state=2)
Out[125]:
   col1  col2
2     3     4
1     2     3
```

4.2.11 Setting with enlargement

The `.loc/[]` operations can perform enlargement when setting a non-existent key for that axis.

In the `Series` case this is effectively an appending operation.

```
In [126]: se = pd.Series([1, 2, 3])

In [127]: se
Out[127]:
0    1
1    2
2    3
dtype: int64

In [128]: se[5] = 5.

In [129]: se
Out[129]:
0    1.0
1    2.0
2    3.0
5    5.0
dtype: float64
```

A `DataFrame` can be enlarged on either axis via `.loc`.

```
In [130]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
   .....:                    columns=['A', 'B'])
   .....:

In [131]: dfi
Out[131]:
   A  B
0  0  1
1  2  3
2  4  5

In [132]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']
```

(continues on next page)

(continued from previous page)

```
In [133]: dfi
Out[133]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an `append` operation on the `DataFrame`.

```
In [134]: dfi.loc[3] = 5

In [135]: dfi
Out[135]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```

4.2.12 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 youre asking for. If you only want to access a scalar value, the fastest way is to use the `at` and `iat` methods, which are implemented on all of the data structures.

Similarly to `loc`, `at` provides **label** based scalar lookups, while, `iat` provides **integer** based lookups analogously to `iloc`

```
In [136]: s.iat[5]
Out[136]: 5

In [137]: df.at[dates[5], 'A']
Out[137]: -0.15625969875302725

In [138]: df.iat[3, 0]
Out[138]: -1.7233332966009836
```

You can also set using these same indexers.

```
In [139]: df.at[dates[5], 'E'] = 7

In [140]: df.iat[3, 0] = 7
```

`at` may enlarge the object in-place as above if the indexer is missing.

```
In [141]: df.at[dates[-1] + pd.Timedelta('1 day'), 0] = 7

In [142]: df
Out[142]:
                   A         B         C         D    E    0
2000-01-01 -1.157426 -0.096491  0.999344 -1.482012  NaN  NaN
2000-01-02  0.189291  0.926828 -0.029095  1.776600  NaN  NaN
2000-01-03 -1.334294  2.085399 -0.633036  0.208208  NaN  NaN
2000-01-04  7.000000 -0.355486 -0.143959  0.177635  NaN  NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-05  1.071746 -0.516876 -0.382709  0.888600  NaN  NaN
2000-01-06 -0.156260 -0.720254 -0.837161 -0.426902  7.0  NaN
2000-01-07 -0.354174  0.510804  0.156535  0.294767  NaN  NaN
2000-01-08 -1.448608 -1.191084 -0.128338 -0.687717  NaN  NaN
2000-01-09       NaN       NaN       NaN       NaN  NaN  7.0
```

4.2.13 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, since by default Python will evaluate an expression such as `df.A > 2 & df.B < 3` as `df.A > (2 & df.B) < 3`, while the desired evaluation order is `(df.A > 2) & (df.B < 3)`.

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

```
In [143]: s = pd.Series(range(-3, 4))

In [144]: s
Out[144]: 
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int64

In [145]: s[s > 0]
Out[145]: 
4    1
5    2
6    3
dtype: int64

In [146]: s[(s < -1) | (s > 0.5)]
Out[146]: 
0   -3
1   -2
4    1
5    2
6    3
dtype: int64

In [147]: s[~(s < 0)]
Out[147]: 
3    0
4    1
5    2
6    3
dtype: int64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrames index (for example, something derived from one of the columns of the DataFrame):

```
In [148]: df[df['A'] > 0]
Out[148]: 
                   A         B         C         D   E   0
2000-01-02  0.189291  0.926828 -0.029095  1.776600 NaN NaN
2000-01-04  7.000000 -0.355486 -0.143959  0.177635 NaN NaN
2000-01-05  1.071746 -0.516876 -0.382709  0.888600 NaN NaN
```

List comprehensions and the `map` method of Series can also be used to produce more complex criteria:

```
In [149]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'three', 'two',␣
↪'one', 'six'],
   .....:                     'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                     'c': np.random.randn(7)})
   .....: 

# only want 'two' or 'three'
In [150]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [151]: df2[criterion]
Out[151]: 
       a  b         c
2    two  y  0.344065
3  three  x  1.275247
4    two  y  1.303763

# equivalent but slower
In [152]: df2[[x.startswith('t') for x in df2['a']]]
Out[152]: 
       a  b         c
2    two  y  0.344065
3  three  x  1.275247
4    two  y  1.303763

# Multiple criteria
In [153]: df2[criterion & (df2['b'] == 'x')]
Out[153]: 
       a  b         c
3  three  x  1.275247
```

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 [154]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
Out[154]: 
   b         c
3  x  1.275247
```

4.2.14 Indexing with isin

Consider the `isin()` method of `Series`, which returns a boolean vector that is true wherever the `Series` elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

```
In [155]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')
In [156]: s
Out[156]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [157]: s.isin([2, 4, 6])
Out[157]:
4    False
3    False
2     True
1    False
0     True
dtype: bool

In [158]: s[s.isin([2, 4, 6])]
Out[158]:
2    2
0    4
dtype: int64
```

The same method is available for `Index` objects and is useful for the cases when you dont know which of the sought labels are in fact present:

```
In [159]: s[s.index.isin([2, 4, 6])]
Out[159]:
4    0
2    2
dtype: int64

# compare it to the following
In [160]: s.reindex([2, 4, 6])
Out[160]:
2    2.0
4    0.0
6    NaN
dtype: float64
```

In addition to that, `MultiIndex` allows selecting a separate level to use in the membership check:

```
In [161]: s_mi = pd.Series(np.arange(6),
   .....:                  index=pd.MultiIndex.from_product([[0, 1], ['a',␣
↪'b', 'c']]))
   .....:

In [162]: s_mi
Out[162]:
0  a    0
   b    1
   c    2
1  a    3
   b    4
   c    5
dtype: int64

In [163]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
Out[163]:
0  c    2
1  a    3
dtype: int64

In [164]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
Out[164]:
0  a    0
   c    2
1  a    3
   c    5
dtype: int64
```

DataFrame also has an `isin()` method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

```
In [165]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
   .....:                    'ids2': ['a', 'n', 'c', 'n']})
   .....:

In [166]: values = ['a', 'b', 1, 3]

In [167]: df.isin(values)
Out[167]:
    vals    ids   ids2
0   True   True   True
1  False   True  False
2   True  False  False
3  False  False  False
```

Oftentimes youll want to match certain values with certain columns. Just make values a `dict` where the key is the column, and the value is a list of items you want to check for.

```
In [168]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [169]: df.isin(values)
Out[169]:
    vals    ids   ids2
0   True   True  False
1  False   True  False
2   True  False  False
3  False  False  False
```

Combine DataFrames `isin` with the `any()` and `all()` methods to quickly select subsets of your data that meet a given criteria. To select a row where each column meets its own criterion:

```
In [170]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}

In [171]: row_mask = df.isin(values).all(1)

In [172]: df[row_mask]
Out[172]:
```

(continues on next page)

(continued from previous page)

```
   vals ids ids2
0     1   a    a
```

4.2.15 The `where()` Method and Masking

Selecting values from a Series with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows:

```
In [173]: s[s > 0]
Out[173]:
3    1
2    2
1    3
0    4
dtype: int64
```

To return a Series of the same shape as the original:

```
In [174]: s.where(s > 0)
Out[174]:
4    NaN
3    1.0
2    2.0
1    3.0
0    4.0
dtype: float64
```

Selecting values from a DataFrame with a boolean criterion now also preserves input data shape. `where` is used under the hood as the implementation. The code below is equivalent to `df.where(df < 0)`.

```
In [175]: df[df < 0]
Out[175]:
                   A         B         C         D
2000-01-01 -0.986205 -1.719758 -2.230079 -0.439106
2000-01-02 -2.397242       NaN       NaN -0.237058
2000-01-03 -1.482014       NaN -0.782186       NaN
2000-01-04 -0.480306 -1.051903 -0.987736 -0.182060
2000-01-05 -0.379467       NaN -0.138556       NaN
2000-01-06 -0.514897 -0.117796 -0.108906 -1.142649
2000-01-07 -1.349120       NaN -0.128845 -1.352644
2000-01-08       NaN       NaN       NaN -1.495080
```

In addition, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```
In [176]: df.where(df < 0, -df)
Out[176]:
                   A         B         C         D
2000-01-01 -0.986205 -1.719758 -2.230079 -0.439106
2000-01-02 -2.397242 -0.124508 -1.493995 -0.237058
2000-01-03 -1.482014 -0.429889 -0.782186 -0.389666
2000-01-04 -0.480306 -1.051903 -0.987736 -0.182060
2000-01-05 -0.379467 -0.273248 -0.138556 -0.881904
```

(continues on next page)

(continued from previous page)

```
2000-01-06 -0.514897 -0.117796 -0.108906 -1.142649
2000-01-07 -1.349120 -0.316880 -0.128845 -1.352644
2000-01-08 -0.161458 -0.739064 -0.165377 -1.495080
```

You may wish to set values based on some boolean criteria. This can be done intuitively like so:

```
In [177]: s2 = s.copy()

In [178]: s2[s2 < 0] = 0

In [179]: s2
Out[179]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [180]: df2 = df.copy()

In [181]: df2[df2 < 0] = 0

In [182]: df2
Out[182]:
                   A         B         C         D
2000-01-01  0.000000  0.000000  0.000000  0.000000
2000-01-02  0.000000  0.124508  1.493995  0.000000
2000-01-03  0.000000  0.429889  0.000000  0.389666
2000-01-04  0.000000  0.000000  0.000000  0.000000
2000-01-05  0.000000  0.273248  0.000000  0.881904
2000-01-06  0.000000  0.000000  0.000000  0.000000
2000-01-07  0.000000  0.316880  0.000000  0.000000
2000-01-08  0.161458  0.739064  0.165377  0.000000
```

By default, `where` returns a modified copy of the data. There is an optional parameter `inplace` so that the original data can be modified without creating a copy:

```
In [183]: df_orig = df.copy()

In [184]: df_orig.where(df > 0, -df, inplace=True)

In [185]: df_orig
Out[185]:
                   A         B         C         D
2000-01-01  0.986205  1.719758  2.230079  0.439106
2000-01-02  2.397242  0.124508  1.493995  0.237058
2000-01-03  1.482014  0.429889  0.782186  0.389666
2000-01-04  0.480306  1.051903  0.987736  0.182060
2000-01-05  0.379467  0.273248  0.138556  0.881904
2000-01-06  0.514897  0.117796  0.108906  1.142649
2000-01-07  1.349120  0.316880  0.128845  1.352644
2000-01-08  0.161458  0.739064  0.165377  1.495080
```

Note: The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

```
In [186]: df.where(df < 0, -df) == np.where(df < 0, df, -df)
Out[186]:
               A     B     C     D
2000-01-01  True  True  True  True
2000-01-02  True  True  True  True
2000-01-03  True  True  True  True
2000-01-04  True  True  True  True
2000-01-05  True  True  True  True
2000-01-06  True  True  True  True
2000-01-07  True  True  True  True
2000-01-08  True  True  True  True
```

Alignment

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

```
In [187]: df2 = df.copy()

In [188]: df2[df2[1:4] > 0] = 3

In [189]: df2
Out[189]:
                   A         B         C         D
2000-01-01 -0.986205 -1.719758 -2.230079 -0.439106
2000-01-02 -2.397242  3.000000  3.000000 -0.237058
2000-01-03 -1.482014  3.000000 -0.782186  3.000000
2000-01-04 -0.480306 -1.051903 -0.987736 -0.182060
2000-01-05 -0.379467  0.273248 -0.138556  0.881904
2000-01-06 -0.514897 -0.117796 -0.108906 -1.142649
2000-01-07 -1.349120  0.316880 -0.128845 -1.352644
2000-01-08  0.161458  0.739064  0.165377 -1.495080
```

Where can also accept `axis` and `level` parameters to align the input when performing the `where`.

```
In [190]: df2 = df.copy()

In [191]: df2.where(df2 > 0, df2['A'], axis='index')
Out[191]:
                   A         B         C         D
2000-01-01 -0.986205 -0.986205 -0.986205 -0.986205
2000-01-02 -2.397242  0.124508  1.493995 -2.397242
2000-01-03 -1.482014  0.429889 -1.482014  0.389666
2000-01-04 -0.480306 -0.480306 -0.480306 -0.480306
2000-01-05 -0.379467  0.273248 -0.379467  0.881904
2000-01-06 -0.514897 -0.514897 -0.514897 -0.514897
2000-01-07 -1.349120  0.316880 -1.349120 -1.349120
2000-01-08  0.161458  0.739064  0.165377  0.161458
```

This is equivalent to (but faster than) the following.

```
In [192]: df2 = df.copy()

In [193]: df.apply(lambda x, y: x.where(x > 0, y), y=df['A'])
Out[193]:
                   A         B         C         D
```

(continues on next page)

(continued from previous page)

```
2000-01-01 -0.986205 -0.986205 -0.986205 -0.986205
2000-01-02 -2.397242  0.124508  1.493995 -2.397242
2000-01-03 -1.482014  0.429889 -1.482014  0.389666
2000-01-04 -0.480306 -0.480306 -0.480306 -0.480306
2000-01-05 -0.379467  0.273248 -0.379467  0.881904
2000-01-06 -0.514897 -0.514897 -0.514897 -0.514897
2000-01-07 -1.349120  0.316880 -1.349120 -1.349120
2000-01-08  0.161458  0.739064  0.165377  0.161458
```

New in version 0.18.1.

Where can accept a callable as condition and `other` arguments. The function must be with one argument (the calling Series or DataFrame) and that returns valid output as condition and `other` argument.

```
In [194]: df3 = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [4, 5, 6],
   .....:                     'C': [7, 8, 9]})
   .....:

In [195]: df3.where(lambda x: x > 4, lambda x: x + 10)
Out[195]:
    A   B  C
0  11  14  7
1  12   5  8
2  13   6  9
```

Mask

mask() is the inverse boolean operation of `where`.

```
In [196]: s.mask(s >= 0)
Out[196]:
4   NaN
3   NaN
2   NaN
1   NaN
0   NaN
dtype: float64

In [197]: df.mask(df >= 0)
Out[197]:
                   A         B         C         D
2000-01-01 -0.986205 -1.719758 -2.230079 -0.439106
2000-01-02 -2.397242       NaN       NaN -0.237058
2000-01-03 -1.482014       NaN -0.782186       NaN
2000-01-04 -0.480306 -1.051903 -0.987736 -0.182060
2000-01-05 -0.379467       NaN -0.138556       NaN
2000-01-06 -0.514897 -0.117796 -0.108906 -1.142649
2000-01-07 -1.349120       NaN -0.128845 -1.352644
2000-01-08       NaN       NaN       NaN -1.495080
```

4.2.16 The `query()` Method

DataFrame objects have a *query()* method that allows selection using an expression.

You can get the value of the frame where column b has values between the values of columns a and c. For example:

```
In [198]: n = 10

In [199]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [200]: df
Out[200]:
          a         b         c
0  0.564354  0.446221  0.031135
1  0.902823  0.612305  0.538181
2  0.663245  0.164731  0.782808
3  0.194429  0.253263  0.923684
4  0.698026  0.425477  0.840259
5  0.739830  0.738261  0.105783
6  0.214871  0.824421  0.246428
7  0.821225  0.243330  0.650292
8  0.373653  0.171969  0.479279
9  0.450277  0.365091  0.217416

# pure python
In [201]: df[(df.a < df.b) & (df.b < df.c)]
Out[201]:
          a         b         c
3  0.194429  0.253263  0.923684

# query
In [202]: df.query('(a < b) & (b < c)')
Out[202]:
          a         b         c
3  0.194429  0.253263  0.923684
```

Do the same thing but fall back on a named index if there is no column with the name a.

```
In [203]: df = pd.DataFrame(np.random.randint(n / 2, size=(n, 2)),␣
↪columns=list('bc'))

In [204]: df.index.name = 'a'

In [205]: df
Out[205]:
   b  c
a
0  4  0
1  3  1
2  4  3
3  0  2
4  1  4
5  4  4
6  4  2
7  2  0
8  0  0
9  1  3

In [206]: df.query('a < b and b < c')
```

```
Out[206]:
Empty DataFrame
Columns: [b, c]
Index: []
```

If instead you dont want to or cannot name your index, you can use the name `index` in your query expression:

```
In [207]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)),␣
↪columns=list('bc'))

In [208]: df
Out[208]:
   b  c
0  4  5
1  4  1
2  0  8
3  4  7
4  0  0
5  8  5
6  1  2
7  4  3
8  1  0
9  6  3

In [209]: df.query('index < b < c')
Out[209]:
   b  c
0  4  5
3  4  7
```

Note: If the name of your index overlaps with a column name, the column name is given precedence. For example,

```
In [210]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})

In [211]: df.index.name = 'a'

In [212]: df.query('a > 2')  # uses the column 'a', not the index
Out[212]:
   a
a
0  4
1  4
2  3
```

You can still use the index in a query expression by using the special identifier index:

```
In [213]: df.query('index > 2')
Out[213]:
   a
a
3  1
4  1
```

If for some reason you have a column named `index`, then you can refer to the index as `ilevel_0` as well, but at this point you should consider renaming your columns to something less ambiguous.

`MultiIndex` `query()` Syntax

You can also use the levels of a `DataFrame` with a *`MultiIndex`* as if they were columns in the frame:

```
In [214]: n = 10

In [215]: colors = np.random.choice(['red', 'green'], size=n)

In [216]: foods = np.random.choice(['eggs', 'ham'], size=n)

In [217]: colors
Out[217]:
array(['red', 'red', 'green', 'green', 'red', 'red', 'green', 'green',
       'green', 'red'], dtype='<U5')

In [218]: foods
Out[218]:
array(['eggs', 'ham', 'ham', 'ham', 'ham', 'ham', 'eggs', 'eggs', 'ham',
       'eggs'], dtype='<U4')

In [219]: index = pd.MultiIndex.from_arrays([colors, foods], names=['color',␣
↪'food'])

In [220]: df = pd.DataFrame(np.random.randn(n, 2), index=index)

In [221]: df
Out[221]:
                   0         1
color food
red   eggs  0.397240  1.722883
      ham   0.634589  1.761948
green ham   1.191222 -0.748678
      ham  -0.013401 -0.982325
red   ham   0.272726  1.042615
      ham   0.267082  0.191461
green eggs -0.435659 -0.035917
      eggs  0.194931  0.970348
      ham   2.187055  0.383666
red   eggs -0.812383 -0.497327

In [222]: df.query('color == "red"')
Out[222]:
                   0         1
color food
red   eggs  0.397240  1.722883
      ham   0.634589  1.761948
      ham   0.272726  1.042615
      ham   0.267082  0.191461
      eggs -0.812383 -0.497327
```

If the levels of the `MultiIndex` are unnamed, you can refer to them using special names:

```
In [223]: df.index.names = [None, None]

In [224]: df
```

```
Out[224]:
                   0         1
red   eggs  0.397240  1.722883
      ham   0.634589  1.761948
green ham   1.191222 -0.748678
      ham  -0.013401 -0.982325
red   ham   0.272726  1.042615
      ham   0.267082  0.191461
green eggs -0.435659 -0.035917
      eggs  0.194931  0.970348
      ham   2.187055  0.383666
red   eggs -0.812383 -0.497327

In [225]: df.query('ilevel_0 == "red"')
Out[225]:
                 0         1
red eggs  0.397240  1.722883
    ham   0.634589  1.761948
    ham   0.272726  1.042615
    ham   0.267082  0.191461
    eggs -0.812383 -0.497327
```

The convention is `ilevel_0`, which means index level 0 for the 0th level of the `index`.

`query()` Use Cases

A use case for *`query()`* is when you have a collection of *`DataFrame`* objects that have a subset of column names (or index levels/names) in common. You can pass the same query to both frames *without* having to specify which frame youre interested in querying

```
In [226]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [227]: df
Out[227]:
          a         b         c
0  0.483974  0.645639  0.413412
1  0.611039  0.585546  0.848970
2  0.523271  0.811649  0.517849
3  0.947506  0.143525  0.055154
4  0.934891  0.214973  0.271028
5  0.832143  0.777114  0.572133
6  0.304056  0.712288  0.960006
7  0.965451  0.803696  0.866318
8  0.965355  0.383391  0.647743
9  0.639263  0.218103  0.886788

In [228]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)

In [229]: df2
Out[229]:
           a         b         c
0   0.563472  0.298326  0.361543
1   0.021200  0.761846  0.279478
2   0.274321  0.127032  0.025433
3   0.789059  0.154680  0.999703
```

(continues on next page)

(continued from previous page)

```
4   0.195936  0.042450  0.475367
5   0.970329  0.053024  0.293762
6   0.877607  0.352530  0.300746
7   0.259895  0.666779  0.920354
8   0.861035  0.176572  0.638339
9   0.083984  0.834057  0.673247
10  0.190267  0.647251  0.586836
11  0.395322  0.575815  0.184662

In [230]: expr = '0.0 <= a <= c <= 0.5'

In [231]: map(lambda frame: frame.query(expr), [df, df2])
Out[231]: <map at 0x125179610>
```

query() Python versus pandas Syntax Comparison

Full numpy-like syntax:

```
In [232]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)),␣
↪columns=list('abc'))

In [233]: df
Out[233]:
   a  b  c
0  4  1  0
1  9  3  7
2  6  7  9
3  1  9  4
4  6  9  9
5  1  1  2
6  6  6  6
7  9  6  3
8  0  7  9
9  3  8  3

In [234]: df.query('(a < b) & (b < c)')
Out[234]:
   a  b  c
2  6  7  9
8  0  7  9

In [235]: df[(df.a < df.b) & (df.b < df.c)]
Out[235]:
   a  b  c
2  6  7  9
8  0  7  9
```

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than & and |).

```
In [236]: df.query('a < b & b < c')
Out[236]:
   a  b  c
2  6  7  9
8  0  7  9
```

Use English instead of symbols:

```
In [237]: df.query('a < b and b < c')
Out[237]:
   a  b  c
2  6  7  9
8  0  7  9
```

Pretty close to how you might write it on paper:

```
In [238]: df.query('a < b < c')
Out[238]:
   a  b  c
2  6  7  9
8  0  7  9
```

The in and not in operators

query() also supports special use of Pythons in and not in comparison operators, providing a succinct syntax for calling the isin method of a Series or DataFrame.

```
# get all rows where columns "a" and "b" have overlapping values
In [239]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b':␣
↪list('aaaabbbbcccc'),
   .....:                    'c': np.random.randint(5, size=12),
   .....:                    'd': np.random.randint(9, size=12)})
   .....:

In [240]: df
Out[240]:
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
2   b  a  4  1
3   b  a  4  3
4   c  b  4  5
5   c  b  0  0
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
9   e  c  2  0
10  f  c  2  2
11  f  c  4  7

In [241]: df.query('a in b')
Out[241]:
   a  b  c  d
0  a  a  1  5
1  a  a  1  1
2  b  a  4  1
3  b  a  4  3
4  c  b  4  5
5  c  b  0  0

# How you'd do it in pure Python
```

```
In [242]: df[df.a.isin(df.b)]
Out[242]:
   a  b  c  d
0  a  a  1  5
1  a  a  1  1
2  b  a  4  1
3  b  a  4  3
4  c  b  4  5
5  c  b  0  0

In [243]: df.query('a not in b')
Out[243]:
    a  b  c  d
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
9   e  c  2  0
10  f  c  2  2
11  f  c  4  7

# pure Python
In [244]: df[~df.a.isin(df.b)]
Out[244]:
    a  b  c  d
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
9   e  c  2  0
10  f  c  2  2
11  f  c  4  7
```

You can combine this with other expressions for very succinct queries:

```
# rows where cols a and b have overlapping values
# and col c's values are less than col d's
In [245]: df.query('a in b and c < d')
Out[245]:
   a  b  c  d
0  a  a  1  5
4  c  b  4  5

# pure Python
In [246]: df[df.b.isin(df.a) & (df.c < df.d)]
Out[246]:
    a  b  c  d
0   a  a  1  5
4   c  b  4  5
6   d  b  0  4
8   e  c  4  5
11  f  c  4  7
```

Note: Note that `in` and `not in` are evaluated in Python, since `numexpr` has no equivalent of this operation. However, **only the** `in`/`not in` **expression itself** is evaluated in vanilla Python. For example, in the expression

```
df.query('a in b + c + d')
```

`(b + c + d)` is evaluated by `numexpr` and *then* the `in` operation is evaluated in plain Python. In general, any operations that can be evaluated using `numexpr` will be.

Special use of the == operator with `list` objects

Comparing a `list` of values to a column using ==/!= works similarly to `in`/`not in`.

```
In [247]: df.query('b == ["a", "b", "c"]')
Out[247]:
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
2   b  a  4  1
3   b  a  4  3
4   c  b  4  5
5   c  b  0  0
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
9   e  c  2  0
10  f  c  2  2
11  f  c  4  7

# pure Python
In [248]: df[df.b.isin(["a", "b", "c"])]
Out[248]:
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
2   b  a  4  1
3   b  a  4  3
4   c  b  4  5
5   c  b  0  0
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
9   e  c  2  0
10  f  c  2  2
11  f  c  4  7

In [249]: df.query('c == [1, 2]')
Out[249]:
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
9   e  c  2  0
10  f  c  2  2

In [250]: df.query('c != [1, 2]')
Out[250]:
    a  b  c  d
2   b  a  4  1
3   b  a  4  3
4   c  b  4  5
5   c  b  0  0
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
11  f  c  4  7

# using in/not in
In [251]: df.query('[1, 2] in c')
Out[251]: 
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
9   e  c  2  0
10  f  c  2  2

In [252]: df.query('[1, 2] not in c')
Out[252]: 
    a  b  c  d
2   b  a  4  1
3   b  a  4  3
4   c  b  4  5
5   c  b  0  0
6   d  b  0  4
7   d  b  3  3
8   e  c  4  5
11  f  c  4  7

# pure Python
In [253]: df[df.c.isin([1, 2])]
Out[253]: 
    a  b  c  d
0   a  a  1  5
1   a  a  1  1
9   e  c  2  0
10  f  c  2  2
```

Boolean operators

You can negate boolean expressions with the word `not` or the ~ operator.

```
In [254]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [255]: df['bools'] = np.random.rand(len(df)) > 0.5

In [256]: df.query('~bools')
Out[256]: 
          a         b         c  bools
0  0.499604  0.981560  0.137759  False
4  0.104751  0.782568  0.977198  False
7  0.973197  0.245000  0.977406  False
```

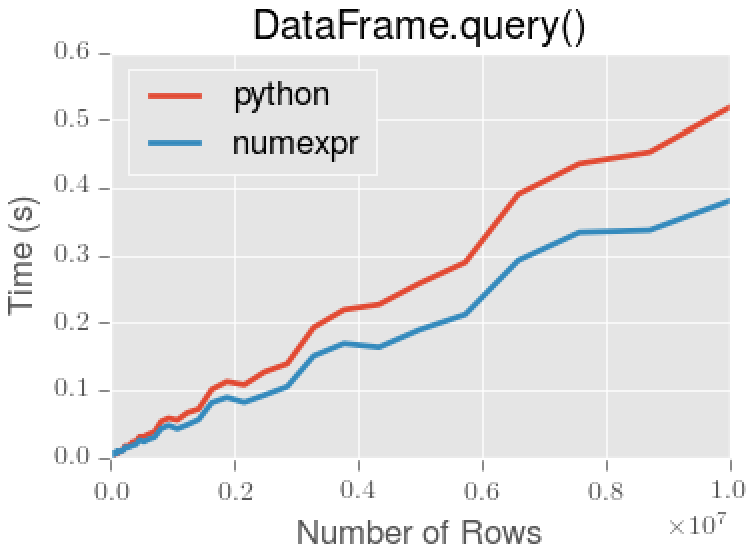
```
9  0.805940  0.451425  0.070470  False

In [257]: df.query('not bools')
Out[257]: 
          a         b         c  bools
0  0.499604  0.981560  0.137759  False
4  0.104751  0.782568  0.977198  False
7  0.973197  0.245000  0.977406  False
9  0.805940  0.451425  0.070470  False

In [258]: df.query('not bools') == df[~df.bools]
Out[258]: 
      a     b     c  bools
0  True  True  True   True
4  True  True  True   True
7  True  True  True   True
9  True  True  True   True
```

Of course, expressions can be arbitrarily complex too:

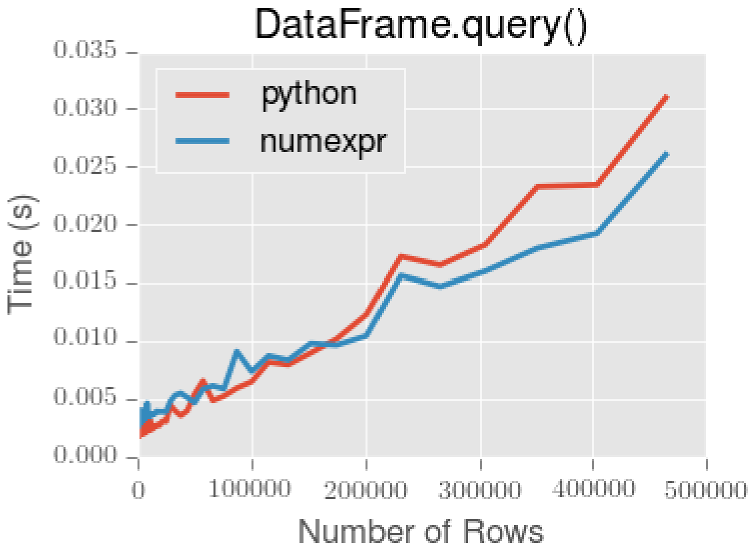
```
# short query syntax
In [259]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [260]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools␣
 ↪> 2)]

In [261]: shorter
Out[261]: 
          a         b         c  bools
4  0.104751  0.782568  0.977198  False

In [262]: longer
Out[262]: 
          a         b         c  bools
4  0.104751  0.782568  0.977198  False

In [263]: shorter == longer
Out[263]: 
      a     b     c  bools
4  True  True  True   True
```

Performance of `query()`

`DataFrame.query()` using `numexpr` is slightly faster than Python for large frames.



Note: You will only see the performance benefits of using the `numexpr` engine with `DataFrame.query()` if your frame has more than approximately 200,000 rows.



This plot was created using a `DataFrame` with 3 columns each containing floating point values generated using `numpy.random.randn()`.

4.2.17 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 keep parameter to specify targets to be kept.

- keep='first' (default): mark / drop duplicates except for the first occurrence.
- keep='last': mark / drop duplicates except for the last occurrence.
- keep=False: mark / drop all duplicates.

```
In [264]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two',␣
↪'three', 'four'],
   .....:                     'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
   .....:                     'c': np.random.randn(7)})
   .....:

In [265]: df2
Out[265]:
       a  b         c
0    one  x -0.086643
1    one  y -0.862428
2    two  x  1.155986
3    two  y -0.583644
4    two  x -1.416461
5  three  x  0.799196
6   four  x -2.063856

In [266]: df2.duplicated('a')
Out[266]:
0    False
1     True
2    False
3     True
4     True
5    False
6    False
dtype: bool

In [267]: df2.duplicated('a', keep='last')
Out[267]:
0     True
1    False
2     True
3     True
4    False
5    False
6    False
dtype: bool

In [268]: df2.duplicated('a', keep=False)
Out[268]:
0     True
1     True
2     True
3     True
```

```
4     True
5    False
6    False
dtype: bool

In [269]: df2.drop_duplicates('a')
Out[269]:
       a  b         c
0    one  x -0.086643
2    two  x  1.155986
5  three  x  0.799196
6   four  x -2.063856

In [270]: df2.drop_duplicates('a', keep='last')
Out[270]:
       a  b         c
1    one  y -0.862428
4    two  x -1.416461
5  three  x  0.799196
6   four  x -2.063856

In [271]: df2.drop_duplicates('a', keep=False)
Out[271]:
       a  b         c
5  three  x  0.799196
6   four  x -2.063856
```

Also, you can pass a list of columns to identify duplications.

```
In [272]: df2.duplicated(['a', 'b'])
Out[272]:
0    False
1    False
2    False
3    False
4     True
5    False
6    False
dtype: bool

In [273]: df2.drop_duplicates(['a', 'b'])
Out[273]:
       a  b         c
0    one  x -0.086643
1    one  y -0.862428
2    two  x  1.155986
3    two  y -0.583644
5  three  x  0.799196
6   four  x -2.063856
```

To drop duplicates by index value, use `Index.duplicated` then perform slicing. The same set of options are available for the `keep` parameter.

```
In [274]: df3 = pd.DataFrame({'a': np.arange(6),
   .....:                     'b': np.random.randn(6)},
   .....:                    index=['a', 'a', 'b', 'c', 'b', 'a'])
   .....:

In [275]: df3
Out[275]: 
   a         b
a  0 -0.457673
a  1  0.315795
b  2 -0.013959
c  3 -0.376069
b  4 -0.715356
a  5  1.802760

In [276]: df3.index.duplicated()
Out[276]: array([False,  True, False, False,  True,  True])

In [277]: df3[~df3.index.duplicated()]
Out[277]: 
   a         b
a  0 -0.457673
b  2 -0.013959
c  3 -0.376069

In [278]: df3[~df3.index.duplicated(keep='last')]
Out[278]: 
   a         b
c  3 -0.376069
b  4 -0.715356
a  5  1.802760

In [279]: df3[~df3.index.duplicated(keep=False)]
Out[279]: 
   a         b
c  3 -0.376069
```

4.2.18 Dictionary-like `get()` method

Each of Series or DataFrame have a `get` method which can return a default value.

```
In [280]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [281]: s.get('a')  # equivalent to s['a']
Out[281]: 1

In [282]: s.get('x', default=-1)
Out[282]: -1
```

4.2.19 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 [283]: dflookup = pd.DataFrame(np.random.rand(20, 4), columns = ['A', 'B', 'C', 'D
↪'])

In [284]: dflookup.lookup(list(range(0, 10, 2)), ['B', 'C', 'A', 'B', 'D'])
Out[284]: array([0.85612986, 0.02825952, 0.99403226, 0.02263787, 0.37166623])
```

4.2.20 Index objects

The pandas *Index* class and its subclasses can be viewed as implementing an *ordered multiset*. Duplicates are allowed. However, if you try to convert an *Index* object with duplicate entries into a `set`, an exception will be raised.

Index also provides the infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create an *Index* directly is to pass a `list` or other sequence to *Index*:

```
In [285]: index = pd.Index(['e', 'd', 'a', 'b'])

In [286]: index
Out[286]: Index(['e', 'd', 'a', 'b'], dtype='object')

In [287]: 'd' in index
Out[287]: True
```

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

```
In [288]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')

In [289]: index.name
Out[289]: 'something'
```

The name, if set, will be shown in the console display:

```
In [290]: index = pd.Index(list(range(5)), name='rows')

In [291]: columns = pd.Index(['A', 'B', 'C'], name='cols')

In [292]: df = pd.DataFrame(np.random.randn(5, 3), index=index,␣
↪columns=columns)

In [293]: df
Out[293]:
cols          A         B         C
rows
0     -1.539526  0.083551  1.819217
1     -0.556258 -1.013751  0.804958
2      1.138849 -0.913212  2.047493
3     -0.894783  1.059103 -0.605857
4     -1.096832  0.217643  2.122047

In [294]: df['A']
Out[294]:
rows
0   -1.539526
1   -0.556258
2    1.138849
```

```
3   -0.894783
4   -1.096832
Name: A, dtype: float64
```

Setting metadata

Indexes are mostly immutable, but it is possible to set and change their metadata, like the index `name` (or, for `MultiIndex`, `levels` and `codes`).

You can use the `rename`, `set_names`, `set_levels`, and `set_codes` to set these attributes directly. They default to returning a copy; however, you can specify `inplace=True` to have the data change in place.

See *Advanced Indexing* for usage of MultiIndexes.

```
In [295]: ind = pd.Index([1, 2, 3])

In [296]: ind.rename("apple")
Out[296]: Int64Index([1, 2, 3], dtype='int64', name='apple')

In [297]: ind
Out[297]: Int64Index([1, 2, 3], dtype='int64')

In [298]: ind.set_names(["apple"], inplace=True)

In [299]: ind.name = "bob"

In [300]: ind
Out[300]: Int64Index([1, 2, 3], dtype='int64', name='bob')
```

`set_names`, `set_levels`, and `set_codes` also take an optional `level` argument

```
In [301]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']],␣
↪names=['first', 'second'])

In [302]: index
Out[302]:
MultiIndex([(0, 'one'),
            (0, 'two'),
            (1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['first', 'second'])

In [303]: index.levels[1]
Out[303]: Index(['one', 'two'], dtype='object', name='second')

In [304]: index.set_levels(["a", "b"], level=1)
Out[304]:
MultiIndex([(0, 'a'),
            (0, 'b'),
            (1, 'a'),
            (1, 'b'),
            (2, 'a'),
            (2, 'b')],
           names=['first', 'second'])
```

Set operations on Index objects

The two main operations are `union (|)` and `intersection (&)`. These can be directly called as instance methods or used via overloaded operators. Difference is provided via the `.difference()` method.

```
In [305]: a = pd.Index(['c', 'b', 'a'])

In [306]: b = pd.Index(['c', 'e', 'd'])

In [307]: a | b
Out[307]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [308]: a & b
Out[308]: Index(['c'], dtype='object')

In [309]: a.difference(b)
Out[309]: Index(['a', 'b'], dtype='object')
```

Also available is the `symmetric_difference (^)` operation, which returns elements that appear in either `idx1` or `idx2`, but not in both. This is equivalent to the Index created by `idx1.difference(idx2).union(idx2.difference(idx1))`, with duplicates dropped.

```
In [310]: idx1 = pd.Index([1, 2, 3, 4])

In [311]: idx2 = pd.Index([2, 3, 4, 5])

In [312]: idx1.symmetric_difference(idx2)
Out[312]: Int64Index([1, 5], dtype='int64')

In [313]: idx1 ^ idx2
Out[313]: Int64Index([1, 5], dtype='int64')
```

Note: The resulting index from a set operation will be sorted in ascending order.

When performing `Index.union()` between indexes with different dtypes, the indexes must be cast to a common dtype. Typically, though not always, this is object dtype. The exception is when performing a union between integer and float data. In this case, the integer values are converted to float

```
In [314]: idx1 = pd.Index([0, 1, 2])

In [315]: idx2 = pd.Index([0.5, 1.5])

In [316]: idx1 | idx2
Out[316]: Float64Index([0.0, 0.5, 1.0, 1.5, 2.0], dtype='float64')
```

Missing values

Important: Even though `Index` can hold missing values (`NaN`), it should be avoided if you do not want any unexpected results. For example, some operations exclude missing values implicitly.

`Index.fillna` fills missing values with specified scalar value.

```
In [317]: idx1 = pd.Index([1, np.nan, 3, 4])
In [318]: idx1
Out[318]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')

In [319]: idx1.fillna(2)
Out[319]: Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')

In [320]: idx2 = pd.DatetimeIndex([pd.Timestamp('2011-01-01'),
   .....:                          pd.NaT,
   .....:                          pd.Timestamp('2011-01-03')])
   .....:

In [321]: idx2
Out[321]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'],␣
↪dtype='datetime64[ns]', freq=None)

In [322]: idx2.fillna(pd.Timestamp('2011-01-02'))
Out[322]: DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'],␣
↪dtype='datetime64[ns]', freq=None)
```

4.2.21 Set / reset index

Occasionally you will load or create a data set into a DataFrame and want to add an index after youve already done so. There are a couple of different ways.

Set an index

DataFrame has a `set_index()` method which takes a column name (for a regular `Index`) or a list of column names (for a `MultiIndex`). To create a new, re-indexed DataFrame:

```
In [323]: data
Out[323]:
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0

In [324]: indexed1 = data.set_index('c')

In [325]: indexed1
Out[325]:
     a    b    d
c
z  bar  one  1.0
y  bar  two  2.0
x  foo  one  3.0
w  foo  two  4.0

In [326]: indexed2 = data.set_index(['a', 'b'])

In [327]: indexed2
Out[327]:
         c    d
```

(continues on next page)

(continued from previous page)

```
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

The append keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [328]: frame = data.set_index('c', drop=False)

In [329]: frame = frame.set_index(['a', 'b'], append=True)

In [330]: frame
Out[330]: 
           c    d
c a   b          
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0
```

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 [331]: data.set_index('c', drop=False)
Out[331]: 
     a    b  c    d
c                  
z  bar  one  z  1.0
y  bar  two  y  2.0
x  foo  one  x  3.0
w  foo  two  w  4.0

In [332]: data.set_index(['a', 'b'], inplace=True)

In [333]: data
Out[333]: 
         c    d
a   b          
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

Reset the index

As a convenience, there is a new function on DataFrame called reset_index() which transfers the index values into the DataFrames columns and sets a simple integer index. This is the inverse operation of set_index().

```
In [334]: data
Out[334]: 
         c    d
a   b          
bar one  z  1.0
    two  y  2.0
```

```
foo one  x  3.0
    two  w  4.0

In [335]: data.reset_index()
Out[335]: 
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0
```

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 [336]: frame
Out[336]: 
           c    d
c a   b          
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0

In [337]: frame.reset_index(level=1)
Out[337]: 
         a  c    d
c b               
z one  bar  z  1.0
y two  bar  y  2.0
x one  foo  x  3.0
w two  foo  w  4.0
```

reset_index takes an optional parameter drop which if true simply discards the index, instead of putting index values in the DataFrames columns.

Adding an ad hoc index

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

```
data.index = index
```

4.2.22 Returning a view versus a copy

When setting values in a pandas object, care must be taken to avoid what is called chained indexing. Here is an example.

```
In [338]: dfmi = pd.DataFrame([list('abcd'),
   .....:                      list('efgh'),
   .....:                      list('ijkl'),
   .....:                      list('mnop')],
   .....:                     columns=pd.MultiIndex.from_product([['one', 'two'],
   .....:                                                      ['first', 'second
↪']]))
```

(continues on next page)

(continued from previous page)

```
   .....:

In [339]: dfmi
Out[339]: 
    one          two       
  first second first second
0     a      b     c      d
1     e      f     g      h
2     i      j     k      l
3     m      n     o      p
```

Compare these two access methods:

```
In [340]: dfmi['one']['second']
Out[340]: 
0    b
1    f
2    j
3    n
Name: second, dtype: object
```

```
In [341]: dfmi.loc[:, ('one', 'second')]
Out[341]: 
0    b
1    f
2    j
3    n
Name: (one, second), dtype: object
```

These both yield the same results, so which should you use? It is instructive to understand the order of operations on these and why method 2 (`.loc`) is much preferred over method 1 (chained `[]`).

`dfmi['one']` selects the first level of the columns and returns a DataFrame that is singly-indexed. Then another Python operation `dfmi_with_one['second']` selects the series indexed by `'second'`. This is indicated by the variable `dfmi_with_one` because pandas sees these operations as separate events. e.g. separate calls to `__getitem__`, so it has to treat them as linear operations, they happen one after another.

Contrast this to `df.loc[:,('one','second')]` which passes a nested tuple of `(slice(None),('one', 'second'))` to a single call to `__getitem__`. This allows pandas to deal with this as a single entity. Furthermore this order of operations *can* be significantly faster, and allows one to index *both* axes if so desired.

Why does assignment fail when using chained indexing?

The problem in the previous section is just a performance issue. Whats up with the `SettingWithCopy` warning? We dont **usually** throw warnings around when you do something that might cost a few extra milliseconds!

But it turns out that assigning to the product of chained indexing has inherently unpredictable results. To see this, think about how the Python interpreter executes this code:

```
dfmi.loc[:, ('one', 'second')] = value
# becomes
dfmi.loc.__setitem__((slice(None), ('one', 'second')), value)
```

But this code is handled differently:

```
dfmi['one']['second'] = value
# becomes
dfmi.__getitem__('one').__setitem__('second', value)
```

See that `__getitem__` in there? Outside of simple cases, its very hard to predict whether it will return a view or a copy (it depends on the memory layout of the array, about which pandas makes no guarantees), and therefore whether the `__setitem__` will modify `dfmi` or a temporary object that gets thrown out immediately afterward. **Thats** what `SettingWithCopy` is warning you about!

Note: You may be wondering whether we should be concerned about the `loc` property in the first example. But `dfmi.loc` is guaranteed to be `dfmi` itself with modified indexing behavior, so `dfmi.loc.__getitem__` / `dfmi.loc.__setitem__` operate on `dfmi` directly. Of course, `dfmi.loc.__getitem__(idx)` may be a view or a copy of `dfmi`.

Sometimes a `SettingWithCopy` warning will arise at times when theres no obvious chained indexing going on. **These** are the bugs that `SettingWithCopy` is designed to catch! Pandas is probably trying to warn you that youve done this:

```
def do_something(df):
    foo = df[['bar', 'baz']]  # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...
    # We don't know whether this will modify df or not!
    foo['quux'] = value
    return foo
```

Yikes!

Evaluation order matters

When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.

Pandas has the `SettingWithCopyWarning` because assigning to a copy of a slice is frequently not intentional, but a mistake caused by chained indexing returning a copy where a slice was expected.

If you would like pandas to be more or less trusting about assignment to a chained indexing expression, you can set the *option* `mode.chained_assignment` to one of these values:

- `'warn'`, the default, means a `SettingWithCopyWarning` is printed.
- `'raise'` means pandas will raise a `SettingWithCopyException` you have to deal with.
- `None` will suppress the warnings entirely.

```
In [342]: dfb = pd.DataFrame({'a': ['one', 'one', 'two',
   .....:                           'three', 'two', 'one', 'six'],
   .....:                     'c': np.arange(7)})
   .....:

# This will show the SettingWithCopyWarning
# but the frame values will be set
In [343]: dfb['c'][dfb.a.str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```
>>> pd.set_option('mode.chained_assignment','warn')
>>> dfb[dfb.a.str.startswith('o')]['c'] = 42
Traceback (most recent call last)
     ...
SettingWithCopyWarning:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

A chained assignment can also crop up in setting in a mixed dtype frame.

Note: These setting rules apply to all of `.loc/.iloc`.

This is the correct access method:

```
In [344]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [345]: dfc.loc[0, 'A'] = 11

In [346]: dfc
Out[346]:
     A  B
0   11  1
1  bbb  2
2  ccc  3
```

This *can* work at times, but it is not guaranteed to, and therefore should be avoided:

```
In [347]: dfc = dfc.copy()

In [348]: dfc['A'][0] = 111

In [349]: dfc
Out[349]:
     A  B
0  111  1
1  bbb  2
2  ccc  3
```

This will **not** work at all, and so should be avoided:

```
>>> pd.set_option('mode.chained_assignment','raise')
>>> dfc.loc[0]['A'] = 1111
Traceback (most recent call last)
     ...
SettingWithCopyException:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

Warning: The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.

{{ header }}

4.3 MultiIndex / advanced indexing

This section covers *indexing with a MultiIndex* and *other advanced indexing features*.

See the *Indexing and Selecting Data* for general indexing documentation.

> **Warning:** Whether a copy or a reference is returned for a setting operation may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

See the *cookbook* for some advanced strategies.

4.3.1 Hierarchical indexing (MultiIndex)

Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like `Series` (1d) and `DataFrame` (2d).

In this section, we will show what exactly we mean by hierarchical indexing and how it integrates with all of the pandas indexing functionality described above and in prior sections. Later, when discussing *group by* and *pivoting and reshaping data*, well show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the *cookbook* for some advanced strategies.

Changed in version 0.24.0: `MultiIndex.labels` has been renamed to `MultiIndex.codes` and `MultiIndex.set_labels` to `MultiIndex.set_codes`.

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` as an array of tuples where each tuple is unique. A `MultiIndex` can be created from a list of arrays (using `MultiIndex.from_arrays()`), an array of tuples (using `MultiIndex.from_tuples()`), a crossed set of iterables (using `MultiIndex.from_product()`), or a `DataFrame` (using `MultiIndex.from_frame()`). The `Index` constructor will attempt to return a `MultiIndex` when it is passed a list of tuples. The following examples demonstrate different ways to initialize MultiIndexes.

```
In [1]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ...:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ...:

In [2]: tuples = list(zip(*arrays))

In [3]: tuples
Out[3]:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]
```

(continues on next page)

(continued from previous page)

```
In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [5]: index
Out[5]:
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')],
           names=['first', 'second'])

In [6]: s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]:
first  second
bar    one      -1.386024
       two      -0.148663
baz    one      -2.607819
       two      -1.005003
foo    one       1.204157
       two       0.866884
qux    one      -0.284878
       two       1.160953
dtype: float64
```

When you want every pairing of the elements in two iterables, it can be easier to use the `MultiIndex.from_product()` method:

```
In [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])
Out[9]:
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')],
           names=['first', 'second'])
```

You can also construct a `MultiIndex` from a `DataFrame` directly, using the method `MultiIndex.from_frame()`. This is a complementary method to `MultiIndex.to_frame()`.

New in version 0.24.0.

```
In [10]: df = pd.DataFrame([['bar', 'one'], ['bar', 'two'],
   ....:                     ['foo', 'one'], ['foo', 'two']],
   ....:                    columns=['first', 'second'])
   ....:
```

(continues on next page)

(continued from previous page)

```
In [11]: pd.MultiIndex.from_frame(df)
Out[11]:
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('foo', 'one'),
            ('foo', 'two')],
           names=['first', 'second'])
```

As a convenience, you can pass a list of arrays directly into `Series` or `DataFrame` to construct a `MultiIndex` automatically:

```
In [12]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
   ....:           np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
   ....:

In [13]: s = pd.Series(np.random.randn(8), index=arrays)

In [14]: s
Out[14]:
bar  one    1.678004
     two    0.957360
baz  one    0.866120
     two   -0.348147
foo  one    0.234443
     two    1.279886
qux  one   -0.079828
     two   -0.398588
dtype: float64

In [15]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)

In [16]: df
Out[16]:
                0         1         2         3
bar one -0.479845 -1.066579  0.304694 -0.092407
    two -1.470428 -1.023536  0.513919  0.429166
baz one -0.869518  0.054924 -0.379843 -0.815517
    two -1.025934 -0.748570  0.938954  1.472919
foo one  1.725907  1.384912  0.795282 -0.576135
    two -1.383644 -0.854899 -0.888839 -1.298461
qux one -0.227461  2.302300  1.989458  0.867323
    two -0.785156 -0.647147 -0.146113 -0.754742
```

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 [17]: df.index.names
Out[17]: FrozenList([None, None])
```

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

```
In [18]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'],␣
↪columns=index)

In [19]: df
Out[19]:
first        bar                 baz                 foo                 qux
```

```
second       one       two       one       two       one       two       one ␣
↪      two
A      -0.134175  0.225218 -0.338687 -1.142207 -0.981045 -0.474525  0.857580 -
↪0.161414
B       1.188426 -0.286643 -1.643376 -0.739265  0.841529 -0.443695 -2.770279 -
↪0.947253
C       0.866885  0.954773  2.018904 -0.650307 -0.347872  1.068422 -0.721383 ␣
↪1.184454

In [20]: pd.DataFrame(np.random.randn(6, 6), index=index[:6],␣
↪columns=index[:6])
Out[20]:
first              bar                 baz                 foo
second             one       two       one       two       one       two
first second
bar   one     2.254090  0.665628  0.209523 -1.872699  0.277580  0.842965
      two    -0.690984 -0.336425  0.688883  0.848639  0.005431  0.997845
baz   one     0.386518  0.763158 -0.864808 -1.670655 -1.049607  0.028482
      two    -1.761586  0.487875  0.131290 -0.262495 -0.324123 -0.108842
foo   one     2.343179  0.747274 -0.456691 -0.014325 -0.397298 -0.242768
      two     1.318750 -1.119168  1.061544 -2.664141 -0.600815  0.500161
```

Weve sparsified the higher levels of the indexes to make the console output a bit easier on the eyes. Note that how the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_options()`:

```
In [21]: with pd.option_context('display.multi_sparse', False):
   ....:     df
   ....:
```

Its worth keeping in mind that theres nothing preventing you from using tuples as atomic labels on an axis:

```
In [22]: pd.Series(np.random.randn(8), index=tuples)
Out[22]:
(bar, one)   -1.253707
(bar, two)   -1.439613
(baz, one)    0.024850
(baz, two)    0.166637
(foo, one)   -0.962691
(foo, two)   -1.527951
(qux, one)    0.456415
(qux, two)    0.323368
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.

Reconstructing the level labels

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

```
In [23]: index.get_level_values(0)
Out[23]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],␣
↪dtype='object', name='first')
```

```
In [24]: index.get_level_values('second')
Out[24]: Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'],␣
↪dtype='object', name='second')
```

Basic indexing on axis with MultiIndex

One of the important features of hierarchical indexing is that you can select data by a partial label identifying a subgroup in the data. **Partial** selection drops levels of the hierarchical index in the result in a completely analogous way to selecting a column in a regular DataFrame:

```
In [25]: df['bar']
Out[25]:
second       one       two
A      -0.134175  0.225218
B       1.188426 -0.286643
C       0.866885  0.954773

In [26]: df['bar', 'one']
Out[26]:
A   -0.134175
B    1.188426
C    0.866885
Name: (bar, one), dtype: float64

In [27]: df['bar']['one']
Out[27]:
A   -0.134175
B    1.188426
C    0.866885
Name: one, dtype: float64

In [28]: s['qux']
Out[28]:
one   -0.079828
two   -0.398588
dtype: float64
```

See *Cross-section with hierarchical index* for how to select on a deeper level.

Defined levels

The `MultiIndex` keeps all the defined levels of an index, even if they are not actually used. When slicing an index, you may notice this. For example:

```
In [29]: df.columns.levels  # original MultiIndex
Out[29]: FrozenList([['bar', 'baz', 'foo', 'qux'], ['one', 'two']])

In [30]: df[['foo','qux']].columns.levels  # sliced
Out[30]: FrozenList([['bar', 'baz', 'foo', 'qux'], ['one', 'two']])
```

This is done to avoid a recomputation of the levels in order to make slicing highly performant. If you want to see only the used levels, you can use the `get_level_values()` method.

```
In [31]: df[['foo', 'qux']].columns.to_numpy()
Out[31]:
```

```
array([('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')],
      dtype=object)

# for a specific level
In [32]: df[['foo', 'qux']].columns.get_level_values(0)
Out[32]: Index(['foo', 'foo', 'qux', 'qux'], dtype='object', name='first')
```

To reconstruct the `MultiIndex` with only the used levels, the `remove_unused_levels()` method may be used.

New in version 0.20.0.

```
In [33]: new_mi = df[['foo', 'qux']].columns.remove_unused_levels()

In [34]: new_mi.levels
Out[34]: FrozenList([['foo', 'qux'], ['one', 'two']])
```

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 [35]: s + s[:-2]
Out[35]:
bar  one    3.356008
     two    1.914720
baz  one    1.732240
     two   -0.696293
foo  one    0.468886
     two    2.559772
qux  one         NaN
     two         NaN
dtype: float64

In [36]: s + s[::2]
Out[36]:
bar  one    3.356008
     two         NaN
baz  one    1.732240
     two         NaN
foo  one    0.468886
     two         NaN
qux  one   -0.159656
     two         NaN
dtype: float64
```

The `reindex()` method of `Series`/`DataFrames` can be called with another `MultiIndex`, or even a list or array of tuples:

```
In [37]: s.reindex(index[:3])
Out[37]:
first  second
bar    one       1.678004
       two       0.957360
baz    one       0.866120
dtype: float64
```

```
In [38]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz',␣
↪'one')])
Out[38]:
foo  two    1.279886
bar  one    1.678004
qux  one   -0.079828
baz  one    0.866120
dtype: float64
```

4.3.2 Advanced indexing with hierarchical index

Syntactically integrating `MultiIndex` in advanced indexing with `.loc` is a bit challenging, but weve made every effort to do so. In general, MultiIndex keys take the form of tuples. For example, the following works as you would expect:

```
In [39]: df = df.T

In [40]: df
Out[40]:
                     A         B         C
first second
bar   one    -0.134175  1.188426  0.866885
      two     0.225218 -0.286643  0.954773
baz   one    -0.338687 -1.643376  2.018904
      two    -1.142207 -0.739265 -0.650307
foo   one    -0.981045  0.841529 -0.347872
      two    -0.474525 -0.443695  1.068422
qux   one     0.857580 -2.770279 -0.721383
      two    -0.161414 -0.947253  1.184454

In [41]: df.loc[('bar', 'two')]
Out[41]:
A    0.225218
B   -0.286643
C    0.954773
Name: (bar, two), dtype: float64
```

Note that `df.loc['bar', 'two']` would also work in this example, but this shorthand notation can lead to ambiguity in general.

If you also want to index a specific column with `.loc`, you must use a tuple like this:

```
In [42]: df.loc[('bar', 'two'), 'A']
Out[42]: 0.22521771909947486
```

You dont have to specify all levels of the `MultiIndex` by passing only the first elements of the tuple. For example, you can use partial indexing to get all elements with `bar` in the first level as follows:

df.loc[bar]

This is a shortcut for the slightly more verbose notation `df.loc[('bar',),]` (equivalent to `df.loc['bar',]` in this example).

Partial slicing also works quite nicely.

```
In [43]: df.loc['baz':'foo']
Out[43]: 
                     A         B         C
first second                              
baz   one    -0.338687 -1.643376  2.018904
      two    -1.142207 -0.739265 -0.650307
foo   one    -0.981045  0.841529 -0.347872
      two    -0.474525 -0.443695  1.068422
```

You can slice with a range of values, by providing a slice of tuples.

```
In [44]: df.loc[('baz', 'two'):('qux', 'one')]
Out[44]: 
                     A         B         C
first second                              
baz   two    -1.142207 -0.739265 -0.650307
foo   one    -0.981045  0.841529 -0.347872
      two    -0.474525 -0.443695  1.068422
qux   one     0.857580 -2.770279 -0.721383

In [45]: df.loc[('baz', 'two'):'foo']
Out[45]: 
                     A         B         C
first second                              
baz   two    -1.142207 -0.739265 -0.650307
foo   one    -0.981045  0.841529 -0.347872
      two    -0.474525 -0.443695  1.068422
```

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

```
In [46]: df.loc[[('bar', 'two'), ('qux', 'one')]]
Out[46]: 
                     A         B         C
first second                              
bar   two     0.225218 -0.286643  0.954773
qux   one     0.857580 -2.770279 -0.721383
```

Note: It is important to note that tuples and lists are not treated identically in pandas when it comes to indexing. Whereas a tuple is interpreted as one multi-level key, a list is used to specify several keys. Or in other words, tuples go horizontally (traversing levels), lists go vertically (scanning levels).

Importantly, a list of tuples indexes several complete `MultiIndex` keys, whereas a tuple of lists refer to several values within a level:

```
In [47]: s = pd.Series([1, 2, 3, 4, 5, 6],
   ....:               index=pd.MultiIndex.from_product([["A", "B"], ["c",␣
↪"d", "e"]]))
   ....: 

In [48]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
Out[48]: 
A  c    1
B  d    5
dtype: int64
```

```
In [49]: s.loc[(["A", "B"], ["c", "d"])]  # tuple of lists
Out[49]:
A  c    1
   d    2
B  c    4
   d    5
dtype: int64
```

Using slicers

You can slice a `MultiIndex` by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see *Selection by Label*, including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of *that* level. You do not need to specify all the *deeper* levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

> **Warning:** You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing *both* axes, rather than into say the `MultiIndex` for the rows.
>
> You should do this:
>
> ```
> df.loc[(slice('A1', 'A3'), ...), :]             # noqa: E999
> ```
>
> You should **not** do this:
>
> ```
> df.loc[(slice('A1', 'A3'), ...)]                # noqa: E999
> ```

```
In [50]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:

In [51]: miindex = pd.MultiIndex.from_product([mklbl('A', 4),
   ....:                                       mklbl('B', 2),
   ....:                                       mklbl('C', 4),
   ....:                                       mklbl('D', 2)])
   ....:

In [52]: micolumns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'),
   ....:                                         ('b', 'foo'), ('b', 'bah')],
   ....:                                       names=['lvl0', 'lvl1'])
   ....:

In [53]: dfmi = pd.DataFrame(np.arange(len(miindex) * len(micolumns))
   ....:                       .reshape((len(miindex), len(micolumns))),
   ....:                     index=miindex,
   ....:                     columns=micolumns).sort_index().sort_index(axis=1)
   ....:

In [54]: dfmi
Out[54]:
lvl0           a         b
```

(continues on next page)

(continued from previous page)

```
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0    9    8   11   10
         D1   13   12   15   14
      C2 D0   17   16   19   18
...          ...  ...  ...  ...
A3 B1 C1 D1  237  236  239  238
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  249  248  251  250
         D1  253  252  255  254

[64 rows x 4 columns]
```

Basic MultiIndex slicing using slices, lists, and labels.

```
In [55]: dfmi.loc[(slice('A1', 'A3'), slice(None), ['C1', 'C3']), :]
Out[55]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A1 B0 C1 D0   73   72   75   74
         D1   77   76   79   78
      C3 D0   89   88   91   90
         D1   93   92   95   94
   B1 C1 D0  105  104  107  106
         D1  109  108  111  110
      C3 D0  121  120  123  122
         D1  125  124  127  126
A2 B0 C1 D0  137  136  139  138
         D1  141  140  143  142
      C3 D0  153  152  155  154
         D1  157  156  159  158
   B1 C1 D0  169  168  171  170
         D1  173  172  175  174
      C3 D0  185  184  187  186
         D1  189  188  191  190
A3 B0 C1 D0  201  200  203  202
         D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
      C3 D0  249  248  251  250
         D1  253  252  255  254
```

You can use *pandas.IndexSlice* to facilitate a more natural syntax using `:`, rather than using `slice(None)`.

```
In [56]: idx = pd.IndexSlice

In [57]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[57]:
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
```

(continues on next page)

(continued from previous page)

```
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
         D1   60   62
A1 B0 C1 D0   72   74
         D1   76   78
      C3 D0   88   90
         D1   92   94
   B1 C1 D0  104  106
         D1  108  110
      C3 D0  120  122
         D1  124  126
A2 B0 C1 D0  136  138
         D1  140  142
      C3 D0  152  154
         D1  156  158
   B1 C1 D0  168  170
         D1  172  174
      C3 D0  184  186
         D1  188  190
A3 B0 C1 D0  200  202
         D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254
```

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```
In [58]: dfmi.loc['A1', (slice(None), 'foo')]
Out[58]: 
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
      D1   76   78
   C2 D0   80   82
      D1   84   86
   C3 D0   88   90
      D1   92   94
B1 C0 D0   96   98
      D1  100  102
   C1 D0  104  106
      D1  108  110
   C2 D0  112  114
      D1  116  118
   C3 D0  120  122
      D1  124  126

In [59]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[59]: 
lvl0           a    b
```

```
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
         D1   60   62
A1 B0 C1 D0   72   74
         D1   76   78
      C3 D0   88   90
         D1   92   94
   B1 C1 D0  104  106
         D1  108  110
      C3 D0  120  122
         D1  124  126
A2 B0 C1 D0  136  138
         D1  140  142
      C3 D0  152  154
         D1  156  158
   B1 C1 D0  168  170
         D1  172  174
      C3 D0  184  186
         D1  188  190
A3 B0 C1 D0  200  202
         D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254
```

Using a boolean indexer you can provide selection related to the *values*.

```
In [60]: mask = dfmi[('a', 'foo')] > 200

In [61]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[61]: 
lvl0           a    b
lvl1         foo  foo
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```
In [62]: dfmi.loc(axis=0)[:, :, ['C1', 'C3']]
Out[62]: 
lvl0           a         b
```

(continues on next page)

(continued from previous page)

```
lvl1         bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
         D1   13   12   15   14
      C3 D0   25   24   27   26
         D1   29   28   31   30
   B1 C1 D0   41   40   43   42
         D1   45   44   47   46
      C3 D0   57   56   59   58
         D1   61   60   63   62
A1 B0 C1 D0   73   72   75   74
         D1   77   76   79   78
      C3 D0   89   88   91   90
         D1   93   92   95   94
   B1 C1 D0  105  104  107  106
         D1  109  108  111  110
      C3 D0  121  120  123  122
         D1  125  124  127  126
A2 B0 C1 D0  137  136  139  138
         D1  141  140  143  142
      C3 D0  153  152  155  154
         D1  157  156  159  158
   B1 C1 D0  169  168  171  170
         D1  173  172  175  174
      C3 D0  185  184  187  186
         D1  189  188  191  190
A3 B0 C1 D0  201  200  203  202
         D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
      C3 D0  249  248  251  250
         D1  253  252  255  254
```

Furthermore, you can *set* the values using the following methods.

```
In [63]: df2 = dfmi.copy()

In [64]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

In [65]: df2
Out[65]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0   17   16   19   18
...          ...  ...  ...  ...
A3 B1 C1 D1  -10  -10  -10  -10
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10

[64 rows x 4 columns]
```

You can use a right-hand-side of an alignable object as well.

```
In [66]: df2 = dfmi.copy()

In [67]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [68]: df2
Out[68]: 
lvl0              a               b        
lvl1            bar     foo     bah     foo
A0 B0 C0 D0       1       0       3       2
         D1       5       4       7       6
      C1 D0    9000    8000   11000   10000
         D1   13000   12000   15000   14000
      C2 D0      17      16      19      18
...             ...     ...     ...     ...
A3 B1 C1 D1  237000  236000  239000  238000
      C2 D0     241     240     243     242
         D1     245     244     247     246
      C3 D0  249000  248000  251000  250000
         D1  253000  252000  255000  254000

[64 rows x 4 columns]
```

Cross-section

The `xs()` method of `DataFrame` additionally takes a level argument to make selecting data at a particular level of a `MultiIndex` easier.

```
In [69]: df
Out[69]: 
                     A         B         C
first second                              
bar   one    -0.134175  1.188426  0.866885
      two     0.225218 -0.286643  0.954773
baz   one    -0.338687 -1.643376  2.018904
      two    -1.142207 -0.739265 -0.650307
foo   one    -0.981045  0.841529 -0.347872
      two    -0.474525 -0.443695  1.068422
qux   one     0.857580 -2.770279 -0.721383
      two    -0.161414 -0.947253  1.184454

In [70]: df.xs('one', level='second')
Out[70]: 
              A         B         C
first                              
bar   -0.134175  1.188426  0.866885
baz   -0.338687 -1.643376  2.018904
foo   -0.981045  0.841529 -0.347872
qux    0.857580 -2.770279 -0.721383
```

```
# using the slicers
In [71]: df.loc[(slice(None), 'one'), :]
Out[71]: 
                     A         B         C
```

(continues on next page)

(continued from previous page)

```
first second
bar   one    -0.134175  1.188426  0.866885
baz   one    -0.338687 -1.643376  2.018904
foo   one    -0.981045  0.841529 -0.347872
qux   one     0.857580 -2.770279 -0.721383
```

You can also select on the columns with xs, by providing the axis argument.

```
In [72]: df = df.T

In [73]: df.xs('one', level='second', axis=1)
Out[73]:
first        bar       baz       foo       qux
A     -0.134175 -0.338687 -0.981045  0.857580
B      1.188426 -1.643376  0.841529 -2.770279
C      0.866885  2.018904 -0.347872 -0.721383
```

```
# using the slicers
In [74]: df.loc[:, (slice(None), 'one')]
Out[74]:
first         bar       baz       foo       qux
second        one       one       one       one
A       -0.134175 -0.338687 -0.981045  0.857580
B        1.188426 -1.643376  0.841529 -2.770279
C        0.866885  2.018904 -0.347872 -0.721383
```

xs also allows selection with multiple keys.

```
In [75]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out[75]:
first         bar
second        one
A       -0.134175
B        1.188426
C        0.866885
```

```
# using the slicers
In [76]: df.loc[:, ('bar', 'one')]
Out[76]:
A   -0.134175
B    1.188426
C    0.866885
Name: (bar, one), dtype: float64
```

You can pass drop_level=False to xs to retain the level that was selected.

```
In [77]: df.xs('one', level='second', axis=1, drop_level=False)
Out[77]:
first         bar       baz       foo       qux
second        one       one       one       one
A       -0.134175 -0.338687 -0.981045  0.857580
B        1.188426 -1.643376  0.841529 -2.770279
C        0.866885  2.018904 -0.347872 -0.721383
```

Compare the above with the result using drop_level=True (the default value).

```
In [78]: df.xs('one', level='second', axis=1, drop_level=True)
Out[78]:
first        bar       baz       foo       qux
A      -0.134175 -0.338687 -0.981045  0.857580
B       1.188426 -1.643376  0.841529 -2.770279
C       0.866885  2.018904 -0.347872 -0.721383
```

Advanced reindexing and alignment

Using the parameter `level` in the `reindex()` and `align()` methods of pandas objects is useful to broadcast values across a level. For instance:

```
In [79]: midx = pd.MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
   ....:                      codes=[[1, 1, 0, 0], [1, 0, 1, 0]])
   ....:

In [80]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)

In [81]: df
Out[81]:
               0         1
one  y  0.229891 -0.556291
     x  0.790519 -0.207853
zero y -0.182679  0.496230
     x  0.177932  1.885439

In [82]: df2 = df.mean(level=0)

In [83]: df2
Out[83]:
             0         1
one   0.510205 -0.382072
zero -0.002373  1.190835

In [84]: df2.reindex(df.index, level=0)
Out[84]:
               0         1
one  y  0.510205 -0.382072
     x  0.510205 -0.382072
zero y -0.002373  1.190835
     x -0.002373  1.190835

# aligning
In [85]: df_aligned, df2_aligned = df.align(df2, level=0)

In [86]: df_aligned
Out[86]:
               0         1
one  y  0.229891 -0.556291
     x  0.790519 -0.207853
zero y -0.182679  0.496230
     x  0.177932  1.885439
```

```
In [87]: df2_aligned
Out[87]:
               0         1
one  y  0.510205 -0.382072
     x  0.510205 -0.382072
zero y -0.002373  1.190835
     x -0.002373  1.190835
```

Swapping levels with `swaplevel`

The `swaplevel()` method can switch the order of two levels:

```
In [88]: df[:5]
Out[88]:
               0         1
one  y  0.229891 -0.556291
     x  0.790519 -0.207853
zero y -0.182679  0.496230
     x  0.177932  1.885439

In [89]: df[:5].swaplevel(0, 1, axis=0)
Out[89]:
               0         1
y one   0.229891 -0.556291
x one   0.790519 -0.207853
y zero -0.182679  0.496230
x zero  0.177932  1.885439
```

Reordering levels with `reorder_levels`

The `reorder_levels()` method generalizes the `swaplevel` method, allowing you to permute the hierarchical index levels in one step:

```
In [90]: df[:5].reorder_levels([1, 0], axis=0)
Out[90]:
               0         1
y one   0.229891 -0.556291
x one   0.790519 -0.207853
y zero -0.182679  0.496230
x zero  0.177932  1.885439
```

Renaming names of an `Index` or `MultiIndex`

The `rename()` method is used to rename the labels of a `MultiIndex`, and is typically used to rename the columns of a `DataFrame`. The `columns` argument of `rename` allows a dictionary to be specified that includes only the columns you wish to rename.

```
In [91]: df.rename(columns={0: "col0", 1: "col1"})
Out[91]:
            col0      col1
one  y  0.229891 -0.556291
     x  0.790519 -0.207853
```

(continues on next page)

(continued from previous page)

```
zero y -0.182679  0.496230
     x  0.177932  1.885439
```

This method can also be used to rename specific labels of the main index of the DataFrame.

```
In [92]: df.rename(index={"one": "two", "y": "z"})
Out[92]:
                0         1
two  z  0.229891 -0.556291
     x  0.790519 -0.207853
zero z -0.182679  0.496230
     x  0.177932  1.885439
```

The rename_axis() method is used to rename the name of a Index or MultiIndex. In particular, the names of the levels of a MultiIndex can be specified, which is useful if reset_index() is later used to move the values from the MultiIndex to a column.

```
In [93]: df.rename_axis(index=['abc', 'def'])
Out[93]:
                  0         1
abc  def
one  y     0.229891 -0.556291
     x     0.790519 -0.207853
zero y    -0.182679  0.496230
     x     0.177932  1.885439
```

Note that the columns of a DataFrame are an index, so that using rename_axis with the columns argument will change the name of that index.

```
In [94]: df.rename_axis(columns="Cols").columns
Out[94]: RangeIndex(start=0, stop=2, step=1, name='Cols')
```

Both rename and rename_axis support specifying a dictionary, Series or a mapping function to map labels/names to new values.

4.3.3 Sorting a MultiIndex

For MultiIndex-ed objects to be indexed and sliced effectively, they need to be sorted. As with any index, you can use sort_index().

```
In [95]: import random

In [96]: random.shuffle(tuples)

In [97]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_
↪tuples(tuples))

In [98]: s
Out[98]:
baz  one    1.432295
qux  two   -0.999104
foo  one    0.099556
bar  one    0.189457
foo  two    1.070405
baz  two   -0.015460
```

```
qux  one    1.018696
bar  two   -0.966176
dtype: float64

In [99]: s.sort_index()
Out[99]:
bar  one    0.189457
     two   -0.966176
baz  one    1.432295
     two   -0.015460
foo  one    0.099556
     two    1.070405
qux  one    1.018696
     two   -0.999104
dtype: float64

In [100]: s.sort_index(level=0)
Out[100]:
bar  one    0.189457
     two   -0.966176
baz  one    1.432295
     two   -0.015460
foo  one    0.099556
     two    1.070405
qux  one    1.018696
     two   -0.999104
dtype: float64

In [101]: s.sort_index(level=1)
Out[101]:
bar  one    0.189457
baz  one    1.432295
foo  one    0.099556
qux  one    1.018696
bar  two   -0.966176
baz  two   -0.015460
foo  two    1.070405
qux  two   -0.999104
dtype: float64
```

You may also pass a level name to `sort_index` if the `MultiIndex` levels are named.

```
In [102]: s.index.set_names(['L1', 'L2'], inplace=True)

In [103]: s.sort_index(level='L1')
Out[103]:
L1   L2
bar  one    0.189457
     two   -0.966176
baz  one    1.432295
     two   -0.015460
foo  one    0.099556
     two    1.070405
qux  one    1.018696
     two   -0.999104
```

```
dtype: float64

In [104]: s.sort_index(level='L2')
Out[104]: 
L1   L2 
bar  one    0.189457
baz  one    1.432295
foo  one    0.099556
qux  one    1.018696
bar  two   -0.966176
baz  two   -0.015460
foo  two    1.070405
qux  two   -0.999104
dtype: float64
```

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

```
In [105]: df.T.sort_index(level=1, axis=1)
Out[105]: 
        one      zero       one      zero
          x         x         y         y
0  0.790519  0.177932  0.229891 -0.182679
1 -0.207853  1.885439 -0.556291  0.496230
```

Indexing will work even if the data are not sorted, but will be rather inefficient (and show a `PerformanceWarning`). It will also return a copy of the data rather than a view:

```
In [106]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
   .....:                     'joe': ['x', 'x', 'z', 'y'],
   .....:                     'jolie': np.random.rand(4)})
   .....: 

In [107]: dfm = dfm.set_index(['jim', 'joe'])

In [108]: dfm
Out[108]: 
            jolie
jim joe          
0   x    0.238982
    x    0.824351
1   z    0.159509
    y    0.076877
```

```
In [4]: dfm.loc[(1, 'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.

Out[4]:
           jolie
jim joe
1   z    0.64094
```

Furthermore, if you try to index something that is not fully lexsorted, this can raise:

```
In [5]: dfm.loc[(0, 'y'):(1, 'z')]
UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'
```

The `is_lexsorted()` method on a `MultiIndex` shows if the index is sorted, and the `lexsort_depth` property returns the sort depth:

```
In [109]: dfm.index.is_lexsorted()
Out[109]: False

In [110]: dfm.index.lexsort_depth
Out[110]: 1
```

```
In [111]: dfm = dfm.sort_index()

In [112]: dfm
Out[112]: 
            jolie
jim joe          
0   x    0.238982
    x    0.824351
1   y    0.076877
    z    0.159509

In [113]: dfm.index.is_lexsorted()
Out[113]: True

In [114]: dfm.index.lexsort_depth
Out[114]: 2
```

And now selection works as expected.

```
In [115]: dfm.loc[(0, 'y'):(1, 'z')]
Out[115]:
            jolie
jim joe
1   y    0.076877
    z    0.159509
```

4.3.4 Take methods

Similar to NumPy ndarrays, pandas `Index`, `Series`, and `DataFrame` also provides the `take()` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [116]: index = pd.Index(np.random.randint(0, 1000, 10))

In [117]: index
Out[117]: Int64Index([951, 384, 317, 822, 525, 60, 525, 328, 210, 945],␣
↪dtype='int64')

In [118]: positions = [0, 9, 3]

In [119]: index[positions]
Out[119]: Int64Index([951, 945, 822], dtype='int64')

In [120]: index.take(positions)
Out[120]: Int64Index([951, 945, 822], dtype='int64')

In [121]: ser = pd.Series(np.random.randn(10))
In [122]: ser.iloc[positions]
Out[122]:
0    0.776615
9   -1.120902
3   -0.232684
dtype: float64

In [123]: ser.take(positions)
Out[123]:
0    0.776615
9   -1.120902
3   -0.232684
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [124]: frm = pd.DataFrame(np.random.randn(5, 3))

In [125]: frm.take([1, 4, 3])
Out[125]:
          0         1         2
1  0.122843 -0.001117 -0.571039
4 -1.630543  0.053506 -1.312211
3 -1.131866  1.675784 -0.420701

In [126]: frm.take([0, 2], axis=1)
Out[126]:
          0         2
0 -0.420297  0.331846
1  0.122843 -0.571039
2  1.182088  1.695155
3 -1.131866 -0.420701
4 -1.630543 -1.312211
```

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 [127]: arr = np.random.randn(10)

In [128]: arr.take([False, False, True, True])
Out[128]: array([ 1.62593494,  1.62593494, -0.90445976, -0.90445976])

In [129]: arr[[0, 1]]
Out[129]: array([ 1.62593494, -0.90445976])

In [130]: ser = pd.Series(np.random.randn(10))

In [131]: ser.take([False, False, True, True])
Out[131]:
0    1.186759
0    1.186759
1    1.194373
1    1.194373
dtype: float64
```

```
In [132]: ser.iloc[[0, 1]]
Out[132]:
0    1.186759
1    1.194373
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.

```
In [133]: arr = np.random.randn(10000, 5)

In [134]: indexer = np.arange(10000)

In [135]: random.shuffle(indexer)

In [136]: %timeit arr[indexer]
   .....: %timeit arr.take(indexer, axis=0)
   .....:
200 us +- 3.32 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
76.6 us +- 1.26 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

```
In [137]: ser = pd.Series(arr[:, 0])

In [138]: %timeit ser.iloc[indexer]
   .....: %timeit ser.take(indexer)
   .....:
117 us +- 2.31 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
106 us +- 1.87 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

4.3.5 Index types

We have discussed MultiIndex in the previous sections pretty extensively. Documentation about DatetimeIndex and PeriodIndex are shown *here*, and documentation about TimedeltaIndex is found *here*.

In the following sub-sections we will highlight some other index types.

CategoricalIndex

CategoricalIndex is a type of index that is useful for supporting indexing with duplicates. This is a container around a Categorical and allows efficient indexing and storage of an index with a large number of duplicated elements.

```
In [139]: from pandas.api.types import CategoricalDtype

In [140]: df = pd.DataFrame({'A': np.arange(6),
   .....:                    'B': list('aabbca')})
   .....:

In [141]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))

In [142]: df
Out[142]:
   A  B
0  0  a
```

```
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [143]: df.dtypes
Out[143]:
A        int64
B     category
dtype: object

In [144]: df.B.cat.categories
Out[144]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a `CategoricalIndex`.

```
In [145]: df2 = df.set_index('B')

In [146]: df2.index
Out[146]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
↪ ordered=False, name='B', dtype='category')
```

Indexing with `__getitem__/.iloc/.loc` works similarly to an `Index` with duplicates. The indexers **must** be in the category or the operation will raise a `KeyError`.

```
In [147]: df2.loc['a']
Out[147]:
   A
B
a  0
a  1
a  5
```

The `CategoricalIndex` is **preserved** after indexing:

```
In [148]: df2.loc['a'].index
Out[148]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
↪ name='B', dtype='category')
```

Sorting the index will sort by the order of the categories (recall that we created the index with `CategoricalDtype(list('cab'))`, so the sorted order is `cab`).

```
In [149]: df2.sort_index()
Out[149]:
   A
B
c  4
a  0
a  1
a  5
b  2
b  3
```

Groupby operations on the index will preserve the index nature as well.

```
In [150]: df2.groupby(level=0).sum()
Out[150]:
   A
B
c  4
a  6
b  5

In [151]: df2.groupby(level=0).sum().index
Out[151]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'],␣
↪ordered=False, name='B', dtype='category')
```

Reindexing operations will return a resulting index based on the type of the passed indexer. Passing a list will return a plain-old `Index`; indexing with a `Categorical` will return a `CategoricalIndex`, indexed according to the categories of the **passed** `Categorical` dtype. This allows one to arbitrarily index these even with values **not** in the categories, similarly to how you can reindex **any** pandas index.

```
In [152]: df2.reindex(['a', 'e'])
Out[152]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [153]: df2.reindex(['a', 'e']).index
Out[153]: Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [154]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde')))
Out[154]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [155]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde'))).
↪index
Out[155]: CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c',␣
↪'d', 'e'], ordered=False, name='B', dtype='category')
```

Warning: Reshaping and Comparison operations on a `CategoricalIndex` must have the same categories or a `TypeError` will be raised.

```
In [9]: df3 = pd.DataFrame({'A': np.arange(6), 'B': pd.Series(list('aabbca')).
↪astype('category')})

In [11]: df3 = df3.set_index('B')

In [11]: df3.index
Out[11]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['a', 'b', 'c
↪'], ordered=False, name='B', dtype='category')

In [12]: pd.concat([df2, df3])
TypeError: categories must match existing categories when appending
```

Int64Index and RangeIndex

Warning: Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see *here*.

`Int64Index` is a fundamental basic index in pandas. This is an immutable array implementing an ordered, sliceable set. Prior to 0.18.0, the `Int64Index` would provide the default index for all `NDFrame` objects.

`RangeIndex` is a sub-class of `Int64Index` added in version 0.18.0, now providing the default index for all `NDFrame` objects. `RangeIndex` is an optimized version of `Int64Index` that can represent a monotonic ordered set. These are analogous to Python range types.

Float64Index

By default a `Float64Index` will be automatically created when passing floating, or mixed-integer-floating values in index creation. This enables a pure label-based slicing paradigm that makes `[],ix,loc` for scalar indexing and slicing work exactly the same.

```
In [156]: indexf = pd.Index([1.5, 2, 3, 4.5, 5])

In [157]: indexf
Out[157]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [158]: sf = pd.Series(range(5), index=indexf)

In [159]: sf
Out[159]: 
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
dtype: int64
```

Scalar selection for `[],.loc` will always be label based. An integer will match an equal float index (e.g. `3` is equivalent to `3.0`).

```
In [160]: sf[3]
Out[160]: 2

In [161]: sf[3.0]
Out[161]: 2

In [162]: sf.loc[3]
Out[162]: 2

In [163]: sf.loc[3.0]
Out[163]: 2
```

The only positional indexing is via `iloc`.

```
In [164]: sf.iloc[3]
Out[164]: 3
```

A scalar index that is not found will raise a `KeyError`. Slicing is primarily on the values of the index when using `[],ix,loc`, and **always** positional when using `iloc`. The exception is when the slice is boolean, in which case it will always be positional.

```
In [165]: sf[2:4]
Out[165]:
2.0    1
3.0    2
dtype: int64

In [166]: sf.loc[2:4]
Out[166]:
2.0    1
3.0    2
dtype: int64

In [167]: sf.iloc[2:4]
Out[167]:
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```
In [168]: sf[2.1:4.6]
Out[168]:
3.0    2
4.5    3
dtype: int64

In [169]: sf.loc[2.1:4.6]
Out[169]:
3.0    2
4.5    3
dtype: int64
```

In non-float indexes, slicing using floats will raise a `TypeError`.

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

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

Warning: Using a scalar float indexer for `.iloc` has been removed in 0.18.0, so the following will raise a `TypeError`:

```
In [3]: pd.Series(range(5)).iloc[3.0]
TypeError: cannot do positional indexing on <class 'pandas.indexes.range.RangeIndex
↪'> with these indexers [3.0] of <type 'float'>
```

Here is a typical use-case for using this type of indexing. Imagine that you have a somewhat irregular timedelta-like indexing scheme, but the data is recorded as floats. This could, for example, be millisecond offsets.

```
In [170]: dfir = pd.concat([pd.DataFrame(np.random.randn(5, 2),
   .....:                                index=np.arange(5) * 250.0,
   .....:                                columns=list('AB')),
   .....:                   pd.DataFrame(np.random.randn(6, 2),
   .....:                                index=np.arange(4, 10) * 250.1,
   .....:                                columns=list('AB'))])
   .....:

In [171]: dfir
Out[171]:
               A         B
0.0    -1.902297 -0.933899
250.0   0.035744 -0.338195
500.0  -1.851855  0.635651
750.0  -0.207053  0.419207
1000.0  0.337604  1.324050
1000.4 -0.623756  0.066847
1250.5  0.597810  0.597494
1500.6 -0.028424 -1.160777
1750.7 -1.591904 -1.183292
2000.8  0.868619  0.417332
2250.9  0.265403 -0.379946
```

Selection operations then will always work on a value basis, for all selection operators.

```
In [172]: dfir[0:1000.4]
Out[172]:
               A         B
0.0    -1.902297 -0.933899
250.0   0.035744 -0.338195
500.0  -1.851855  0.635651
750.0  -0.207053  0.419207
1000.0  0.337604  1.324050
1000.4 -0.623756  0.066847

In [173]: dfir.loc[0:1001, 'A']
Out[173]:
0.0      -1.902297
250.0     0.035744
500.0    -1.851855
750.0    -0.207053
1000.0    0.337604
1000.4   -0.623756
Name: A, dtype: float64

In [174]: dfir.loc[1000.4]
Out[174]:
A   -0.623756
B    0.066847
Name: 1000.4, dtype: float64
```

You could retrieve the first 1 second (1000 ms) of data as such:

```
In [175]: dfir[0:1000]
Out[175]:
               A         B
```

(continues on next page)

(continued from previous page)

```
0.0    -1.902297 -0.933899
250.0   0.035744 -0.338195
500.0  -1.851855  0.635651
750.0  -0.207053  0.419207
1000.0  0.337604  1.324050
```

If you need integer based selection, you should use `iloc`:

```
In [176]: dfir.iloc[0:5]
Out[176]:
               A         B
0.0    -1.902297 -0.933899
250.0   0.035744 -0.338195
500.0  -1.851855  0.635651
750.0  -0.207053  0.419207
1000.0  0.337604  1.324050
```

IntervalIndex

New in version 0.20.0.

`IntervalIndex` together with its own dtype, `IntervalDtype` as well as the `Interval` scalar type, allow first-class support in pandas for interval notation.

The `IntervalIndex` allows some unique indexing and is also used as a return type for the categories in `cut()` and `qcut()`.

Indexing with an `IntervalIndex`

An `IntervalIndex` can be used in `Series` and in `DataFrame` as the index.

```
In [177]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
   .....:                   index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
   .....:

In [178]: df
Out[178]:
        A
(0, 1]  1
(1, 2]  2
(2, 3]  3
(3, 4]  4
```

Label based indexing via `.loc` along the edges of an interval works as you would expect, selecting that particular interval.

```
In [179]: df.loc[2]
Out[179]:
A    2
Name: (1, 2], dtype: int64

In [180]: df.loc[[2, 3]]
Out[180]:
        A
```

```
(1, 2]  2
(2, 3]  3
```

If you select a label *contained* within an interval, this will also select the interval.

```
In [181]: df.loc[2.5]
Out[181]:
A    3
Name: (2, 3], dtype: int64

In [182]: df.loc[[2.5, 3.5]]
Out[182]:
        A
(2, 3]  3
(3, 4]  4
```

Selecting using an `Interval` will only return exact matches (starting from pandas 0.25.0).

```
In [183]: df.loc[pd.Interval(1, 2)]
Out[183]:
A    2
Name: (1, 2], dtype: int64
```

Trying to select an `Interval` that is not exactly contained in the `IntervalIndex` will raise a `KeyError`.

```
In [7]: df.loc[pd.Interval(0.5, 2.5)]
---------------------------------------------------------------------------
KeyError: Interval(0.5, 2.5, closed='right')
```

Selecting all `Intervals` that overlap a given `Interval` can be performed using the `overlaps()` method to create a boolean indexer.

```
In [184]: idxr = df.index.overlaps(pd.Interval(0.5, 2.5))

In [185]: idxr
Out[185]: array([ True,  True,  True, False])

In [186]: df[idxr]
Out[186]:
        A
(0, 1]  1
(1, 2]  2
(2, 3]  3
```

Binning data with `cut` and `qcut`

`cut()` and `qcut()` both return a `Categorical` object, and the bins they create are stored as an `IntervalIndex` in its `.categories` attribute.

```
In [187]: c = pd.cut(range(4), bins=2)

In [188]: c
Out[188]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
In [189]: c.categories
Out[189]:
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]],
              closed='right',
              dtype='interval[float64]')
```

`cut()` also accepts an `IntervalIndex` for its `bins` argument, which enables a useful pandas idiom. First, We call `cut()` with some data and `bins` set to a fixed number, to generate the bins. Then, we pass the values of `.categories` as the `bins` argument in subsequent calls to `cut()`, supplying new data which will be binned into the same bins.

```
In [190]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[190]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
```

Any value which falls outside all bins will be assigned a `NaN` value.

Generating ranges of intervals

If we need intervals on a regular frequency, we can use the `interval_range()` function to create an `IntervalIndex` using various combinations of `start`, `end`, and `periods`. The default frequency for `interval_range` is a 1 for numeric intervals, and calendar day for datetime-like intervals:

```
In [191]: pd.interval_range(start=0, end=5)
Out[191]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right',
              dtype='interval[int64]')

In [192]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4)
Out[192]:
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03], (2017-01-
↪03, 2017-01-04], (2017-01-04, 2017-01-05]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [193]: pd.interval_range(end=pd.Timedelta('3 days'), periods=3)
Out[193]:
IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:00:00, 2 days␣
↪00:00:00], (2 days 00:00:00, 3 days 00:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

The `freq` parameter can used to specify non-default frequencies, and can utilize a variety of *frequency aliases* with datetime-like intervals:

```
In [194]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[194]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]],
              closed='right',
              dtype='interval[float64]')

In [195]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4,␣
↪freq='W')
```

```
Out[195]:
IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-
↪15, 2017-01-22], (2017-01-22, 2017-01-29]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [196]: pd.interval_range(start=pd.Timedelta('0 days'), periods=3,␣
↪freq='9H')
Out[196]:
IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days␣
↪18:00:00], (0 days 18:00:00, 1 days 03:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

Additionally, the `closed` parameter can be used to specify which side(s) the intervals are closed on. Intervals are closed on the right side by default.

```
In [197]: pd.interval_range(start=0, end=4, closed='both')
Out[197]:
IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]],
              closed='both',
              dtype='interval[int64]')

In [198]: pd.interval_range(start=0, end=4, closed='neither')
Out[198]:
IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)],
              closed='neither',
              dtype='interval[int64]')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced intervals from `start` to `end` inclusively, with `periods` number of elements in the resulting `IntervalIndex`:

```
In [199]: pd.interval_range(start=0, end=6, periods=4)
Out[199]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
              closed='right',
              dtype='interval[float64]')

In [200]: pd.interval_range(pd.Timestamp('2018-01-01'),
   .....:                   pd.Timestamp('2018-02-28'), periods=3)
   .....:
Out[200]:
IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-
↪02-08 16:00:00], (2018-02-08 16:00:00, 2018-02-28]],
              closed='right',
              dtype='interval[datetime64[ns]]')
```

4.3.6 Miscellaneous indexing FAQ

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 `.loc`. The following code will generate exceptions:

```
In [201]: s = pd.Series(range(5))

In [202]: s[-1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-202-76c3dce40054> in <module>
----> 1 s[-1]

~/sandbox/pandas-release/pandas/pandas/core/series.py in __getitem__(self,␣
↪key)
   1069         key = com.apply_if_callable(key, self)
   1070         try:
-> 1071             result = self.index.get_value(self, key)
   1072 
   1073             if not is_scalar(result):

~/sandbox/pandas-release/pandas/pandas/core/indexes/base.py in get_value(self,
↪ series, key)
   4728         k = self._convert_scalar_indexer(k, kind="getitem")
   4729         try:
-> 4730             return self._engine.get_value(s, k, tz=getattr(series.
↪dtype, "tz", None))
   4731         except KeyError as e1:
   4732             if len(self) > 0 and (self.holds_integer() or self.is_
↪boolean()):

~/sandbox/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.
↪IndexEngine.get_value()

~/sandbox/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.
↪IndexEngine.get_value()

~/sandbox/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.
↪IndexEngine.get_loc()

~/sandbox/pandas-release/pandas/pandas/_libs/hashtable_class_helper.pxi in␣
↪pandas._libs.hashtable.Int64HashTable.get_item()

~/sandbox/pandas-release/pandas/pandas/_libs/hashtable_class_helper.pxi in␣
↪pandas._libs.hashtable.Int64HashTable.get_item()

KeyError: -1

In [203]: df = pd.DataFrame(np.random.randn(5, 4))

In [204]: df
Out[204]: 
          0         1         2         3
0  0.066370  0.481162 -0.247207  0.021250
1  0.768621 -0.053824  1.219183  2.211600
2 -0.381577 -0.068826 -1.112808 -0.976648
```

```
3 -0.177430 -1.029073  0.118622  0.872648
4 -1.240728  1.519051 -0.122180 -0.665184

In [205]: df.loc[-2:]
Out[205]:
          0         1         2         3
0  0.066370  0.481162 -0.247207  0.021250
1  0.768621 -0.053824  1.219183  2.211600
2 -0.381577 -0.068826 -1.112808 -0.976648
3 -0.177430 -1.029073  0.118622  0.872648
4 -1.240728  1.519051 -0.122180 -0.665184
```

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).

Non-monotonic indexes require exact matches

If the index of a `Series` or `DataFrame` is monotonically increasing or decreasing, then the bounds of a label-based slice can be outside the range of the index, much like slice indexing a normal Python `list`. Monotonicity of an index can be tested with the `is_monotonic_increasing()` and `is_monotonic_decreasing()` attributes.

```
In [206]: df = pd.DataFrame(index=[2, 3, 3, 4, 5], columns=['data'],␣
↪data=list(range(5)))

In [207]: df.index.is_monotonic_increasing
Out[207]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [208]: df.loc[0:4, :]
Out[208]:
   data
2     0
3     1
3     2
4     3

# slice is are outside the index, so empty DataFrame is returned
In [209]: df.loc[13:15, :]
Out[209]:
Empty DataFrame
Columns: [data]
Index: []
```

On the other hand, if the index is not monotonic, then both slice bounds must be *unique* members of the index.

```
In [210]: df = pd.DataFrame(index=[2, 3, 1, 4, 3, 5],
   .....:                   columns=['data'], data=list(range(6)))
   .....:

In [211]: df.index.is_monotonic_increasing
Out[211]: False

# OK because 2 and 4 are in the index
In [212]: df.loc[2:4, :]
Out[212]:
   data
2     0
3     1
1     2
4     3
```

```
# 0 is not in the index
In [9]: df.loc[0:4, :]
KeyError: 0

# 3 is not a unique label
In [11]: df.loc[2:3, :]
KeyError: 'Cannot get right slice bound for non-unique label: 3'
```

`Index.is_monotonic_increasing` and `Index.is_monotonic_decreasing` only check that an index is weakly monotonic. To check for strict monotonicity, you can combine one of those with the `is_unique()` attribute.

```
In [213]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])

In [214]: weakly_monotonic
Out[214]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [215]: weakly_monotonic.is_monotonic_increasing
Out[215]: True

In [216]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_
↪unique
Out[216]: False
```

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 [217]: s = pd.Series(np.random.randn(6), index=list('abcdef'))

In [218]: s
Out[218]:
a    0.895739
b    1.349538
c    0.040227
d    1.154586
e    0.162300
f   -1.213270
dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be accomplished as such:

```
In [219]: s[2:5]
Out[219]:
c    0.040227
d    1.154586
e    0.162300
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.loc['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 choice to make label-based slicing include both endpoints:

```
In [220]: s.loc['c':'e']
Out[220]: 
c    0.040227
d    1.154586
e    0.162300
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.

Indexing potentially changes underlying Series dtype

The different indexing operation can potentially change the dtype of a Series.

```
In [221]: series1 = pd.Series([1, 2, 3])

In [222]: series1.dtype
Out[222]: dtype('int64')

In [223]: res = series1.reindex([0, 4])

In [224]: res.dtype
Out[224]: dtype('float64')

In [225]: res
Out[225]: 
0    1.0
4    NaN
dtype: float64
```

```
In [226]: series2 = pd.Series([True])

In [227]: series2.dtype
Out[227]: dtype('bool')

In [228]: res = series2.reindex_like(series1)

In [229]: res.dtype
Out[229]: dtype('O')

In [230]: res
Out[230]: 
0    True
1     NaN
2     NaN
dtype: object
```

This is because the (re)indexing operations above 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. {{ header }}

4.4 Merge, join, and concatenate

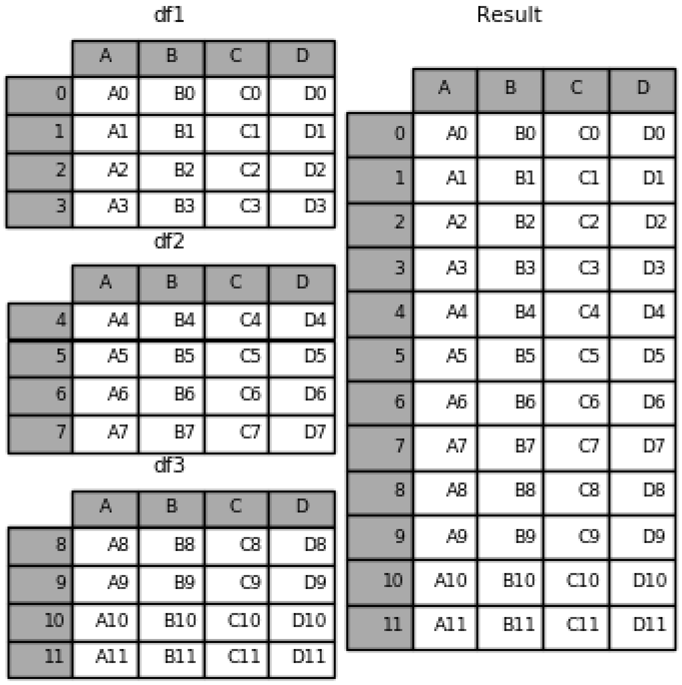
pandas provides various facilities for easily combining together Series or DataFrame with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

4.4.1 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]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ...:                     'B': ['B0', 'B1', 'B2', 'B3'],
   ...:                     'C': ['C0', 'C1', 'C2', 'C3'],
   ...:                     'D': ['D0', 'D1', 'D2', 'D3']},
   ...:                    index=[0, 1, 2, 3])
   ...:

In [2]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
   ...:                     'B': ['B4', 'B5', 'B6', 'B7'],
   ...:                     'C': ['C4', 'C5', 'C6', 'C7'],
   ...:                     'D': ['D4', 'D5', 'D6', 'D7']},
   ...:                    index=[4, 5, 6, 7])
   ...:

In [3]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
   ...:                     'B': ['B8', 'B9', 'B10', 'B11'],
   ...:                     'C': ['C8', 'C9', 'C10', 'C11'],
   ...:                     'D': ['D8', 'D9', 'D10', 'D11']},
   ...:                    index=[8, 9, 10, 11])
   ...:

In [4]: frames = [df1, df2, df3]

In [5]: result = pd.concat(frames)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

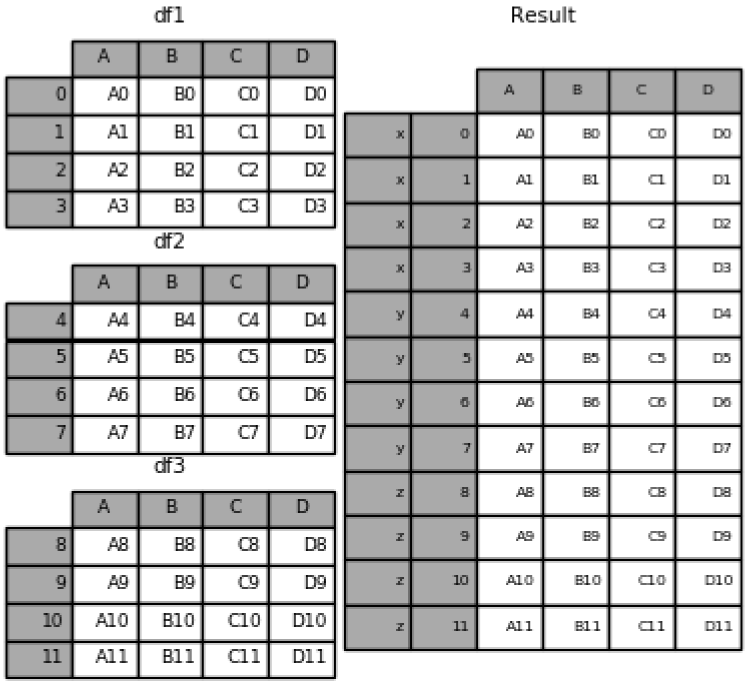
Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of what to do with the other axes:

```
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
          levels=None, names=None, verify_integrity=False, copy=True)
```

- `objs` : a sequence or mapping of Series or DataFrame objects. If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.
- `axis` : {0, 1, }, default 0. The axis to concatenate along.
- `join` : {inner, outer}, default outer. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- `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. Note the index values on the other axes are still respected in the join.
- `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. 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.
- `verify_integrity` : boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.
- `copy` : boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments dont make much sense. Lets revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the `keys` argument:

```
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

As you can see (if youve read the rest of the documentation), the resulting objects index has a *hierarchical index*. This means that we can now select out each chunk by key:

```
In [7]: result.loc['y']
Out[7]:
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

Its not a stretch to see how this can be very useful. More detail on this functionality below.

Note: It is worth noting that *concat()* (and therefore `append()`) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

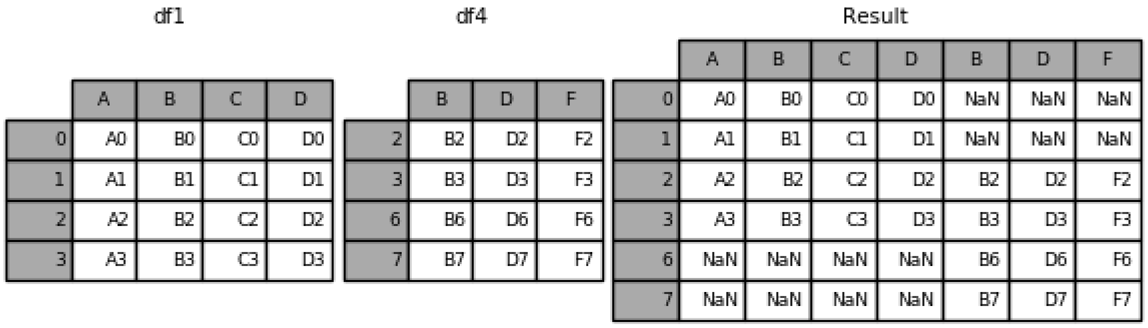
Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following two ways:

- Take the union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.

Here is an example of each of these methods. First, the default `join='outer'` behavior:

```
In [8]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
   ...:                     'D': ['D2', 'D3', 'D6', 'D7'],
   ...:                     'F': ['F2', 'F3', 'F6', 'F7']},
   ...:                    index=[2, 3, 6, 7])
   ...:

In [9]: result = pd.concat([df1, df4], axis=1, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

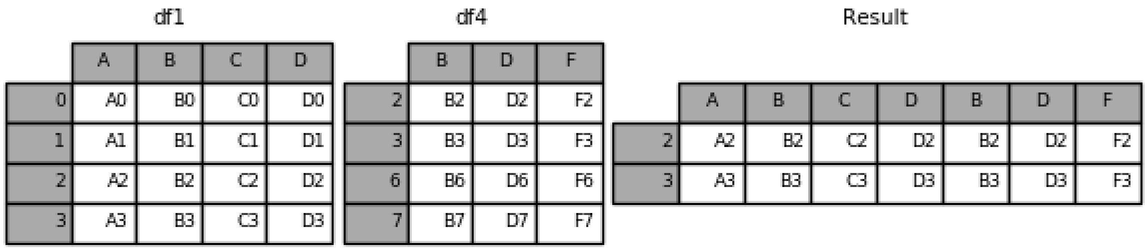
	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Warning: Changed in version 0.23.0.

The default behavior with `join='outer'` is to sort the other axis (columns in this case). In a future version of pandas, the default will be to not sort. We specified `sort=False` to opt in to the new behavior now.

Here is the same thing with `join='inner'`:

```
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

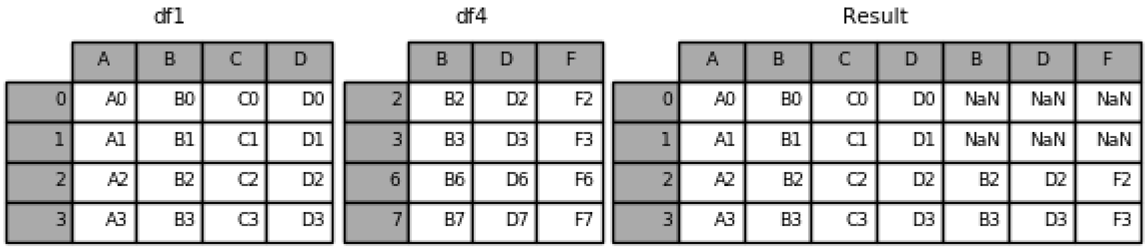
	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

```
In [11]: result = pd.concat([df1, df4], axis=1).reindex(df1.index)
```

Similarly, we could index before the concatenation:

```
In [12]: pd.concat([df1, df4.reindex(df1.index)], axis=1)
Out[12]:
    A   B   C   D    B    D    F
0  A0  B0  C0  D0  NaN  NaN  NaN
1  A1  B1  C1  D1  NaN  NaN  NaN
2  A2  B2  C2  D2   B2   D2   F2
3  A3  B3  C3  D3   B3   D3   F3
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

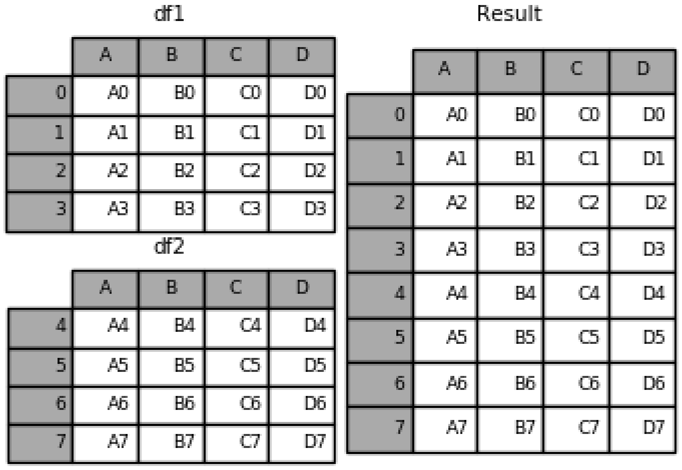
Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

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 [13]: result = df1.append(df2)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

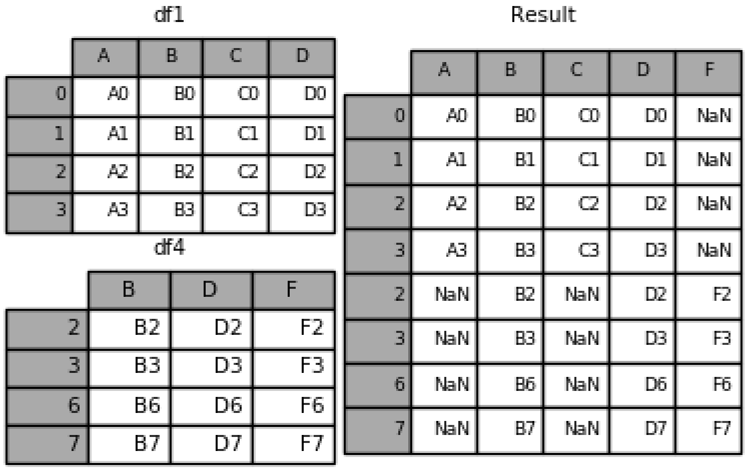
	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In the case of `DataFrame`, the indexes must be disjoint but the columns do not need to be:

```
In [14]: result = df1.append(df4, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

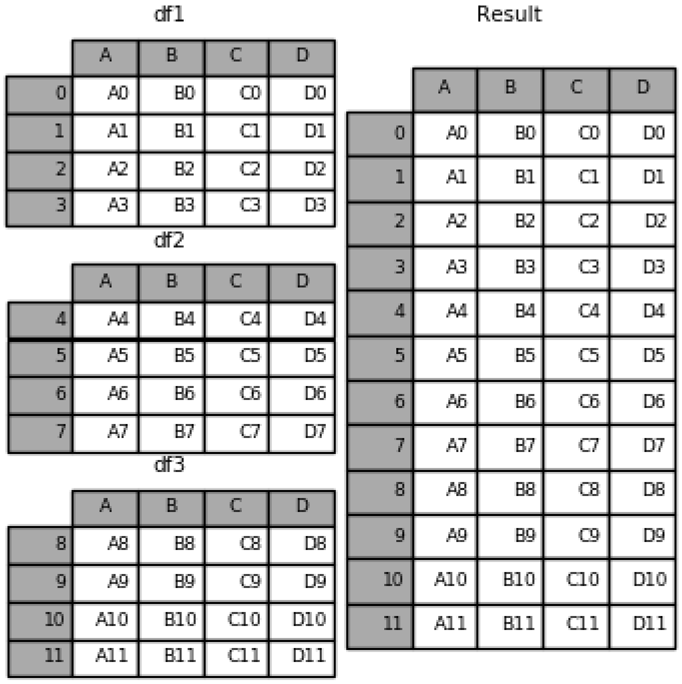
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

append may take multiple objects to concatenate:

```
In [15]: result = df1.append([df2, df3])
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

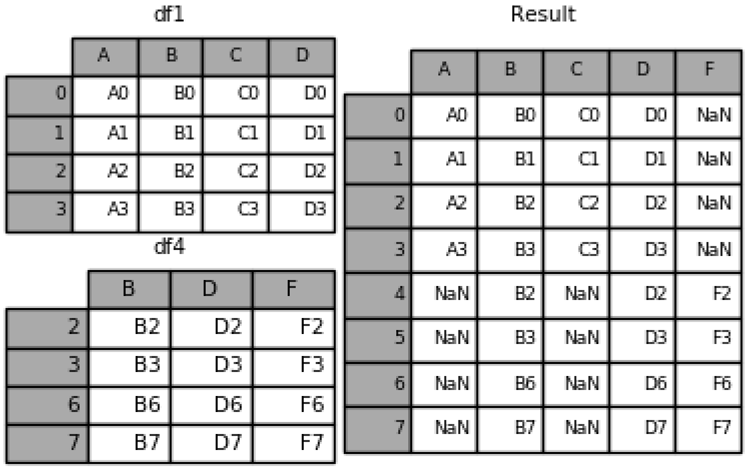
	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Note: Unlike the append() method, which appends to the original list and returns None, append() here **does not** modify df1 and returns its copy with df2 appended.

Ignoring indexes on the concatenation axis

For `DataFrame` objects which dont have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the `ignore_index` argument:

```
In [16]: result = pd.concat([df1, df4], ignore_index=True, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

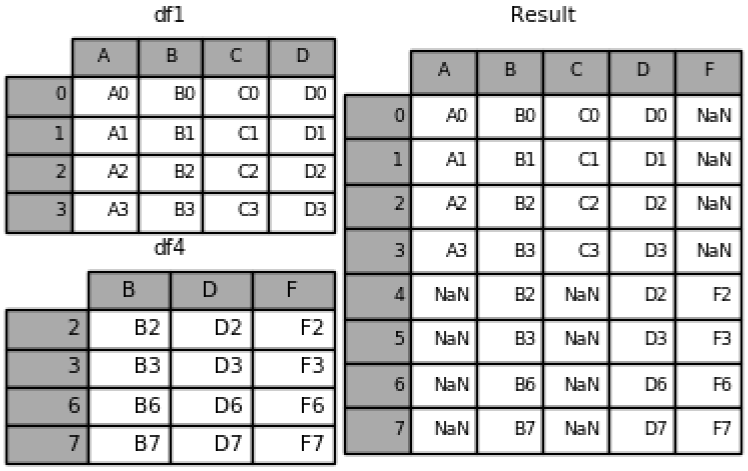
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

This is also a valid argument to `DataFrame.append()`:

```
In [17]: result = df1.append(df4, ignore_index=True, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

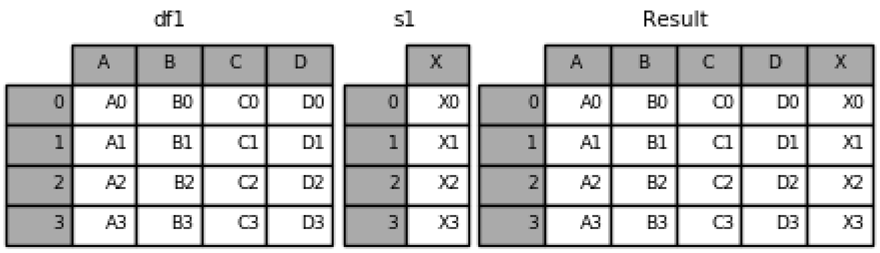
Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Concatenating with mixed ndims

You can concatenate a mix of `Series` and `DataFrame` objects. The `Series` will be transformed to `DataFrame` with the column name as the name of the `Series`.

```
In [18]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')

In [19]: result = pd.concat([df1, s1], axis=1)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

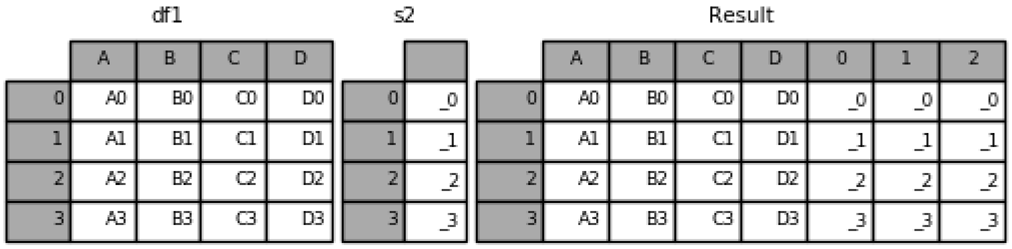
Result

	A	B	C	D	X
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

Note: Since were concatenating a `Series` to a `DataFrame`, we could have achieved the same result with `DataFrame.assign()`. To concatenate an arbitrary number of pandas objects (`DataFrame` or `Series`), use `concat`.

If unnamed `Series` are passed they will be numbered consecutively.

```
In [20]: s2 = pd.Series(['_0', '_1', '_2', '_3'])

In [21]: result = pd.concat([df1, s2, s2, s2], axis=1)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

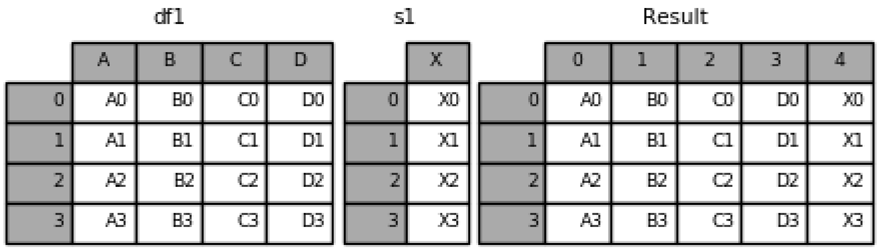
0	_0
1	_1
2	_2
3	_3

Result

	A	B	C	D	0	1	2
0	A0	B0	C0	D0	_0	_0	_0
1	A1	B1	C1	D1	_1	_1	_1
2	A2	B2	C2	D2	_2	_2	_2
3	A3	B3	C3	D3	_3	_3	_3

Passing `ignore_index=True` will drop all name references.

```
In [22]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

Result

	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

More concatenating with group keys

A fairly common use of the keys argument is to override the column names when creating a new DataFrame based on existing Series. Notice how the default behaviour consists on letting the resulting DataFrame inherit the parent Series name, when these existed.

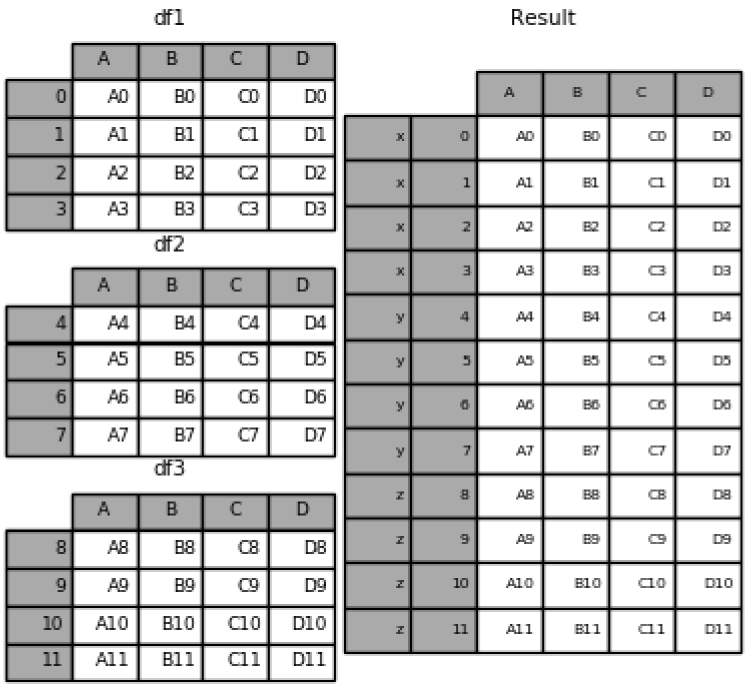
```
In [23]: s3 = pd.Series([0, 1, 2, 3], name='foo')

In [24]: s4 = pd.Series([0, 1, 2, 3])

In [25]: s5 = pd.Series([0, 1, 4, 5])

In [26]: pd.concat([s3, s4, s5], axis=1)
Out[26]:
   foo  0  1
0    0  0  0
1    1  1  1
2    2  2  4
3    3  3  5
```

Through the keys argument we can override the existing column names.

```
In [27]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
Out[27]:
   red  blue  yellow
0    0     0       0
1    1     1       1
2    2     2       4
3    3     3       5
```

Lets consider a variation of the very first example presented:

```
In [28]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

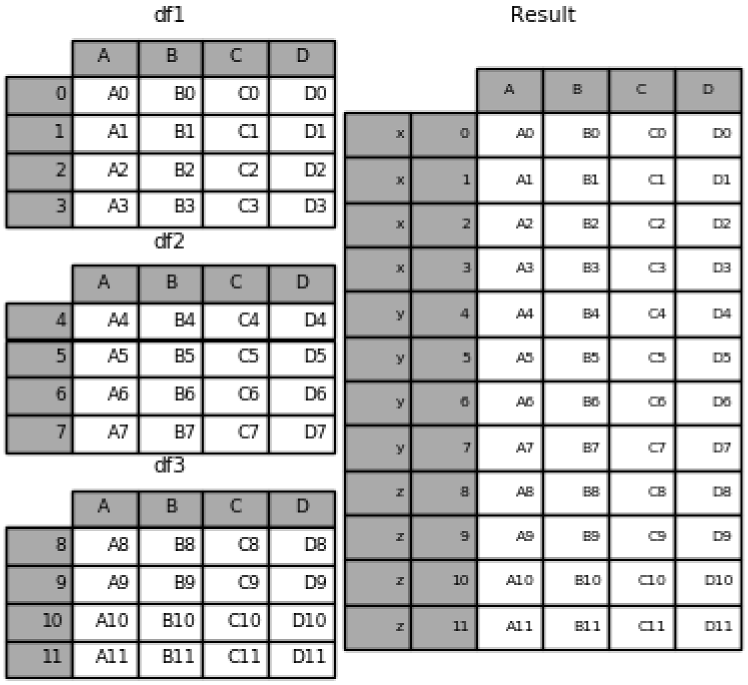
	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

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 [29]: pieces = {'x': df1, 'y': df2, 'z': df3}

In [30]: result = pd.concat(pieces)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

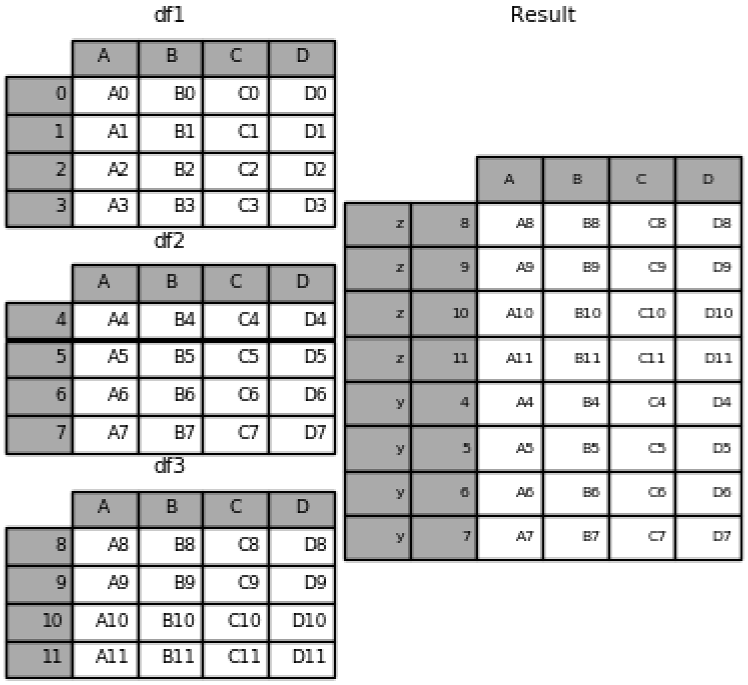
df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

```
In [31]: result = pd.concat(pieces, keys=['z', 'y'])
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

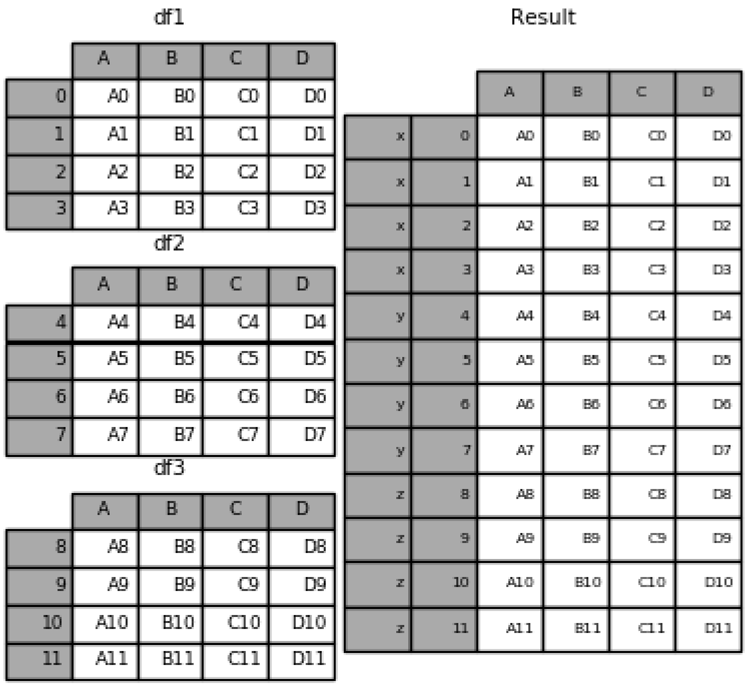
		A	B	C	D
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7

The MultiIndex created has levels that are constructed from the passed keys and the index of the `DataFrame` pieces:

```
In [32]: result.index.levels
Out[32]: FrozenList([['z', 'y'], [4, 5, 6, 7, 8, 9, 10, 11]])
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

```
In [33]: result = pd.concat(pieces, keys=['x', 'y', 'z'],
   ....:                    levels=[['z', 'y', 'x', 'w']],
   ....:                    names=['group_key'])
   ....:
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

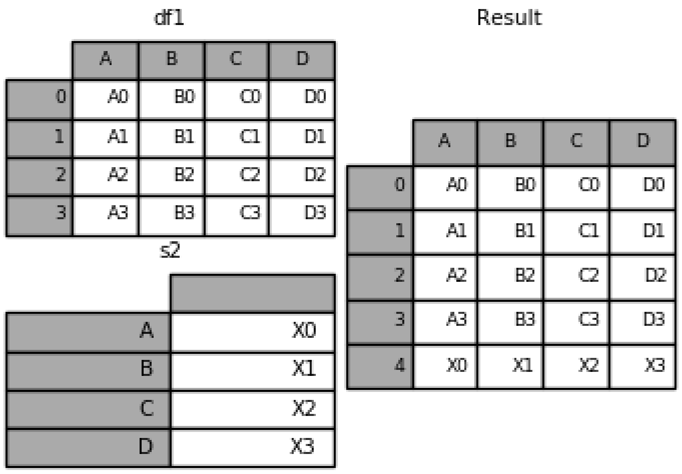
```
In [34]: result.index.levels
Out[34]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
```

This is fairly esoteric, but it is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

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 [35]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])

In [36]: result = df1.append(s2, ignore_index=True)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

A	X0
B	X1
C	X2
D	X3

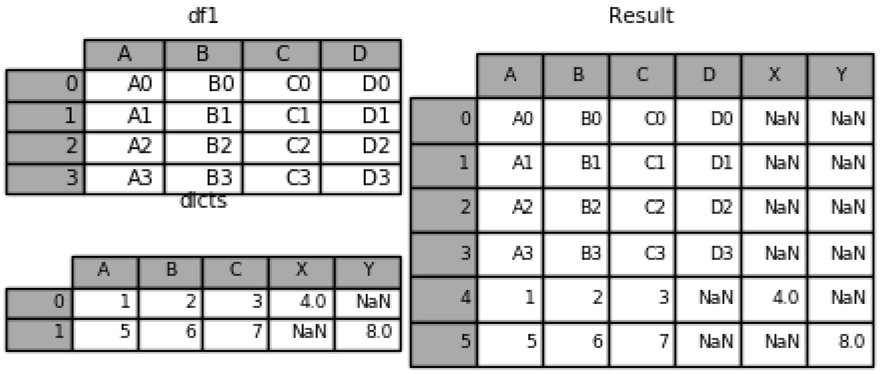
Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	X0	X1	X2	X3

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 [37]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
   ....:          {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]
   ....:

In [38]: result = df1.append(dicts, ignore_index=True, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

dicts

	A	B	C	X	Y
0	1	2	3	4.0	NaN
1	5	6	7	NaN	8.0

Result

	A	B	C	D	X	Y
0	A0	B0	C0	D0	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN
4	1	2	3	NaN	4.0	NaN
5	5	6	7	NaN	NaN	8.0

4.4.2 Database-style DataFrame or named Series 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 the internal layout of the data in `DataFrame`.

See the *cookbook* for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a *comparison with SQL*.

pandas provides a single function, *merge()*, as the entry point for all standard database join operations between `DataFrame` or named `Series` objects:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
```

- `left`: A DataFrame or named Series object.
- `right`: Another DataFrame or named Series object.
- `on`: Column or index level names to join on. Must be found in both the left and right DataFrame and/or Series objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the DataFrames and/or Series will be inferred to be the join keys.
- `left_on`: Columns or index levels from the left DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.
- `right_on`: Columns or index levels from the right DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.
- `left_index`: If `True`, use the index (row labels) from the left DataFrame or Series as its join key(s). In the case of a DataFrame or Series with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame or Series.
- `right_index`: Same usage as `left_index` for the right DataFrame or Series
- `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 or named Series 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.
- `indicator`: Add a column to the output DataFrame called `_merge` with information on the source of each row. `_merge` is Categorical-type and takes on a value of `left_only` for observations whose merge key only appears in `'left'` DataFrame or Series, `right_only` for observations whose merge key only appears in `'right'` DataFrame or Series, and `both` if the observations merge key is found in both.
- `validate` : string, default None. If specified, checks if merge is of specified type.
  - one_to_one or 1:1: checks if merge keys are unique in both left and right datasets.
  - one_to_many or 1:m: checks if merge keys are unique in left dataset.
  - many_to_one or m:1: checks if merge keys are unique in right dataset.
  - many_to_many or m:m: allowed, but does not result in checks.

  New in version 0.21.0.

Note: Support for specifying index levels as the `on`, `left_on`, and `right_on` parameters was added in version 0.23.0. Support for merging named `Series` objects was added in version 0.24.0.

The return type will be the same as `left`. If `left` is a `DataFrame` or named `Series` and `right` is a subclass of `DataFrame`, the return type will still be `DataFrame`.

`merge` is a function in the pandas namespace, and it is also available as a `DataFrame` instance method `merge()`, with the calling `DataFrame` being implicitly considered the left object in the join.

The related `join()` method, uses `merge` internally for the index-on-index (by default) and column(s)-on-index join. If you are joining on index only, you may wish to use `DataFrame.join` to save yourself some typing.

Brief primer on merge methods (relational algebra)

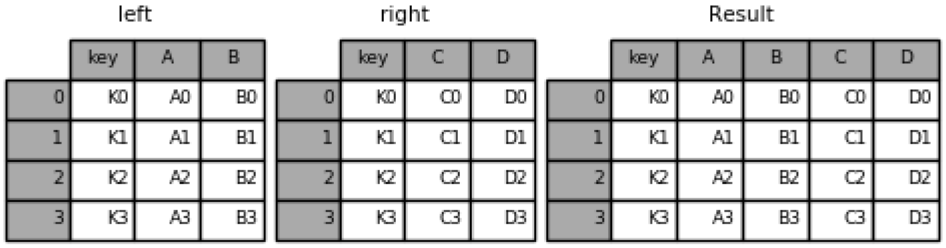
Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (`DataFrame` objects). There are several cases to consider which are very important to understand:

- **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 different `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 [39]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:

In [40]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [41]: result = pd.merge(left, right, on='key')
```



left

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

right

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

Result

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

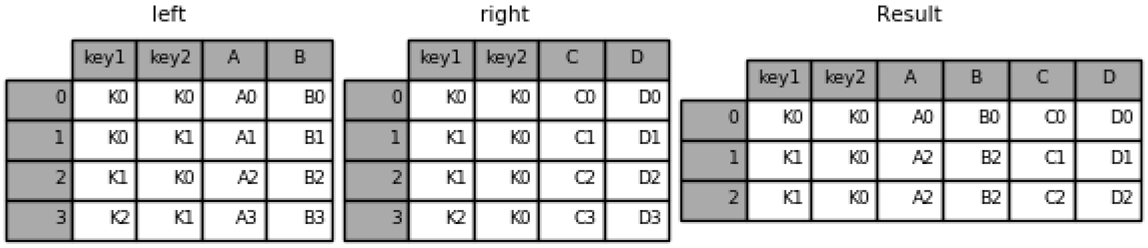
Here is a more complicated example with multiple join keys. Only the keys appearing in `left` and `right` are present (the intersection), since `how='inner'` by default.

```
In [42]: left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [43]: right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
   ....:                       'key2': ['K0', 'K0', 'K0', 'K0'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [44]: result = pd.merge(left, right, on=['key1', 'key2'])
```



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

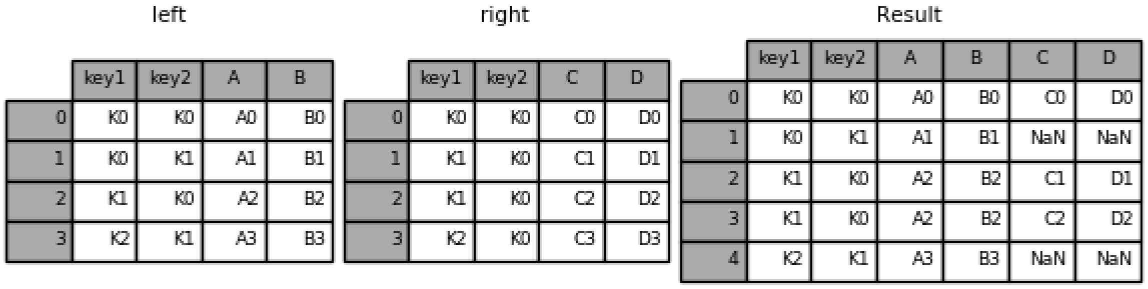
Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

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

```
In [45]: result = pd.merge(left, right, how='left', on=['key1', 'key2'])
```



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

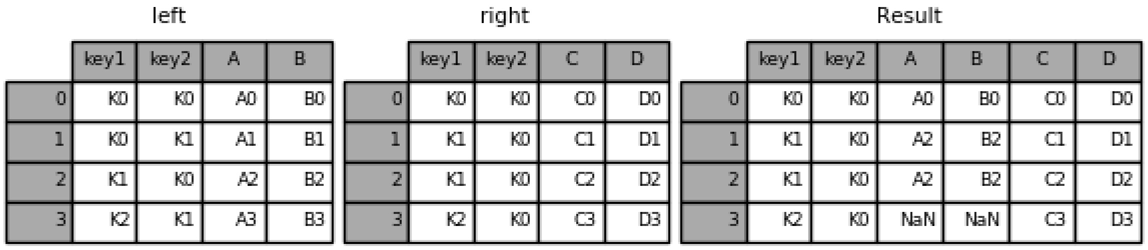
right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

```
In [46]: result = pd.merge(left, right, how='right', on=['key1', 'key2'])
```



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

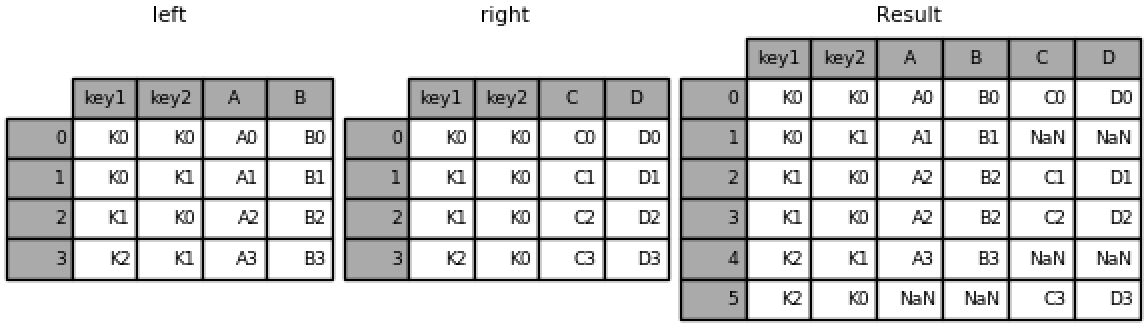
right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K2	K0	NaN	NaN	C3	D3

```
In [47]: result = pd.merge(left, right, how='outer', on=['key1', 'key2'])
```



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

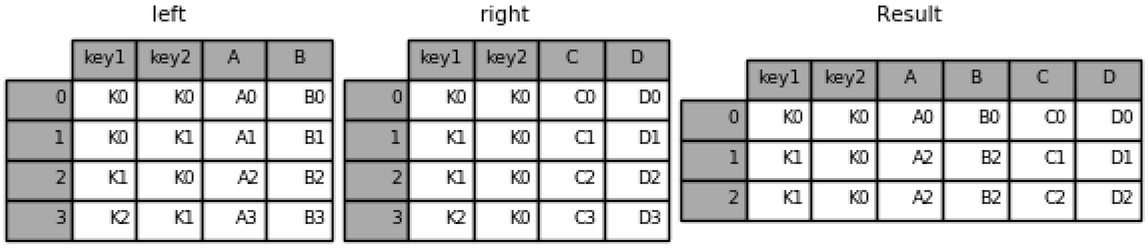
right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

```
In [48]: result = pd.merge(left, right, how='inner', on=['key1', 'key2'])
```



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

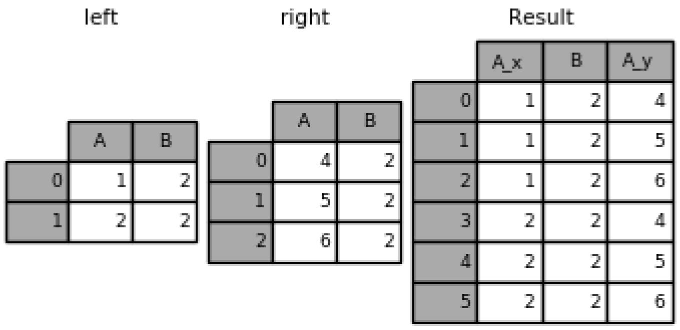
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

Here is another example with duplicate join keys in DataFrames:

```
In [49]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})

In [50]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})

In [51]: result = pd.merge(left, right, on='B', how='outer')
```



left

	A	B
0	1	2
1	2	2

right

	A	B
0	4	2
1	5	2
2	6	2

Result

	A_x	B	A_y
0	1	2	4
1	1	2	5
2	1	2	6
3	2	2	4
4	2	2	5
5	2	2	6

Warning: Joining / merging on duplicate keys can cause a returned frame that is the multiplication of the row dimensions, which may result in memory overflow. It is the user s responsibility to manage duplicate values in keys before joining large DataFrames.

Checking for duplicate keys

New in version 0.21.0.

Users can use the `validate` argument to automatically check whether there are unexpected duplicates in their merge keys. Key uniqueness is checked before merge operations and so should protect against memory overflows. Checking key uniqueness is also a good way to ensure user data structures are as expected.

In the following example, there are duplicate values of `B` in the right `DataFrame`. As this is not a one-to-one merge – as specified in the `validate` argument – an exception will be raised.

```
In [52]: left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})

In [53]: right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2, 2]})
```

```
In [53]: result = pd.merge(left, right, on='B', how='outer', validate="one_to_one")
...
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right `DataFrame` but wants to ensure there are no duplicates in the left DataFrame, one can use the `validate='one_to_many'` argument instead, which will not raise an exception.

```
In [54]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[54]:
   A_x  B  A_y
0    1  1  NaN
1    2  2  4.0
2    2  2  5.0
3    2  2  6.0
```

The merge indicator

`merge()` accepts the argument `indicator`. If `True`, a Categorical-type column called `_merge` will be added to the output object that takes on values:

Observation Origin	_merge value
Merge key only in 'left' frame	left_only
Merge key only in 'right' frame	right_only
Merge key in both frames	both

```
In [55]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})

In [56]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})

In [57]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[57]:
   col1 col_left  col_right      _merge
0     0        a        NaN   left_only
1     1        b        2.0        both
2     2      NaN        2.0  right_only
3     2      NaN        2.0  right_only
```

The indicator argument will also accept string arguments, in which case the indicator function will use the value of the passed string as the name for the indicator column.

```
In [58]: pd.merge(df1, df2, on='col1', how='outer', indicator='indicator_column')
Out[58]:
   col1 col_left  col_right indicator_column
0     0        a        NaN        left_only
1     1        b        2.0             both
2     2      NaN        2.0       right_only
3     2      NaN        2.0       right_only
```

Merge dtypes

New in version 0.19.0.

Merging will preserve the dtype of the join keys.

```
In [59]: left = pd.DataFrame({'key': [1], 'v1': [10]})

In [60]: left
Out[60]:
   key  v1
0    1  10

In [61]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [62]: right
Out[62]:
   key  v1
0    1  20
1    2  30
```

We are able to preserve the join keys:

```
In [63]: pd.merge(left, right, how='outer')
Out[63]:
   key  v1
0    1  10
```

```
1    1  20
2    2  30

In [64]: pd.merge(left, right, how='outer').dtypes
Out[64]:
key    int64
v1     int64
dtype: object
```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.

```
In [65]: pd.merge(left, right, how='outer', on='key')
Out[65]:
   key  v1_x  v1_y
0    1  10.0    20
1    2   NaN    30

In [66]: pd.merge(left, right, how='outer', on='key').dtypes
Out[66]:
key       int64
v1_x    float64
v1_y      int64
dtype: object
```

New in version 0.20.0.

Merging will preserve `category` dtypes of the mergands. See also the section on *categoricals*.

The left frame.

```
In [67]: from pandas.api.types import CategoricalDtype

In [68]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [69]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))

In [70]: left = pd.DataFrame({'X': X,
   ....:                      'Y': np.random.choice(['one', 'two', 'three'],
   ....:                                            size=(10,))})
   ....:

In [71]: left
Out[71]:
     X      Y
0  foo  three
1  bar  three
2  foo    two
3  foo  three
4  foo    one
5  foo    one
6  bar    one
7  foo    two
8  foo    one
9  bar    two

In [72]: left.dtypes
Out[72]:
```

```
X    category
Y      object
dtype: object
```

The right frame.

```
In [73]: right = pd.DataFrame({'X': pd.Series(['foo', 'bar'],
   ....:                                       dtype=CategoricalDtype(['foo',␣
 ↪'bar'])),
   ....:                       'Z': [1, 2]})
   ....:

In [74]: right
Out[74]: 
     X  Z
0  foo  1
1  bar  2

In [75]: right.dtypes
Out[75]: 
X    category
Z       int64
dtype: object
```

The merged result:

```
In [76]: result = pd.merge(left, right, how='outer')

In [77]: result
Out[77]: 
     X      Y  Z
0  foo  three  1
1  foo    two  1
2  foo  three  1
3  foo    one  1
4  foo    one  1
5  foo    two  1
6  foo    one  1
7  bar  three  2
8  bar    one  2
9  bar    two  2

In [78]: result.dtypes
Out[78]: 
X    category
Y      object
Z       int64
dtype: object
```

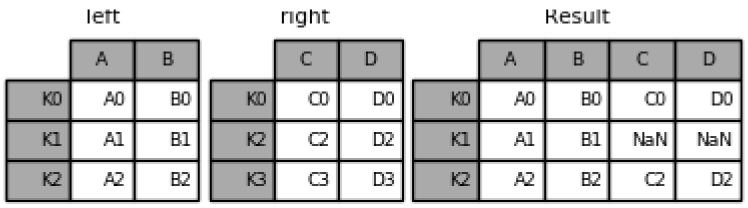
Note: The category dtypes must be *exactly* the same, meaning the same categories and the ordered attribute. Otherwise the result will coerce to `object` dtype.

Note: Merging on `category` dtypes that are the same can be quite performant compared to `object` dtype merging.

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 [79]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                     index=['K0', 'K1', 'K2'])
   ....:

In [80]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D2', 'D3']},
   ....:                      index=['K0', 'K2', 'K3'])
   ....:

In [81]: result = left.join(right)
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

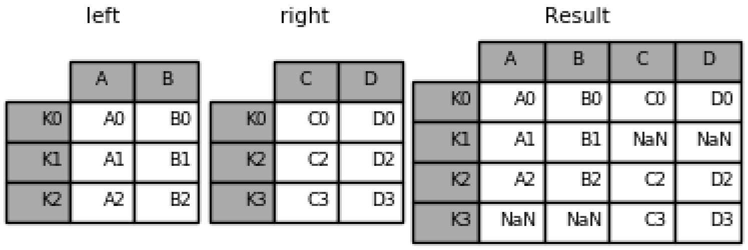
right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

```
In [82]: result = left.join(right, how='outer')
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

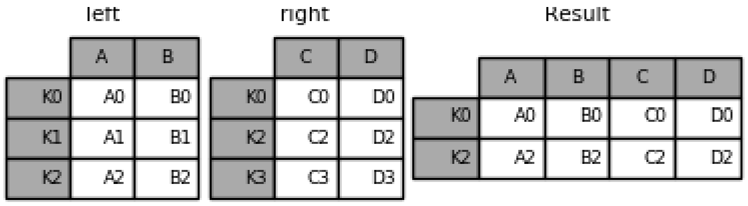
	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

The same as above, but with `how='inner'`.

```
In [83]: result = left.join(right, how='inner')
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

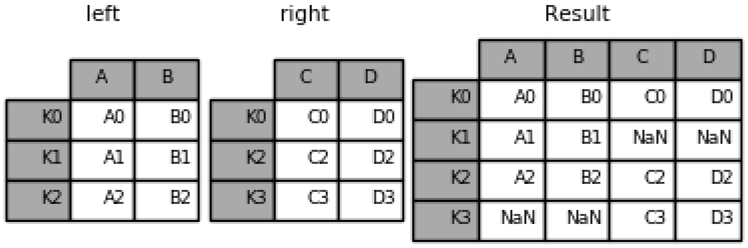
Result

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

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 [84]: result = pd.merge(left, right, left_index=True, right_index=True, how='outer
↪')
```

```

```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

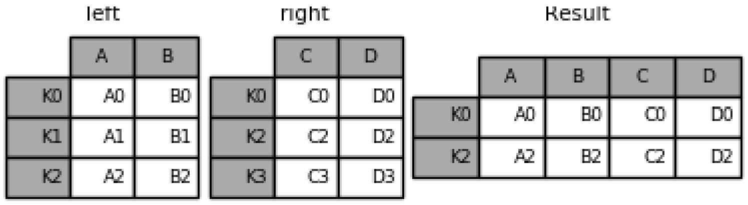
	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

```
In [85]: result = pd.merge(left, right, left_index=True, right_index=True, how='inner
↪');
```

```

```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

Joining key columns on an index

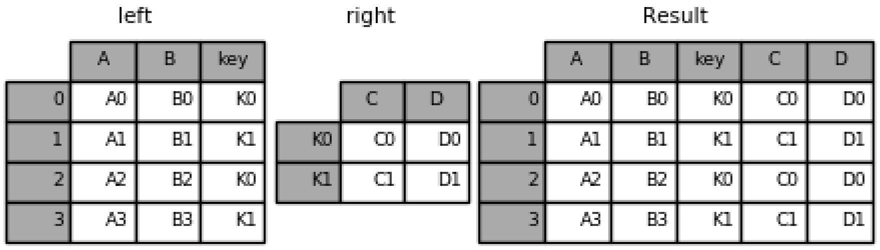
`join()` takes an optional `on` argument which may be a column or multiple column names, which specifies that the passed `DataFrame` is to be aligned on that column in the `DataFrame`. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
pd.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 `DataFrames` is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [86]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [87]: right = pd.DataFrame({'C': ['C0', 'C1'],
   ....:                       'D': ['D0', 'D1']},
   ....:                      index=['K0', 'K1'])
   ....:

In [88]: result = left.join(right, on='key')
```

```

```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

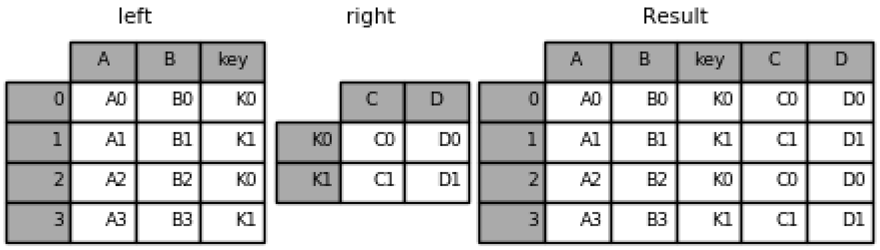
right

	C	D
K0	C0	D0
K1	C1	D1

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

```
In [89]: result = pd.merge(left, right, left_on='key', right_index=True,
   ....:                   how='left', sort=False);
   ....:
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

right

	C	D
K0	C0	D0
K1	C1	D1

Result

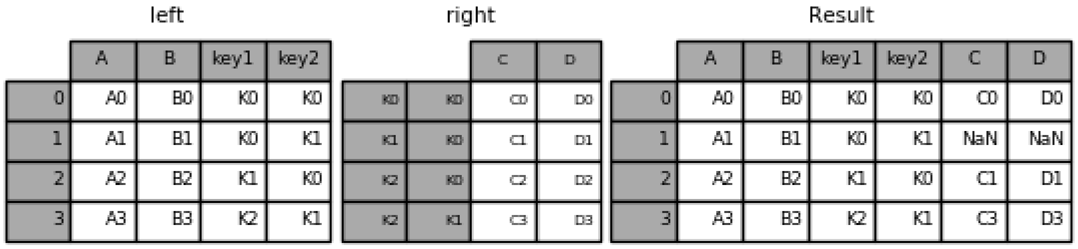
	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

To join on multiple keys, the passed DataFrame must have a `MultiIndex`:

```
In [90]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [91]: index = pd.MultiIndex.from_tuples([('K0', 'K0'), ('K1', 'K0'),
   ....:                                   ('K2', 'K0'), ('K2', 'K1')])
   ....:

In [92]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                      index=index)
   ....:
```

Now this can be joined by passing the two key column names:

```
In [93]: result = left.join(right, on=['key1', 'key2'])
```



left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

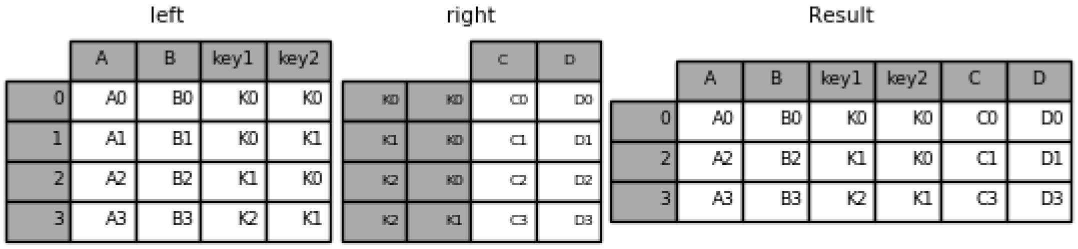
		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A3	B3	K2	K1	C3	D3

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 [94]: result = left.join(right, on=['key1', 'key2'], how='inner')
```



left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

Result

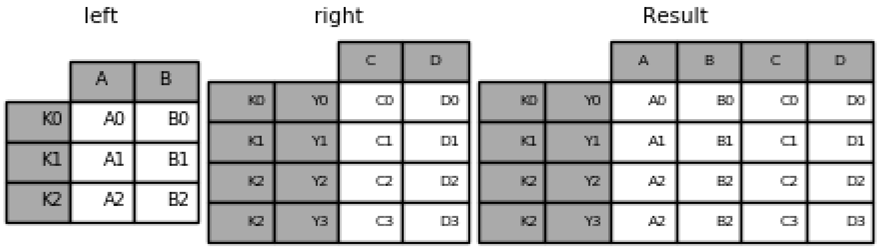
	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
2	A2	B2	K1	K0	C1	D1
3	A3	B3	K2	K1	C3	D3

As you can see, this drops any rows where there was no match.

Joining a single Index to a MultiIndex

You can join a singly-indexed DataFrame with a level of a MultiIndexed DataFrame. The level will match on the name of the index of the singly-indexed frame against a level name of the MultiIndexed frame.

```
In [95]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                      index=pd.Index(['K0', 'K1', 'K2'], name='key'))
   ....:

In [96]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   ....:                                   ('K2', 'Y2'), ('K2', 'Y3')],
   ....:                                    names=['key', 'Y'])
   ....:

In [97]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                       index=index)
   ....:

In [98]: result = left.join(right, how='inner')
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

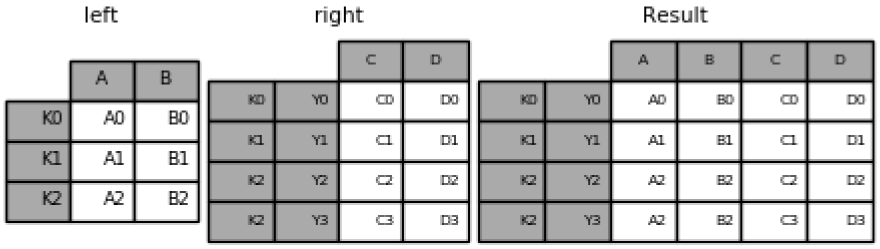
		C	D
K0	Y0	C0	D0
K1	Y1	C1	D1
K2	Y2	C2	D2
K2	Y3	C3	D3

Result

		A	B	C	D
K0	Y0	A0	B0	C0	D0
K1	Y1	A1	B1	C1	D1
K2	Y2	A2	B2	C2	D2
K2	Y3	A2	B2	C3	D3

This is equivalent but less verbose and more memory efficient / faster than this.

```
In [99]: result = pd.merge(left.reset_index(), right.reset_index(),
   ....:       on=['key'], how='inner').set_index(['key','Y'])
   ....:
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

		C	D
K0	Y0	C0	D0
K1	Y1	C1	D1
K2	Y2	C2	D2
K2	Y3	C3	D3

Result

		A	B	C	D
K0	Y0	A0	B0	C0	D0
K1	Y1	A1	B1	C1	D1
K2	Y2	A2	B2	C2	D2
K2	Y3	A2	B2	C3	D3

Joining with two MultiIndexes

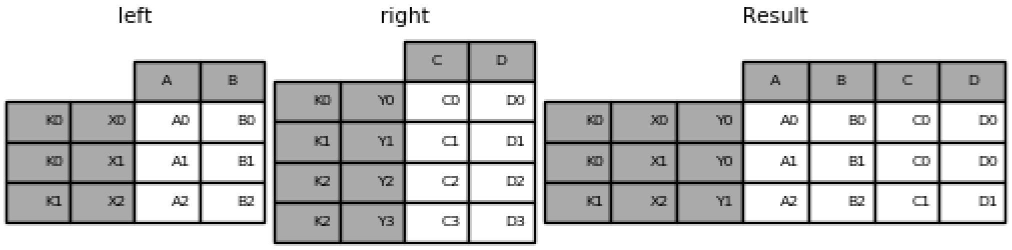
This is supported in a limited way, provided that the index for the right argument is completely used in the join, and is a subset of the indices in the left argument, as in this example:

```
In [100]: leftindex = pd.MultiIndex.from_product([list('abc'), list('xy'), [1,
↪ 2]],
   .....:                                        names=['abc', 'xy', 'num'])
   .....:

In [101]: left = pd.DataFrame({'v1': range(12)}, index=leftindex)

In [102]: left
Out[102]:
            v1
abc xy num
a   x  1     0
       2     1
    y  1     2
       2     3
b   x  1     4
       2     5
    y  1     6
       2     7
c   x  1     8
       2     9
    y  1    10
       2    11

In [103]: rightindex = pd.MultiIndex.from_product([list('abc'), list('xy')],
   .....:                                         names=['abc', 'xy'])
   .....:

In [104]: right = pd.DataFrame({'v2': [100 * i for i in range(1, 7)]},␣
↪index=rightindex)

In [105]: right
Out[105]:
         v2
abc xy
a   x   100
    y   200
```

```
b   x   300
    y   400
c   x   500
    y   600

In [106]: left.join(right, on=['abc', 'xy'], how='inner')
Out[106]:
            v1   v2
abc xy num
a   x  1     0  100
       2     1  100
    y  1     2  200
       2     3  200
b   x  1     4  300
       2     5  300
    y  1     6  400
       2     7  400
c   x  1     8  500
       2     9  500
    y  1    10  600
       2    11  600
```

If that condition is not satisfied, a join with two multi-indexes can be done using the following code.

```
In [107]: leftindex = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
   .....:                                        ('K1', 'X2')],
   .....:                                       names=['key', 'X'])
   .....:

In [108]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   .....:                      'B': ['B0', 'B1', 'B2']},
   .....:                     index=leftindex)
   .....:

In [109]: rightindex = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   .....:                                         ('K2', 'Y2'), ('K2', 'Y3')],
   .....:                                        names=['key', 'Y'])
   .....:

In [110]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   .....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   .....:                      index=rightindex)
   .....:

In [111]: result = pd.merge(left.reset_index(), right.reset_index(),
   .....:                   on=['key'], how='inner').set_index(['key', 'X', 'Y'])
   .....:
```



left

		A	B
K0	X0	A0	B0
K0	X1	A1	B1
K1	X2	A2	B2

right

		C	D
K0	Y0	C0	D0
K1	Y1	C1	D1
K2	Y2	C2	D2
K2	Y3	C3	D3

Result

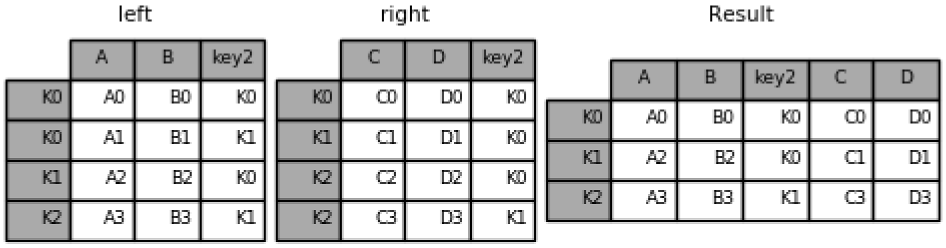
			A	B	C	D
K0	X0	Y0	A0	B0	C0	D0
K0	X1	Y0	A1	B1	C0	D0
K1	X2	Y1	A2	B2	C1	D1

Merging on a combination of columns and index levels

New in version 0.23.

Strings passed as the `on`, `left_on`, and `right_on` parameters may refer to either column names or index level names. This enables merging `DataFrame` instances on a combination of index levels and columns without resetting indexes.

```
In [112]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [113]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   .....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   .....:                      'key2': ['K0', 'K1', 'K0', 'K1']},
   .....:                     index=left_index)
   .....:

In [114]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [115]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   .....:                       'D': ['D0', 'D1', 'D2', 'D3'],
   .....:                       'key2': ['K0', 'K0', 'K0', 'K1']},
   .....:                      index=right_index)
   .....:

In [116]: result = left.merge(right, on=['key1', 'key2'])
```



left

	A	B	key2
K0	A0	B0	K0
K0	A1	B1	K1
K1	A2	B2	K0
K2	A3	B3	K1

right

	C	D	key2
K0	C0	D0	K0
K1	C1	D1	K0
K2	C2	D2	K0
K2	C3	D3	K1

Result

	A	B	key2	C	D
K0	A0	B0	K0	C0	D0
K1	A2	B2	K0	C1	D1
K2	A3	B3	K1	C3	D3

Note: When DataFrames are merged on a string that matches an index level in both frames, the index level is preserved as an index level in the resulting DataFrame.

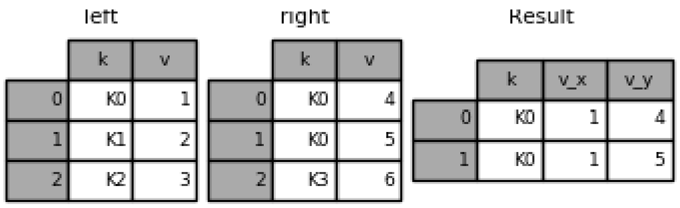
Note: When DataFrames are merged using only some of the levels of a *MultiIndex*, the extra levels will be dropped from the resulting merge. In order to preserve those levels, use `reset_index` on those level names to move those levels to columns prior to doing the merge.

Note: If a string matches both a column name and an index level name, then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

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 [117]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})

In [118]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})

In [119]: result = pd.merge(left, right, on='k')
```



left

	k	v
0	K0	1
1	K1	2
2	K2	3

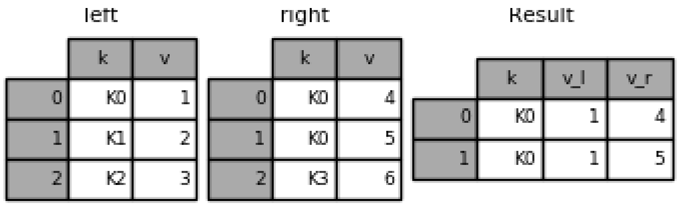
right

	k	v
0	K0	4
1	K0	5
2	K3	6

Result

	k	v_x	v_y
0	K0	1	4
1	K0	1	5

```
In [120]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```



left

	k	v
0	K0	1
1	K1	2
2	K2	3

right

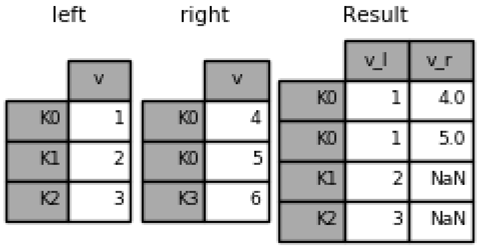
	k	v
0	K0	4
1	K0	5
2	K3	6

Result

	k	v_l	v_r
0	K0	1	4
1	K0	1	5

`DataFrame.join()` has `lsuffix` and `rsuffix` arguments which behave similarly.

```
In [121]: left = left.set_index('k')

In [122]: right = right.set_index('k')

In [123]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```



left

	v
K0	1
K1	2
K2	3

right

	v
K0	4
K0	5
K3	6

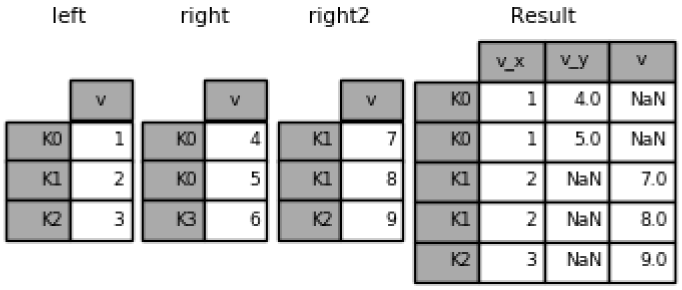
Result

	v_l	v_r
K0	1	4.0
K0	1	5.0
K1	2	NaN
K2	3	NaN

Joining multiple DataFrames

A list or tuple of `DataFrames` can also be passed to `join()` to join them together on their indexes.

```
In [124]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])

In [125]: result = left.join([right, right2])
```



left

	v
K0	1
K1	2
K2	3

right

	v
K0	4
K0	5
K3	6

right2

	v
K1	7
K1	8
K2	9

Result

	v_x	v_y	v
K0	1	4.0	NaN
K0	1	5.0	NaN
K1	2	NaN	7.0
K1	2	NaN	8.0
K2	3	NaN	9.0

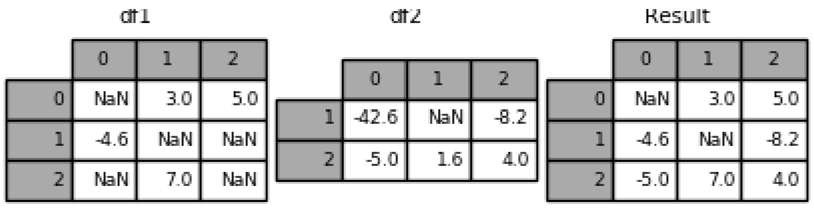
Merging together values within Series or DataFrame columns

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 [126]: df1 = pd.DataFrame([[np.nan, 3., 5.], [-4.6, np.nan, np.nan],
   .....:                    [np.nan, 7., np.nan]])
   .....:

In [127]: df2 = pd.DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4]],
   .....:                    index=[1, 2])
   .....:
```

For this, use the `combine_first()` method:

```
In [128]: result = df1.combine_first(df2)
```



df1

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	NaN
2	NaN	7.0	NaN

df2

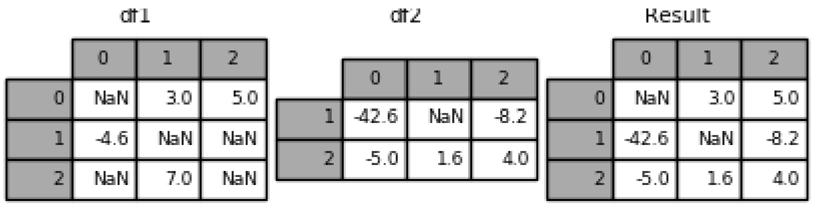
	0	1	2
1	-42.6	NaN	-8.2
2	-5.0	1.6	4.0

Result

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	-8.2
2	-5.0	7.0	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 in place:

```
In [129]: df1.update(df2)
```



df1

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	NaN
2	NaN	7.0	NaN

df2

	0	1	2
1	-42.6	NaN	-8.2
2	-5.0	1.6	4.0

Result

	0	1	2
0	NaN	3.0	5.0
1	-42.6	NaN	-8.2
2	-5.0	1.6	4.0

4.4.3 Timeseries friendly merging

Merging ordered data

A merge_ordered() function allows combining time series and other ordered data. In particular it has an optional fill_method keyword to fill/interpolate missing data:

```
In [130]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
   .....:                      'lv': [1, 2, 3, 4],
   .....:                      's': ['a', 'b', 'c', 'd']})
   .....:

In [131]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
   .....:                       'rv': [1, 2, 3]})
   .....:

In [132]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')
Out[132]:
     k   lv  s   rv
0   K0  1.0  a  NaN
1   K1  1.0  a  1.0
2   K2  1.0  a  2.0
3   K4  1.0  a  3.0
4   K1  2.0  b  1.0
5   K2  2.0  b  2.0
6   K4  2.0  b  3.0
7   K1  3.0  c  1.0
8   K2  3.0  c  2.0
9   K4  3.0  c  3.0
10  K1  NaN  d  1.0
11  K2  4.0  d  2.0
12  K4  4.0  d  3.0
```

Merging asof

New in version 0.19.0.

A merge_asof() is similar to an ordered left-join except that we match on nearest key rather than equal keys. For each row in the left DataFrame, we select the last row in the right DataFrame whose on key is less than the lefts key. Both DataFrames must be sorted by the key.

Optionally an asof merge can perform a group-wise merge. This matches the by key equally, in addition to the nearest match on the on key.

For example; we might have trades and quotes and we want to asof merge them.

```
In [133]: trades = pd.DataFrame({
   .....:     'time': pd.to_datetime(['20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.038',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.048']),
   .....:     'ticker': ['MSFT', 'MSFT',
   .....:                'GOOG', 'GOOG', 'AAPL'],
   .....:     'price': [51.95, 51.95,
   .....:               720.77, 720.92, 98.00],
   .....:     'quantity': [75, 155,
   .....:                  100, 100, 100]},
   .....:     columns=['time', 'ticker', 'price', 'quantity'])
   .....:

In [134]: quotes = pd.DataFrame({
   .....:     'time': pd.to_datetime(['20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.030',
   .....:                             '20160525 13:30:00.041',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.049',
   .....:                             '20160525 13:30:00.072',
   .....:                             '20160525 13:30:00.075']),
   .....:     'ticker': ['GOOG', 'MSFT', 'MSFT',
   .....:                'MSFT', 'GOOG', 'AAPL', 'GOOG',
   .....:                'MSFT'],
   .....:     'bid': [720.50, 51.95, 51.97, 51.99,
   .....:             720.50, 97.99, 720.50, 52.01],
   .....:     'ask': [720.93, 51.96, 51.98, 52.00,
   .....:             720.93, 98.01, 720.88, 52.03]},
   .....:     columns=['time', 'ticker', 'bid', 'ask'])
   .....:
```

```
In [135]: trades
Out[135]:
                     time ticker   price  quantity
0 2016-05-25 13:30:00.023   MSFT   51.95        75
1 2016-05-25 13:30:00.038   MSFT   51.95       155
2 2016-05-25 13:30:00.048   GOOG  720.77       100
3 2016-05-25 13:30:00.048   GOOG  720.92       100
4 2016-05-25 13:30:00.048   AAPL   98.00       100

In [136]: quotes
Out[136]:
                     time ticker     bid     ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT   51.95   51.96
2 2016-05-25 13:30:00.030   MSFT   51.97   51.98
3 2016-05-25 13:30:00.041   MSFT   51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL   97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT   52.01   52.03
```

By default we are taking the asof of the quotes.

```
In [137]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker')
   .....: 
Out[137]: 
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within `2ms` between the quote time and the trade time.

```
In [138]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('2ms'))
   .....: 
Out[138]: 
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155     NaN     NaN
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

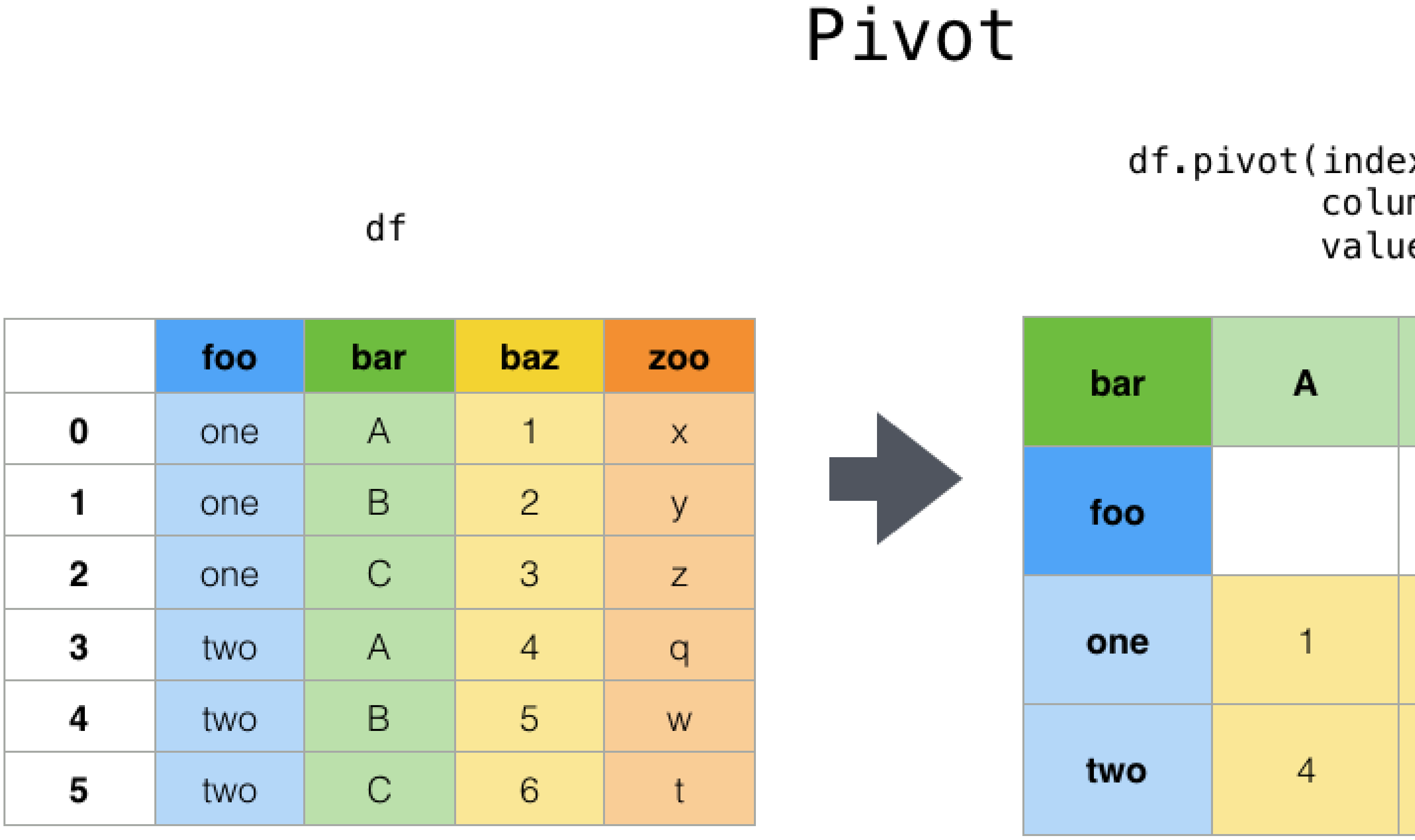
We only asof within `10ms` between the quote time and the trade time and we exclude exact matches on time. Note that though we exclude the exact matches (of the quotes), prior quotes **do** propagate to that point in time.

```
In [139]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('10ms'),
   .....:               allow_exact_matches=False)
   .....: 
Out[139]: 
                     time ticker   price  quantity    bid    ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75    NaN    NaN
1 2016-05-25 13:30:00.038   MSFT   51.95       155  51.97  51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100    NaN    NaN
3 2016-05-25 13:30:00.048   GOOG  720.92       100    NaN    NaN
4 2016-05-25 13:30:00.048   AAPL   98.00       100    NaN    NaN
```

{{ header }}

4.5 Reshaping and pivot tables

4.5.1 Reshaping by pivoting DataFrame objects



	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A
foo	
one	1
two	4

Data is often stored in so-called stacked or record format:

```
In [1]: df
Out[1]:
         date variable     value
0  2000-01-03        A  0.259126
1  2000-01-04        A  0.774683
2  2000-01-05        A  0.713390
3  2000-01-03        B -0.837638
4  2000-01-04        B  0.064089
5  2000-01-05        B -0.034934
6  2000-01-03        C -0.452395
7  2000-01-04        C  2.488398
8  2000-01-05        C  1.429389
9  2000-01-03        D  0.547623
10 2000-01-04        D  0.008959
11 2000-01-05        D  1.097237
```

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.to_numpy().ravel('F'),
            'variable': np.asarray(frame.columns).repeat(N),
            'date': np.tile(np.asarray(frame.index), K)}
    return pd.DataFrame(data, columns=['date', 'variable', 'value'])


df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable A we could do:

```
In [2]: df[df['variable'] == 'A']
Out[2]: 
        date variable     value
0 2000-01-03        A  0.259126
1 2000-01-04        A  0.774683
2 2000-01-05        A  0.713390
```

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, we use the DataFrame.pivot() method (also implemented as a top level function *pivot()*):

```
In [3]: df.pivot(index='date', columns='variable', values='value')
Out[3]: 
variable           A         B         C         D
date                                              
2000-01-03  0.259126 -0.837638 -0.452395  0.547623
2000-01-04  0.774683  0.064089  2.488398  0.008959
2000-01-05  0.713390 -0.034934  1.429389  1.097237
```

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(index='date', columns='variable')

In [6]: pivoted
Out[6]: 
               value                                       value2                ␣
↪
variable           A         B         C         D         A         B         C ␣
↪    D
date                                                                             ␣
↪
2000-01-03  0.259126 -0.837638 -0.452395  0.547623  0.518252 -1.675276 -0.904790  1.
↪095246
2000-01-04  0.774683  0.064089  2.488398  0.008959  1.549366  0.128178  4.976796  0.
↪017919
2000-01-05  0.713390 -0.034934  1.429389  1.097237  1.426780 -0.069868  2.858779  2.
↪194473
```

(continues on next page)

(continued from previous page)

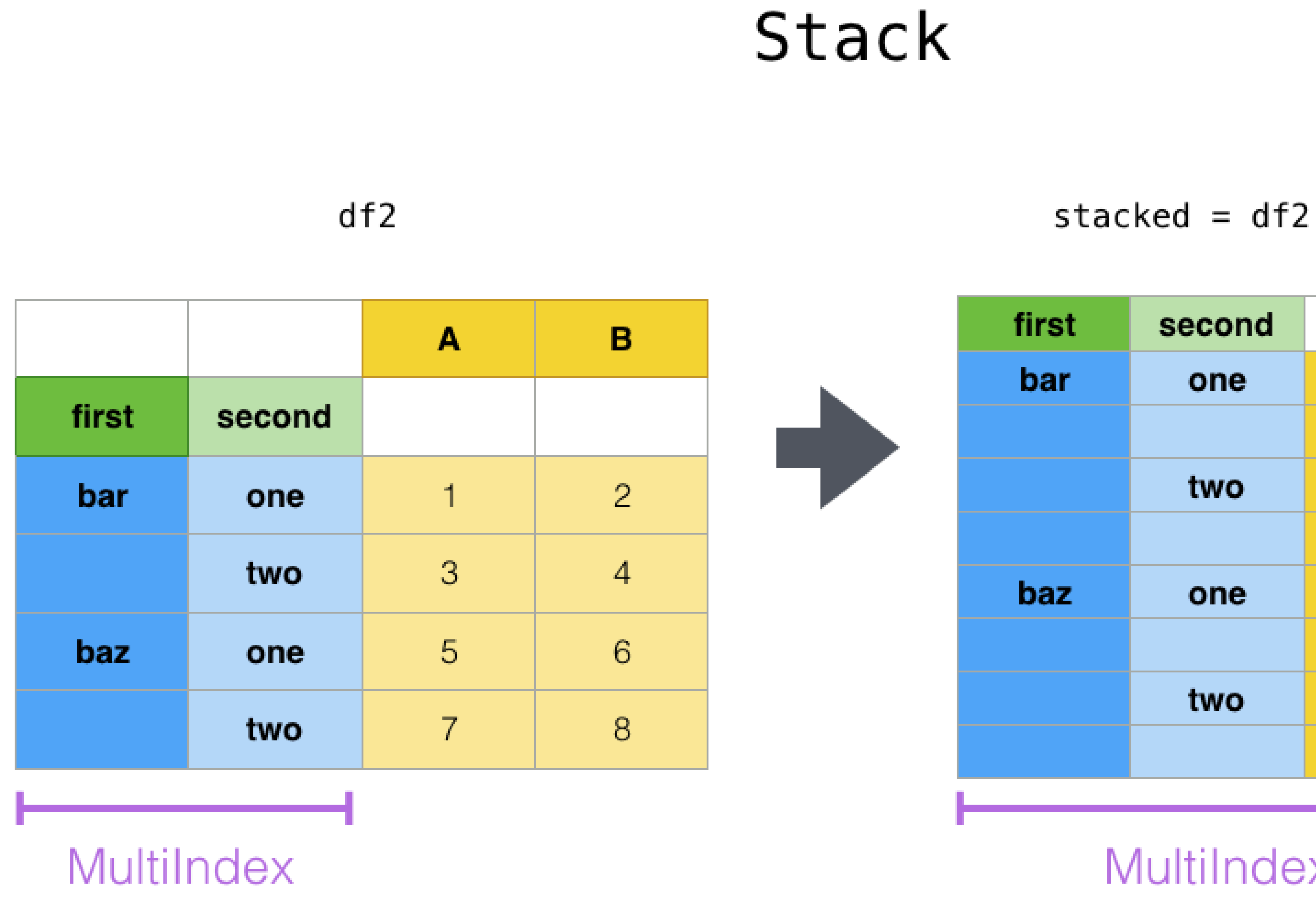
You can then select subsets from the pivoted `DataFrame`:

```
In [7]: pivoted['value2']
Out[7]: 
variable           A         B         C         D
date                                              
2000-01-03  0.518252 -1.675276 -0.904790  1.095246
2000-01-04  1.549366  0.128178  4.976796  0.017919
2000-01-05  1.426780 -0.069868  2.858779  2.194473
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

Note: *`pivot()`* will error with a `ValueError: Index contains duplicate entries, cannot reshape` if the index/column pair is not unique. In this case, consider using *`pivot_table()`* which is a generalization of pivot that can handle duplicate values for one index/column pair.

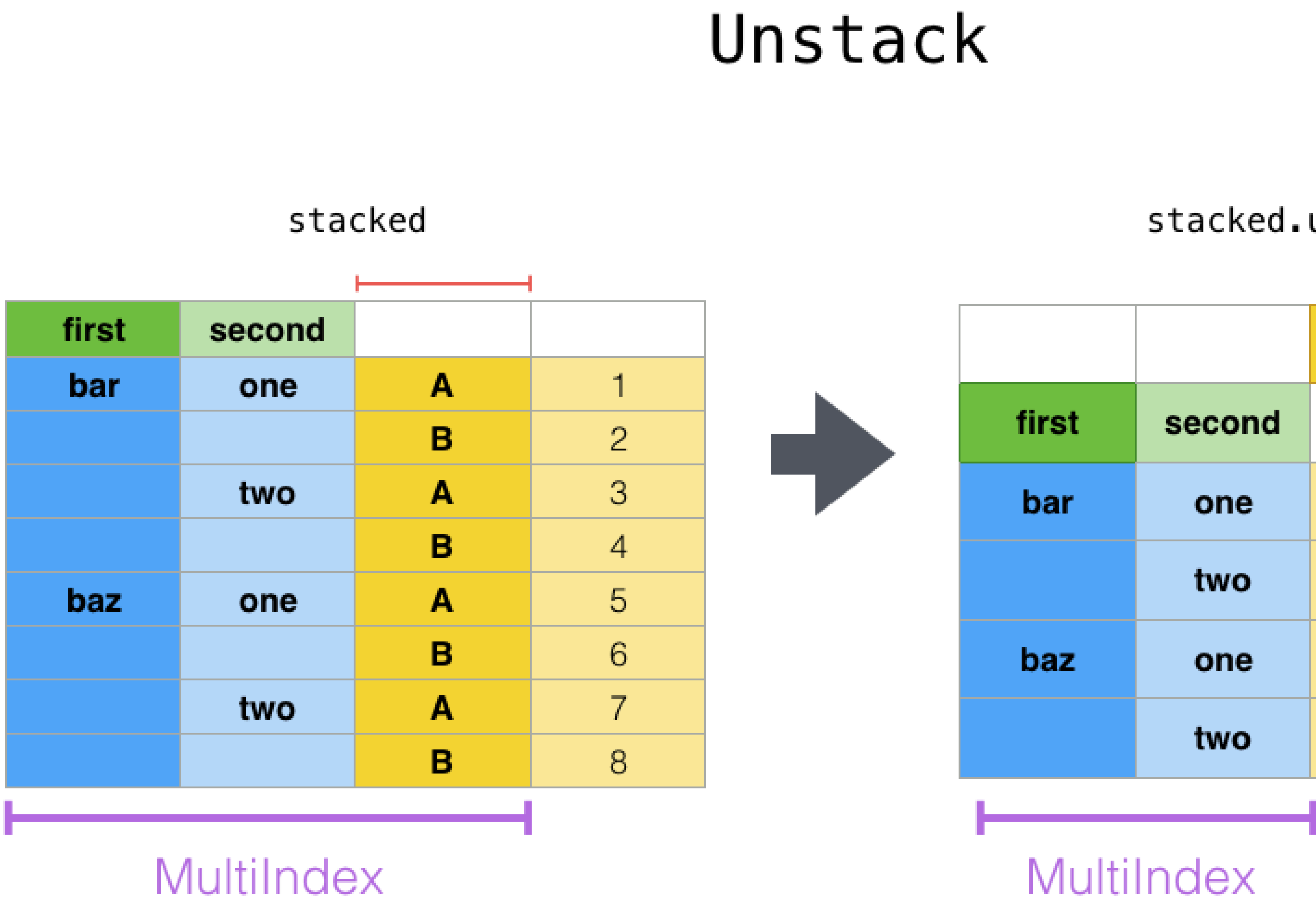
4.5.2 Reshaping by stacking and unstacking



		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

Closely related to the `pivot()` method are the related `stack()` and `unstack()` methods available on `Series` and `DataFrame`. These methods are designed to work together with `MultiIndex` objects (see the section on *hierarchical indexing*). Here are essentially what these methods 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 of `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. Lets take a prior example data set from the hierarchical indexing section:

```
In [8]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
   ...:                      'foo', 'foo', 'qux', 'qux'],
   ...:                     ['one', 'two', 'one', 'two',
   ...:                      'one', 'two', 'one', 'two']]))
   ...:

In [9]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [10]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [11]: df2 = df[:4]

In [12]: df2
Out[12]:
                     A         B
first second
bar   one    -1.427767  1.011174
      two    -0.227837  0.260297
```

(continues on next page)

(continued from previous page)

```
baz   one    -0.664499 -1.085553
      two    -1.392521 -0.426500
```

The `stack` function compresses a level in the `DataFrame`s columns to produce either:

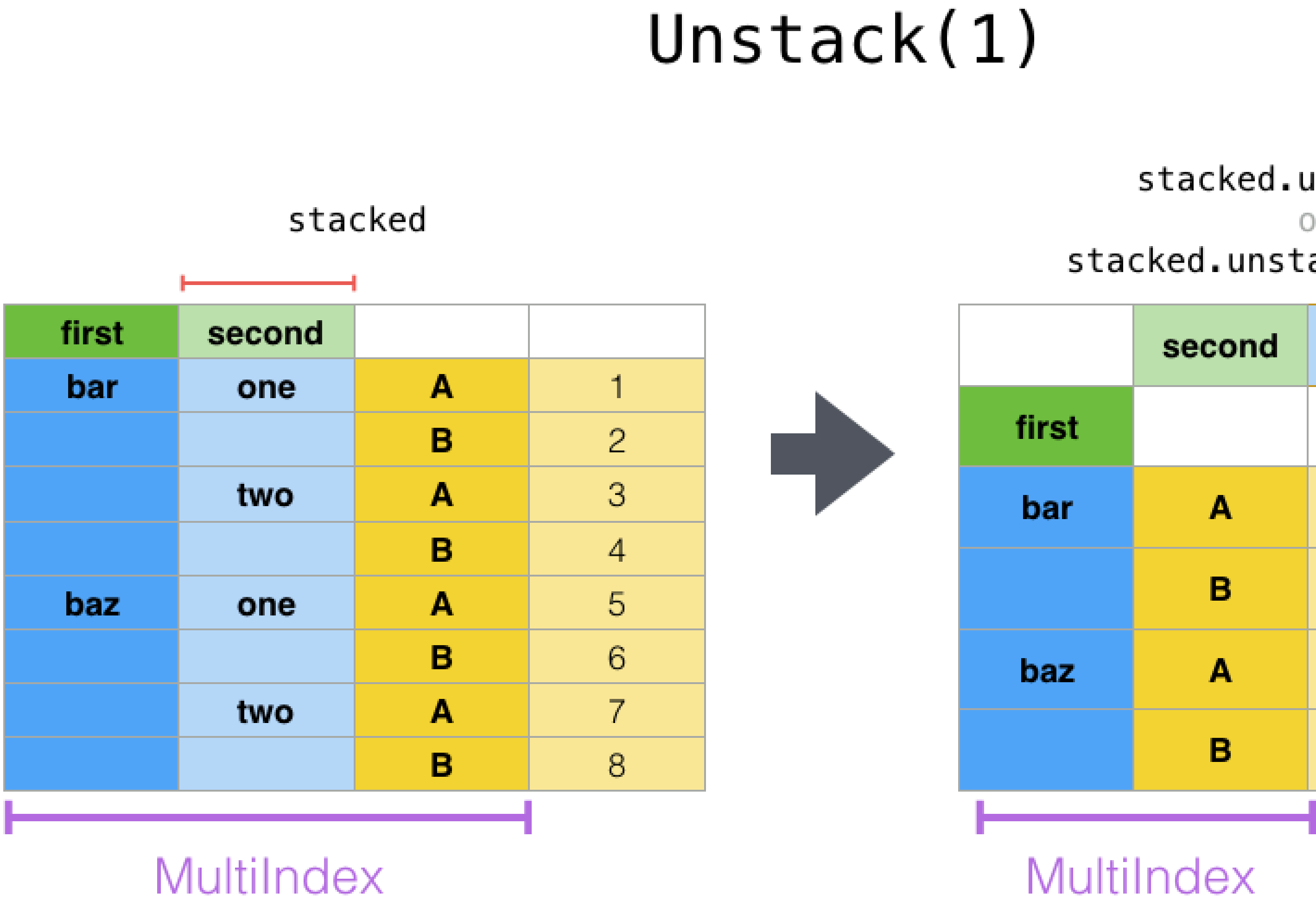
- A `Series`, in the case of a simple column Index.
- A `DataFrame`, in the case of a `MultiIndex` in the columns.

If the columns have a `MultiIndex`, you can choose which level to stack. The stacked level becomes the new lowest level in a `MultiIndex` on the columns:

```
In [13]: stacked = df2.stack()

In [14]: stacked
Out[14]: 
first  second   
bar    one     A   -1.427767
               B    1.011174
       two     A   -0.227837
               B    0.260297
baz    one     A   -0.664499
               B   -1.085553
       two     A   -1.392521
               B   -0.426500
dtype: float64
```

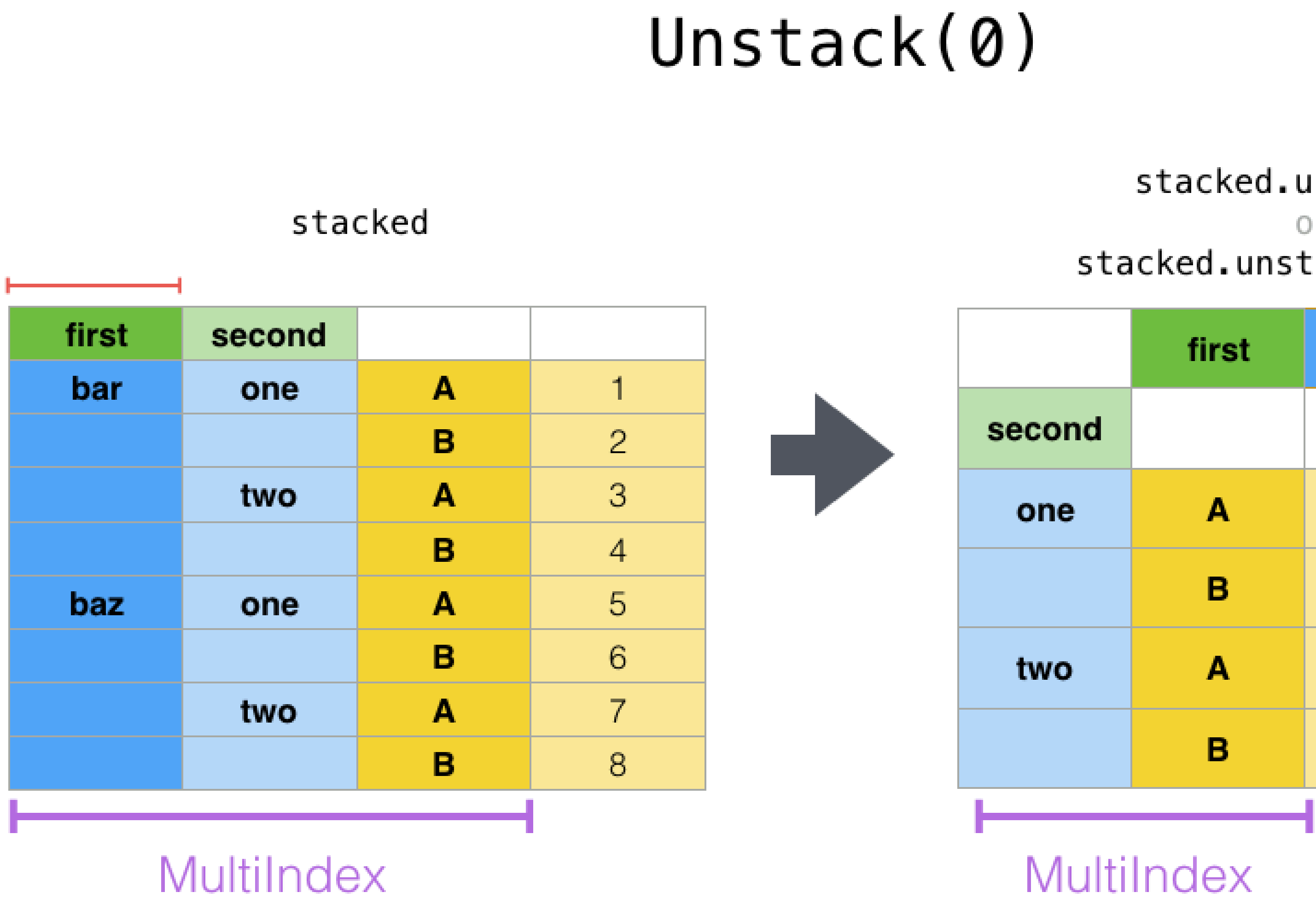
With a stacked `DataFrame` or `Series` (having a `MultiIndex` as the `index`), the inverse operation of `stack` is `unstack`, which by default unstacks the **last level**:

```
In [15]: stacked.unstack()
Out[15]: 
                     A         B
first second                    
bar   one    -1.427767  1.011174
      two    -0.227837  0.260297
baz   one    -0.664499 -1.085553
      two    -1.392521 -0.426500

In [16]: stacked.unstack(1)
Out[16]: 
second        one       two
first                      
bar   A -1.427767 -0.227837
      B  1.011174  0.260297
baz   A -0.664499 -1.392521
      B -1.085553 -0.426500

In [17]: stacked.unstack(0)
Out[17]: 
first          bar       baz
second                      
one    A -1.427767 -0.664499
       B  1.011174 -1.085553
two    A -0.227837 -1.392521
       B  0.260297 -0.426500
```



If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [18]: stacked.unstack('second')
Out[18]:
second          one       two
first
bar    A -1.427767 -0.227837
       B  1.011174  0.260297
baz    A -0.664499 -1.392521
       B -1.085553 -0.426500
```



Notice that the `stack` and `unstack` methods implicitly sort the index levels involved. Hence a call to `stack` and then `unstack`, or vice versa, will result in a **sorted** copy of the original `DataFrame` or `Series`:

```
In [19]: index = pd.MultiIndex.from_product([[2, 1], ['a', 'b']])

In [20]: df = pd.DataFrame(np.random.randn(4), index=index, columns=['A'])

In [21]: df
Out[21]:
            A
2 a -1.727109
  b -1.966122
1 a -0.004308
  b  0.627249

In [22]: all(df.unstack().stack() == df.sort_index())
Out[22]: True
```

The above code will raise a `TypeError` if the call to `sort_index` is removed.

Multiple levels

You may also stack or unstack more than one level at a time by passing a list of levels, in which case the end result is as if each level in the list were processed individually.

```
In [23]: columns = pd.MultiIndex.from_tuples([
   ....:     ('A', 'cat', 'long'), ('B', 'cat', 'long'),
   ....:     ('A', 'dog', 'short'), ('B', 'dog', 'short')],
   ....:     names=['exp', 'animal', 'hair_length']
   ....: )
   ....:

In [24]: df = pd.DataFrame(np.random.randn(4, 4), columns=columns)

In [25]: df
Out[25]:
exp                 A         B         A         B
animal            cat       cat       dog       dog
hair_length      long      long     short     short
0           -2.260817  0.023953 -1.328037 -0.091360
1           -0.063483  0.691641  1.062240 -1.912934
2           -0.967661  1.438160  1.796396  0.364482
3            0.384514  0.774313 -1.215737  1.533214

In [26]: df.stack(level=['animal', 'hair_length'])
Out[26]:
exp                        A         B
  animal hair_length
0 cat    long        -2.260817  0.023953
  dog    short       -1.328037 -0.091360
1 cat    long        -0.063483  0.691641
  dog    short        1.062240 -1.912934
2 cat    long        -0.967661  1.438160
  dog    short        1.796396  0.364482
3 cat    long         0.384514  0.774313
  dog    short       -1.215737  1.533214
```

The list of levels can contain either level names or level numbers (but not a mixture of the two).

```
# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [27]: df.stack(level=[1, 2])
Out[27]:
exp                        A         B
  animal hair_length
0 cat    long        -2.260817  0.023953
  dog    short       -1.328037 -0.091360
1 cat    long        -0.063483  0.691641
  dog    short        1.062240 -1.912934
2 cat    long        -0.967661  1.438160
  dog    short        1.796396  0.364482
3 cat    long         0.384514  0.774313
  dog    short       -1.215737  1.533214
```

Missing data

These functions are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling `sort_index`, of course). Here is a more complex example:

```
In [28]: columns = pd.MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
   ....:                                      ('B', 'cat'), ('A', 'dog')],
   ....:                                     names=['exp', 'animal'])
   ....:

In [29]: index = pd.MultiIndex.from_product([('bar', 'baz', 'foo', 'qux'),
   ....:                                     ('one', 'two')],
   ....:                                    names=['first', 'second'])
   ....:

In [30]: df = pd.DataFrame(np.random.randn(8, 4), index=index, columns=columns)

In [31]: df2 = df.iloc[[0, 1, 2, 4, 5, 7]]

In [32]: df2
Out[32]:
exp                  A         B                   A
animal             cat       dog       cat       dog
first second
bar   one    -1.436953  0.561814 -0.346880 -0.546060
      two    -1.993054 -0.689365 -0.877031  1.507935
baz   one    -1.866380  0.043384  0.252683 -0.479004
foo   one    -1.613149 -0.622599  0.291003  0.792238
      two     0.807151 -0.758613 -2.393856  1.098272
qux   two    -0.411186  1.584705 -1.042868 -0.295906
```

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [33]: df2.stack('exp')
Out[33]:
animal                 cat       dog
first second exp
bar   one    A   -1.436953 -0.546060
             B   -0.346880  0.561814
      two    A   -1.993054  1.507935
             B   -0.877031 -0.689365
baz   one    A   -1.866380 -0.479004
             B    0.252683  0.043384
foo   one    A   -1.613149  0.792238
             B    0.291003 -0.622599
      two    A    0.807151  1.098272
             B   -2.393856 -0.758613
qux   two    A   -0.411186 -0.295906
             B   -1.042868  1.584705

In [34]: df2.stack('animal')
Out[34]:
exp                          A         B
first second animal
bar   one    cat     -1.436953 -0.346880
             dog     -0.546060  0.561814
      two    cat    -1.993054 -0.877031
             dog     1.507935 -0.689365
baz   one    cat    -1.866380  0.252683
             dog    -0.479004  0.043384
foo   one    cat    -1.613149  0.291003
             dog     0.792238 -0.622599
      two    cat     0.807151 -2.393856
             dog     1.098272 -0.758613
qux   two    cat    -0.411186 -1.042868
             dog    -0.295906  1.584705
```

Unstacking can result in missing values if subgroups do not have the same set of labels. By default, missing values will be replaced with the default fill value for that data type, NaN for float, NaT for datetimelike, etc. For integer types, by default data will converted to float and missing values will be set to NaN.

```
In [35]: df3 = df.iloc[[0, 1, 4, 7], [1, 2]]

In [36]: df3
Out[36]: 
exp                  B          
animal             dog       cat
first second                    
bar   one     0.561814 -0.346880
      two    -0.689365 -0.877031
foo   one    -0.622599  0.291003
qux   two     1.584705 -1.042868

In [37]: df3.unstack()
Out[37]: 
exp            B                                 
animal       dog                 cat             
second       one       two       one       two
first                                          
bar     0.561814 -0.689365 -0.346880 -0.877031
foo    -0.622599       NaN  0.291003       NaN
qux          NaN  1.584705       NaN -1.042868
```

New in version 0.18.0.

Alternatively, unstack takes an optional fill_value argument, for specifying the value of missing data.

```
In [38]: df3.unstack(fill_value=-1e9)
Out[38]: 
exp                B                                                    
animal           dog                         cat                        
second           one           two           one           two
first                                                                 
bar     5.618142e-01 -6.893650e-01 -3.468802e-01 -8.770315e-01
foo    -6.225992e-01 -1.000000e+09  2.910029e-01 -1.000000e+09
qux    -1.000000e+09  1.584705e+00 -1.000000e+09 -1.042868e+00
```

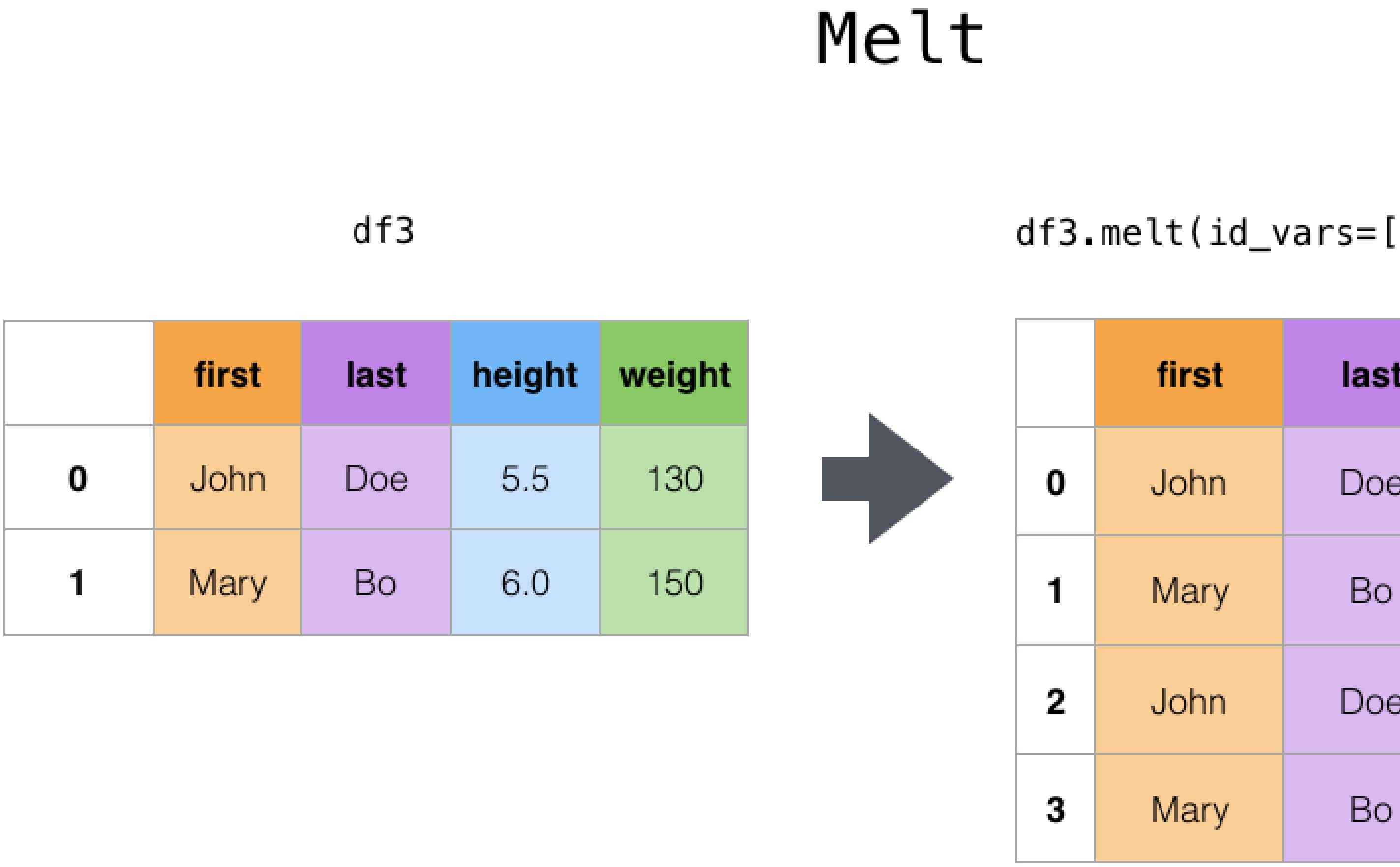
With a MultiIndex

Unstacking when the columns are a MultiIndex is also careful about doing the right thing:

```
In [39]: df[:3].unstack(0)
```

```
Out[39]:
exp            A                  B                                       A
animal       cat                dog                 cat                 dog
first        bar      baz       bar       baz       bar       baz       bar ␣
↪     baz
second
one    -1.436953 -1.86638  0.561814  0.043384 -0.346880  0.252683 -0.546060 -
↪0.479004
two    -1.993054      NaN -0.689365       NaN -0.877031       NaN  1.507935 ␣
↪     NaN

In [40]: df2.unstack(1)
Out[40]:
exp            A                   B                                       A
animal       cat                 dog                 cat                 dog
second       one       two       one       two       one       two       one ␣
↪      two
first
bar    -1.436953 -1.993054  0.561814 -0.689365 -0.346880 -0.877031 -0.546060 ␣
↪1.507935
baz    -1.866380       NaN  0.043384       NaN  0.252683       NaN -0.479004 ␣
↪      NaN
foo    -1.613149  0.807151 -0.622599 -0.758613  0.291003 -2.393856  0.792238 ␣
↪1.098272
qux          NaN -0.411186       NaN  1.584705       NaN -1.042868       NaN -
↪0.295906
```

4.5.3 Reshaping by Melt



	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

	first	last
0	John	Doe
1	Mary	Bo
2	John	Doe
3	Mary	Bo

The top-level `melt()` function and the corresponding `DataFrame.melt()` are useful to massage a `DataFrame` into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are unpivoted to the row axis, leaving just two non-identifier columns, variable and value. The names of those columns can be customized by supplying the `var_name` and `value_name` parameters.

For instance,

```
In [41]: cheese = pd.DataFrame({'first': ['John', 'Mary'],
   ....:                        'last': ['Doe', 'Bo'],
   ....:                        'height': [5.5, 6.0],
   ....:                        'weight': [130, 150]})
   ....:

In [42]: cheese
Out[42]:
  first last  height  weight
0  John  Doe     5.5     130
1  Mary   Bo     6.0     150

In [43]: cheese.melt(id_vars=['first', 'last'])
Out[43]:
  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 [44]: cheese.melt(id_vars=['first', 'last'], var_name='quantity')
Out[44]: 
  first last quantity  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

Another way to transform is to use the *wide_to_long()* panel data convenience function. It is less flexible than *melt()*, but more user-friendly.

```
In [45]: dft = pd.DataFrame({"A1970": {0: "a", 1: "b", 2: "c"},
   ....:                     "A1980": {0: "d", 1: "e", 2: "f"},
   ....:                     "B1970": {0: 2.5, 1: 1.2, 2: .7},
   ....:                     "B1980": {0: 3.2, 1: 1.3, 2: .1},
   ....:                     "X": dict(zip(range(3), np.random.randn(3)))
   ....:                    })
   ....: 

In [46]: dft["id"] = dft.index

In [47]: dft
Out[47]: 
  A1970 A1980  B1970  B1980         X  id
0     a     d    2.5    3.2 -2.013095   0
1     b     e    1.2    1.3 -1.711797   1
2     c     f    0.7    0.1  0.975018   2

In [48]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")
Out[48]: 
                X  A    B
id year                  
0  1970 -2.013095  a  2.5
1  1970 -1.711797  b  1.2
2  1970  0.975018  c  0.7
0  1980 -2.013095  d  3.2
1  1980 -1.711797  e  1.3
2  1980  0.975018  f  0.1
```

4.5.4 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 [49]: df
Out[49]: 
exp                  A         B                   A
animal             cat       dog       cat       dog
first second                                        
bar   one    -1.436953  0.561814 -0.346880 -0.546060
```

```
      two    -1.993054 -0.689365 -0.877031  1.507935
baz   one    -1.866380  0.043384  0.252683 -0.479004
      two    -0.190425 -1.520207  1.697480  0.923916
foo   one    -1.613149 -0.622599  0.291003  0.792238
      two     0.807151 -0.758613 -2.393856  1.098272
qux   one    -0.168090 -0.243576  0.316513 -0.633213
      two    -0.411186  1.584705 -1.042868 -0.295906

In [50]: df.stack().mean(1).unstack()
Out[50]:
animal             cat       dog
first second
bar   one    -0.891917  0.007877
      two    -1.435043  0.409285
baz   one    -0.806849 -0.217810
      two     0.753528 -0.298146
foo   one    -0.661073  0.084819
      two    -0.793352  0.169830
qux   one     0.074211 -0.438394
      two    -0.727027  0.644399

# same result, another way
In [51]: df.groupby(level=1, axis=1).mean()
Out[51]:
animal             cat       dog
first second
bar   one    -0.891917  0.007877
      two    -1.435043  0.409285
baz   one    -0.806849 -0.217810
      two     0.753528 -0.298146
foo   one    -0.661073  0.084819
      two    -0.793352  0.169830
qux   one     0.074211 -0.438394
      two    -0.727027  0.644399

In [52]: df.stack().groupby(level=1).mean()
Out[52]:
exp            A         B
second
one    -0.743827  0.031543
two     0.180838 -0.499969

In [53]: df.mean().unstack(0)
Out[53]:
exp            A         B
animal
cat    -0.859011 -0.262870
dog     0.296022 -0.205557
```

4.5.5 Pivot tables

While `pivot()` provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides *`pivot_table()`* for pivoting with aggregation of numeric data.

The function *pivot_table()* can be used to create spreadsheet-style pivot tables. See the *cookbook* for some advanced strategies.

It takes a number of arguments:

- `data`: a DataFrame object.
- `values`: a column or a list of columns to aggregate.
- `index`: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- `columns`: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- `aggfunc`: function to use for aggregation, defaulting to `numpy.mean`.

Consider a data set like this:

```
In [54]: import datetime

In [55]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 6,
   ....:                    'B': ['A', 'B', 'C'] * 8,
   ....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
   ....:                    'D': np.random.randn(24),
   ....:                    'E': np.random.randn(24),
   ....:                    'F': [datetime.datetime(2013, i, 1) for i in range(1, 13)]
   ....:                    + [datetime.datetime(2013, i, 15) for i in range(1, 13)]})
   ....:

In [56]: df
Out[56]:
        A  B    C         D         E          F
0     one  A  foo  1.772891 -1.594231 2013-01-01
1     one  B  foo  0.498050 -1.165583 2013-02-01
2     two  C  foo  0.476510  0.451407 2013-03-01
3   three  A  bar  0.877022 -1.724523 2013-04-01
4     one  B  bar -0.301831  2.097509 2013-05-01
5     one  C  bar -0.733711  1.078343 2013-06-01
6     two  A  foo  0.826360 -0.394125 2013-07-01
7   three  B  foo  0.466766 -0.726083 2013-08-01
8     one  C  foo  1.634589  0.490827 2013-09-01
9     one  A  bar  0.958132 -0.559443 2013-10-01
10    two  B  bar -0.970543 -0.395876 2013-11-01
11  three  C  bar -0.932172 -0.927091 2013-12-01
12    one  A  foo -1.273567 -0.417776 2013-01-15
13    one  B  foo  0.270030 -0.708679 2013-02-15
14    two  C  foo -1.388749 -1.557855 2013-03-15
15  three  A  bar  2.090035 -1.742504 2013-04-15
16    one  B  bar -0.815984 -0.201498 2013-05-15
17    one  C  bar -0.020855  0.070382 2013-06-15
18    two  A  foo -0.108292 -0.749683 2013-07-15
19  three  B  foo  0.093700  1.434318 2013-08-15
20    one  C  foo -0.851281  1.226799 2013-09-15
21    one  A  bar -2.154727 -0.706492 2013-10-15
22    two  B  bar -0.120914  0.299114 2013-11-15
23  three  C  bar -0.976377 -0.081937 2013-12-15
```

We can produce pivot tables from this data very easily:

```
In [57]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
```

```
Out[57]:
C              bar       foo
A     B
one   A -0.598297  0.249662
      B -0.558907  0.384040
      C -0.377283  0.391654
three A  1.483528       NaN
      B       NaN  0.280233
      C -0.954274       NaN
two   A       NaN  0.359034
      B -0.545728       NaN
      C       NaN -0.456119

In [58]: pd.pivot_table(df, values='D', index=['B'], columns=['A', 'C'],␣
↪aggfunc=np.sum)
Out[58]:
A        one                 three                 two
C        bar       foo       bar       foo       bar       foo
B
A -1.196595  0.499324  2.967057       NaN       NaN  0.718067
B -1.117815  0.768081       NaN  0.560466 -1.091457       NaN
C -0.754566  0.783308 -1.908549       NaN       NaN -0.912238

In [59]: pd.pivot_table(df, values=['D', 'E'], index=['B'], columns=['A',␣
↪'C'],
   ....:                 aggfunc=np.sum)
   ....:
Out[59]:
          D                                                   ...          E
A        one                 three                 two  ...        one      three ␣
↪               two
C        bar       foo       bar       foo       bar  ...        foo        bar ␣
↪      foo       bar       foo
B                                                     ...
A -1.196595  0.499324  2.967057       NaN       NaN  ... -2.012007 -3.467026 ␣
↪      NaN       NaN -1.143807
B -1.117815  0.768081       NaN  0.560466 -1.091457  ... -1.874262       NaN ␣
↪0.708235 -0.096762       NaN
C -0.754566  0.783308 -1.908549       NaN       NaN  ...  1.717626 -1.009028 ␣
↪      NaN       NaN -1.106447

[3 rows x 12 columns]
```

The result object is a `DataFrame` having potentially hierarchical indexes on the rows and columns. If the `values` column name is not given, the pivot table will include all of the data that can be aggregated in an additional level of hierarchy in the columns:

```
In [60]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])
Out[60]:
                D                   E
C              bar       foo       bar       foo
A     B
one   A -0.598297  0.249662 -0.632968 -1.006004
      B -0.558907  0.384040  0.948005 -0.937131
```

(continues on next page)

(continued from previous page)

```
      C -0.377283  0.391654  0.574362  0.858813
three A  1.483528       NaN -1.733513       NaN
      B       NaN  0.280233       NaN  0.354117
      C -0.954274       NaN -0.504514       NaN
two   A       NaN  0.359034       NaN -0.571904
      B -0.545728       NaN -0.048381       NaN
      C       NaN -0.456119       NaN -0.553224
```

Also, you can use `Grouper` for `index` and `columns` keywords. For detail of `Grouper`, see *Grouping with a Grouper specification*.

```
In [61]: pd.pivot_table(df, values='D', index=pd.Grouper(freq='M', key='F'),
   ....:                columns='C')
   ....:
Out[61]:
C                bar       foo
F
2013-01-31       NaN  0.249662
2013-02-28       NaN  0.384040
2013-03-31       NaN -0.456119
2013-04-30  1.483528       NaN
2013-05-31 -0.558907       NaN
2013-06-30 -0.377283       NaN
2013-07-31       NaN  0.359034
2013-08-31       NaN  0.280233
2013-09-30       NaN  0.391654
2013-10-31 -0.598297       NaN
2013-11-30 -0.545728       NaN
2013-12-31 -0.954274       NaN
```

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```
In [62]: table = pd.pivot_table(df, index=['A', 'B'], columns=['C'])

In [63]: print(table.to_string(na_rep=''))
                D                   E
C             bar       foo       bar       foo
A     B
one   A -0.598297  0.249662 -0.632968 -1.006004
      B -0.558907  0.384040  0.948005 -0.937131
      C -0.377283  0.391654  0.574362  0.858813
three A  1.483528           -1.733513
      B            0.280233            0.354117
      C -0.954274           -0.504514
two   A            0.359034           -0.571904
      B -0.545728           -0.048381
      C           -0.456119           -0.553224
```

Note that `pivot_table` is also available as an instance method on DataFrame, i.e. `DataFrame.pivot_table()`.

Adding margins

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 [64]: df.pivot_table(index=['A', 'B'], columns='C', margins=True, aggfunc=np.std)
Out[64]: 
                D                                E                    
C             bar       foo       All       bar       foo       All
A     B                                                            
one   A  2.201124  2.154171  1.844310  0.103979  0.831880  0.529777
      B  0.363561  0.161235  0.590854  1.625643  0.323080  1.449234
      C  0.504065  1.757775  1.145298  0.712737  0.520411  0.535329
three A  0.857730       NaN  0.857730  0.012714       NaN  0.012714
      B       NaN  0.263797  0.263797       NaN  1.527634  1.527634
      C  0.031258       NaN  0.031258  0.597614       NaN  0.597614
two   A       NaN  0.660899  0.660899       NaN  0.251417  0.251417
      B  0.600778       NaN  0.600778  0.491432       NaN  0.491432
      C       NaN  1.318937  1.318937       NaN  1.420763  1.420763
All      1.126982  1.003301  1.047076  1.082597  1.004895  1.000732
```

4.5.6 Cross tabulations

Use *crosstab()* to compute a cross-tabulation of two (or more) factors. By default crosstab computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- index: array-like, values to group by in the rows.
- columns: array-like, values to group by in the columns.
- values: array-like, optional, array of values to aggregate according to the factors.
- aggfunc: function, optional, If no values array is passed, computes a frequency table.
- rownames: sequence, default None, must match number of row arrays passed.
- colnames: sequence, default None, if passed, must match number of column arrays passed.
- margins: boolean, default False, Add row/column margins (subtotals)
- normalize: boolean, {all, index, columns}, or {0,1}, default False. Normalize by dividing all values by the sum of values.

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

```
In [65]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [66]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [67]: b = np.array([one, one, two, one, two, one], dtype=object)

In [68]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [69]: pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
Out[69]: 
b    one        two      
c   dull shiny dull shiny
a                        
bar    1     0    0     1
foo    2     1    1     0
```

If crosstab receives only two Series, it will provide a frequency table.

```
In [70]: df = pd.DataFrame({'A': [1, 2, 2, 2, 2], 'B': [3, 3, 4, 4, 4],
   ....:                    'C': [1, 1, np.nan, 1, 1]})
   ....:

In [71]: df
Out[71]:
   A  B    C
0  1  3  1.0
1  2  3  1.0
2  2  4  NaN
3  2  4  1.0
4  2  4  1.0

In [72]: pd.crosstab(df.A, df.B)
Out[72]:
B  3  4
A
1  1  0
2  1  3
```

Any input passed containing Categorical data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

```
In [73]: foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])

In [74]: bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])

In [75]: pd.crosstab(foo, bar)
Out[75]:
col_0  d  e
row_0
a      1  0
b      0  1
```

Normalization

New in version 0.18.1.

Frequency tables can also be normalized to show percentages rather than counts using the normalize argument:

```
In [76]: pd.crosstab(df.A, df.B, normalize=True)
Out[76]:
B    3    4
A
1  0.2  0.0
2  0.2  0.6
```

normalize can also normalize values within each row or within each column:

```
In [77]: pd.crosstab(df.A, df.B, normalize='columns')
Out[77]:
B    3    4
A
```

(continues on next page)

(continued from previous page)

```
1  0.5  0.0
2  0.5  1.0
```

`crosstab` can also be passed a third `Series` and an aggregation function (`aggfunc`) that will be applied to the values of the third `Series` within each group defined by the first two `Series`:

```
In [78]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum)
Out[78]:
B    3    4
A
1  1.0  NaN
2  1.0  2.0
```

Adding margins

Finally, one can also add margins or normalize this output.

```
In [79]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum, normalize=True,
   ....:             margins=True)
   ....:
Out[79]:
B       3    4   All
A
1    0.25  0.0  0.25
2    0.25  0.5  0.75
All  0.50  0.5  1.00
```

4.5.7 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 [80]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [81]: pd.cut(ages, bins=3)
Out[81]:
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.
↪95, 26.667], (26.667, 43.333], (43.333, 60.0], (43.333, 60.0]]
Categories (3, interval[float64]): [(9.95, 26.667] < (26.667, 43.333] < (43.333, 60.
↪0]]
```

If the `bins` keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [82]: c = pd.cut(ages, bins=[0, 18, 35, 70])

In [83]: c
Out[83]:
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]]
Categories (3, interval[int64]): [(0, 18] < (18, 35] < (35, 70]]
```

New in version 0.20.0.

If the `bins` keyword is an `IntervalIndex`, then these will be used to bin the passed data.:

```
pd.cut([25, 20, 50], bins=c.categories)
```

4.5.8 Computing indicator / dummy variables

To convert a categorical variable into a dummy or indicator `DataFrame`, for example a column in a `DataFrame` (a `Series`) which has `k` distinct values, can derive a `DataFrame` containing `k` columns of 1s and 0s using *`get_dummies()`*:

```
In [84]: df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})

In [85]: pd.get_dummies(df['key'])
Out[85]: 
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Sometimes its useful to prefix the column names, for example when merging the result with the original `DataFrame`:

```
In [86]: dummies = pd.get_dummies(df['key'], prefix='key')

In [87]: dummies
Out[87]: 
   key_a  key_b  key_c
0      0      1      0
1      0      1      0
2      1      0      0
3      0      0      1
4      1      0      0
5      0      1      0

In [88]: df[['data1']].join(dummies)
Out[88]: 
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

This function is often used along with discretization functions like `cut`:

```
In [89]: values = np.random.randn(10)

In [90]: values
Out[90]: 
array([-0.7350435 ,  1.42920375, -1.11519984, -0.97015174, -1.19270064,
        0.02125661, -0.20556342, -0.66677255,  2.12277401, -0.10814128])

In [91]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

(continues on next page)

(continued from previous page)

```
In [92]: pd.get_dummies(pd.cut(values, bins))
Out[92]: 
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           0
1           0           0           0           0           0
2           0           0           0           0           0
3           0           0           0           0           0
4           0           0           0           0           0
5           1           0           0           0           0
6           0           0           0           0           0
7           0           0           0           0           0
8           0           0           0           0           0
9           0           0           0           0           0
```

See also *Series.str.get_dummies*.

get_dummies() also accepts a DataFrame. By default all categorical variables (categorical in the statistical sense, those with *object* or *categorical* dtype) are encoded as dummy variables.

```
In [93]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
   ....:                    'C': [1, 2, 3]})
   ....: 

In [94]: pd.get_dummies(df)
Out[94]: 
   C  A_a  A_b  B_b  B_c
0  1    1    0    0    1
1  2    0    1    0    1
2  3    1    0    1    0
```

All non-object columns are included untouched in the output. You can control the columns that are encoded with the columns keyword.

```
In [95]: pd.get_dummies(df, columns=['A'])
Out[95]: 
   B  C  A_a  A_b
0  c  1    1    0
1  c  2    0    1
2  b  3    1    0
```

Notice that the B column is still included in the output, it just hasnt been encoded. You can drop B before calling get_dummies if you dont want to include it in the output.

As with the Series version, you can pass values for the prefix and prefix_sep. By default the column name is used as the prefix, and _ as the prefix separator. You can specify prefix and prefix_sep in 3 ways:

- string: Use the same value for prefix or prefix_sep for each column to be encoded.
- list: Must be the same length as the number of columns being encoded.
- dict: Mapping column name to prefix.

```
In [96]: simple = pd.get_dummies(df, prefix='new_prefix')

In [97]: simple
Out[97]: 
   C  new_prefix_a  new_prefix_b  new_prefix_b  new_prefix_c
```

(continues on next page)

(continued from previous page)

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

In [98]: from_list = pd.get_dummies(df, prefix=['from_A', 'from_B'])

In [99]: from_list
Out[99]:
   C  from_A_a  from_A_b  from_B_b  from_B_c
0  1         1         0         0         1
1  2         0         1         0         1
2  3         1         0         1         0

In [100]: from_dict = pd.get_dummies(df, prefix={'B': 'from_B', 'A': 'from_A'})

In [101]: from_dict
Out[101]:
   C  from_A_a  from_A_b  from_B_b  from_B_c
0  1         1         0         0         1
1  2         0         1         0         1
2  3         1         0         1         0
```

New in version 0.18.0.

Sometimes it will be useful to only keep k-1 levels of a categorical variable to avoid collinearity when feeding the result to statistical models. You can switch to this mode by turn on `drop_first`.

```
In [102]: s = pd.Series(list('abcaa'))

In [103]: pd.get_dummies(s)
Out[103]:
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0

In [104]: pd.get_dummies(s, drop_first=True)
Out[104]:
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

When a column contains only one level, it will be omitted in the result.

```
In [105]: df = pd.DataFrame({'A': list('aaaaa'), 'B': list('ababc')})

In [106]: pd.get_dummies(df)
Out[106]:
   A_a  B_a  B_b  B_c
0    1    1    0    0
1    1    0    1    0
```

```
2    1    1    0    0
3    1    0    1    0
4    1    0    0    1

In [107]: pd.get_dummies(df, drop_first=True)
Out[107]: 
   B_b  B_c
0    0    0
1    1    0
2    0    0
3    1    0
4    0    1
```

By default new columns will have `np.uint8` dtype. To choose another dtype, use the `dtype` argument:

```
In [108]: df = pd.DataFrame({'A': list('abc'), 'B': [1.1, 2.2, 3.3]})

In [109]: pd.get_dummies(df, dtype=bool).dtypes
Out[109]: 
B      float64
A_a       bool
A_b       bool
A_c       bool
dtype: object
```

New in version 0.23.0.

4.5.9 Factorizing values

To encode 1-d values as an enumerated type use *`factorize()`*:

```
In [110]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])

In [111]: x
Out[111]: 
0       A
1       A
2     NaN
3       B
4    3.14
5     inf
dtype: object

In [112]: labels, uniques = pd.factorize(x)

In [113]: labels
Out[113]: array([ 0,  0, -1,  1,  2,  3])

In [114]: uniques
Out[114]: Index(['A', 'B', 3.14, inf], dtype='object')
```

Note that `factorize` is similar to `numpy.unique`, but differs in its handling of NaN:

Note: The following `numpy.unique` will fail under Python 3 with a `TypeError` because of an ordering bug. See also here.

```
In [1]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])
In [2]: pd.factorize(x, sort=True)
Out[2]:
(array([ 2,  2, -1,  3,  0,  1]),
 Index([3.14, inf, 'A', 'B'], dtype='object'))

In [3]: np.unique(x, return_inverse=True)[::-1]
Out[3]: (array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
```

Note: If you just want to handle one column as a categorical variable (like Rs factor), you can use `df["cat_col"] = pd.Categorical(df["col"])` or `df["cat_col"] = df["col"].astype("category")`. For full docs on *`Categorical`*, see the *Categorical introduction* and the *API documentation*.

4.5.10 Examples

In this section, we will review frequently asked questions and examples. The column names and relevant column values are named to correspond with how this DataFrame will be pivoted in the answers below.

```
In [115]: np.random.seed([3, 1415])

In [116]: n = 20

In [117]: cols = np.array(['key', 'row', 'item', 'col'])

In [118]: df = cols + pd.DataFrame((np.random.randint(5, size=(n, 4))
   .....:                            // [2, 1, 2, 1]).astype(str))
   .....:

In [119]: df.columns = cols

In [120]: df = df.join(pd.DataFrame(np.random.rand(n, 2).round(2)).add_prefix('val'))

In [121]: df
Out[121]:
     key   row   item   col  val0  val1
0   key0  row3  item1  col3  0.81  0.04
1   key1  row2  item1  col2  0.44  0.07
2   key1  row0  item1  col0  0.77  0.01
3   key0  row4  item0  col2  0.15  0.59
4   key1  row0  item2  col1  0.81  0.64
5   key1  row2  item2  col4  0.13  0.88
6   key2  row4  item1  col3  0.88  0.39
7   key1  row4  item1  col1  0.10  0.07
8   key1  row0  item2  col4  0.65  0.02
9   key1  row2  item0  col2  0.35  0.61
10  key2  row0  item2  col1  0.40  0.85
11  key2  row4  item1  col2  0.64  0.25
12  key0  row2  item2  col3  0.50  0.44
13  key0  row4  item1  col4  0.24  0.46
14  key1  row3  item2  col3  0.28  0.11
15  key0  row3  item1  col1  0.31  0.23
```

(continues on next page)

(continued from previous page)

```
16  key0  row0  item2  col3  0.86  0.01
17  key0  row4  item0  col3  0.64  0.21
18  key2  row2  item2  col0  0.13  0.45
19  key0  row2  item0  col4  0.37  0.70
```

Pivoting with single aggregations

Suppose we wanted to pivot `df` such that the `col` values are columns, `row` values are the index, and the mean of `val0` are the values? In particular, the resulting DataFrame should look like:

Note: col col0 col1 col2 col3 col4 row row0 0.77 0.605 NaN 0.860 0.65 row2 0.13 NaN 0.395 0.500 0.25 row3 NaN 0.310 NaN 0.545 NaN row4 NaN 0.100 0.395 0.760 0.24

This solution uses *pivot_table()*. Also note that `aggfunc='mean'` is the default. It is included here to be explicit.

```
In [122]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean')
   .....:
Out[122]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65
row2  0.13    NaN  0.395  0.500  0.25
row3   NaN  0.310    NaN  0.545   NaN
row4   NaN  0.100  0.395  0.760  0.24
```

Note that we can also replace the missing values by using the `fill_value` parameter.

```
In [123]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean', fill_value=0)
   .....:
Out[123]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605  0.000  0.860  0.65
row2  0.13  0.000  0.395  0.500  0.25
row3  0.00  0.310  0.000  0.545  0.00
row4  0.00  0.100  0.395  0.760  0.24
```

Also note that we can pass in other aggregation functions as well. For example, we can also pass in `sum`.

```
In [124]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='sum', fill_value=0)
   .....:
Out[124]:
col   col0  col1  col2  col3  col4
row
row0  0.77  1.21  0.00  0.86  0.65
row2  0.13  0.00  0.79  0.50  0.50
row3  0.00  0.31  0.00  1.09  0.00
row4  0.00  0.10  0.79  1.52  0.24
```

Another aggregation we can do is calculate the frequency in which the columns and rows occur together a.k.a. cross tabulation. To do this, we can pass `size` to the `aggfunc` parameter.

```
In [125]: df.pivot_table(index='row', columns='col', fill_value=0, aggfunc='size')
Out[125]: 
col   col0  col1  col2  col3  col4
row                               
row0     1     2     0     1     1
row2     1     0     2     1     2
row3     0     1     0     2     0
row4     0     1     2     2     1
```

Pivoting with multiple aggregations

We can also perform multiple aggregations. For example, to perform both a `sum` and `mean`, we can pass in a list to the `aggfunc` argument.

```
In [126]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc=['mean', 'sum'])
   .....: 
Out[126]: 
      mean                                  sum                        
col   col0   col1   col2   col3  col4  col0  col1  col2  col3  col4
row                                                                
row0  0.77  0.605    NaN  0.860  0.65  0.77  1.21   NaN  0.86  0.65
row2  0.13    NaN  0.395  0.500  0.25  0.13   NaN  0.79  0.50  0.50
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.31   NaN  1.09   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.10  0.79  1.52  0.24
```

Note to aggregate over multiple value columns, we can pass in a list to the `values` parameter.

```
In [127]: df.pivot_table(
   .....:     values=['val0', 'val1'], index='row', columns='col', aggfunc=['mean'])
   .....: 
Out[127]: 
      mean                                                          
      val0                                val1                      
col   col0   col1   col2   col3  col4  col0   col1  col2   col3  col4
row                                                                 
row0  0.77  0.605    NaN  0.860  0.65  0.01  0.745   NaN  0.010  0.02
row2  0.13    NaN  0.395  0.500  0.25  0.45    NaN  0.34  0.440  0.79
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.230   NaN  0.075   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.070  0.42  0.300  0.46
```

Note to subdivide over multiple columns we can pass in a list to the `columns` parameter.

```
In [128]: df.pivot_table(
   .....:     values=['val0'], index='row', columns=['item', 'col'], aggfunc=['mean'])
   .....: 
Out[128]: 
      mean                                                                 
      val0                                                                 
item item0             item1                         item2                  
col   col2  col3  col4  col0  col1  col2  col3  col4  col0   col1  col3  col4
row                                                                         
row0   NaN   NaN   NaN  0.77   NaN   NaN   NaN   NaN   NaN  0.605  0.86  0.65
```

(continues on next page)

(continued from previous page)

```
row2  0.35   NaN  0.37   NaN   NaN  0.44   NaN   NaN  0.13    NaN  0.50  0.13
row3   NaN   NaN   NaN   NaN  0.31   NaN  0.81   NaN   NaN    NaN  0.28   NaN
row4  0.15  0.64   NaN   NaN  0.10  0.64  0.88  0.24   NaN    NaN   NaN   NaN
```

4.5.11 Exploding a list-like column

New in version 0.25.0.

Sometimes the values in a column are list-like.

```
In [129]: keys = ['panda1', 'panda2', 'panda3']

In [130]: values = [['eats', 'shoots'], ['shoots', 'leaves'], ['eats', 'leaves']]

In [131]: df = pd.DataFrame({'keys': keys, 'values': values})

In [132]: df
Out[132]: 
     keys            values
0  panda1    [eats, shoots]
1  panda2  [shoots, leaves]
2  panda3    [eats, leaves]
```

We can explode the `values` column, transforming each list-like to a separate row, by using `explode()`. This will replicate the index values from the original row:

```
In [133]: df['values'].explode()
Out[133]: 
0      eats
0    shoots
1    shoots
1    leaves
2      eats
2    leaves
Name: values, dtype: object
```

You can also explode the column in the `DataFrame`.

```
In [134]: df.explode('values')
Out[134]: 
     keys  values
0  panda1    eats
0  panda1  shoots
1  panda2  shoots
1  panda2  leaves
2  panda3    eats
2  panda3  leaves
```

`Series.explode()` will replace empty lists with `np.nan` and preserve scalar entries. The dtype of the resulting `Series` is always `object`.

```
In [135]: s = pd.Series([[1, 2, 3], 'foo', [], ['a', 'b']])

In [136]: s
Out[136]: 
0    [1, 2, 3]
```

```
1          foo
2           []
3       [a, b]
dtype: object

In [137]: s.explode()
Out[137]:
0      1
0      2
0      3
1    foo
2    NaN
3      a
3      b
dtype: object
```

Here is a typical usecase. You have comma separated strings in a column and want to expand this.

```
In [138]: df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1},
   .....:                    {'var1': 'd,e,f', 'var2': 2}])
   .....:

In [139]: df
Out[139]:
    var1  var2
0  a,b,c     1
1  d,e,f     2
```

Creating a long form DataFrame is now straightforward using explode and chained operations

```
In [140]: df.assign(var1=df.var1.str.split(',')).explode('var1')
Out[140]:
  var1  var2
0    a     1
0    b     1
0    c     1
1    d     2
1    e     2
1    f     2
```

{{ header }}

4.6 Working with text data

Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the `str` attribute and generally have names matching the equivalent (scalar) built-in string methods:

```
In [1]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog',␣
 ↪'cat'])

In [2]: s.str.lower()
Out[2]:
0        a
1        b
```

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

In [3]: s.str.upper()
Out[3]:
0       A
1       B
2       C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object

In [4]: s.str.len()
Out[4]:
0    1.0
1    1.0
2    1.0
3    4.0
4    4.0
5    NaN
6    4.0
7    3.0
8    3.0
dtype: float64
```

```
In [5]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

In [6]: idx.str.strip()
Out[6]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [7]: idx.str.lstrip()
Out[7]: Index(['jack', 'jill ', 'jesse ', 'frank'], dtype='object')

In [8]: idx.str.rstrip()
Out[8]: Index([' jack', 'jill', ' jesse', 'frank'], dtype='object')
```

The string methods on Index are especially useful for cleaning up or transforming DataFrame columns. For instance, you may have columns with leading or trailing whitespace:

```
In [9]: df = pd.DataFrame(np.random.randn(3, 2),
   ...:                   columns=[' Column A ', ' Column B '], index=range(3))
   ...:

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

(continues on next page)

(continued from previous page)

```
    Column A    Column B 
0   -0.061542    0.528296
1   -0.952849    2.392803
2   -1.524186    2.642204
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```
In [11]: df.columns.str.strip()
Out[11]: Index(['Column A', 'Column B'], dtype='object')

In [12]: df.columns.str.lower()
Out[12]: Index([' column a ', ' column b '], dtype='object')
```

These string methods can then be used to clean up the columns as needed. Here we are removing leading and trailing whitespaces, lower casing all names, and replacing any remaining whitespaces with underscores:

```
In [13]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [14]: df
Out[14]: 
   column_a  column_b
0 -0.061542  0.528296
1 -0.952849  2.392803
2 -1.524186  2.642204
```

Note: If you have a `Series` where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`), it can be faster to convert the original `Series` to one of type `category` and then use `.str.<method>` or `.dt.<property>` on that. The performance difference comes from the fact that, for `Series` of type `category`, the string operations are done on the `.categories` and not on each element of the `Series`.

Please note that a `Series` of type `category` with string `.categories` has some limitations in comparison to `Series` of type string (e.g. you cant add strings to each other: `s + " " + s` wont work if `s` is a `Series` of type `category`). Also, `.str` methods which operate on elements of type `list` are not available on such a `Series`.

Warning: Before v.0.25.0, the `.str`-accessor did only the most rudimentary type checks. Starting with v.0.25.0, the type of the Series is inferred and the allowed types (i.e. strings) are enforced more rigorously.

Generally speaking, the `.str` accessor is intended to work only on strings. With very few exceptions, other uses are not supported, and may be disabled at a later point.

4.6.1 Splitting and replacing strings

Methods like `split` return a Series of lists:

```
In [15]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [16]: s2.str.split('_')
Out[16]: 
0    [a, b, c]
1    [c, d, e]
2          NaN
```

(continues on next page)

(continued from previous page)

```
3    [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using `get` or `[]` notation:

```
In [17]: s2.str.split('_').str.get(1)
Out[17]:
0      b
1      d
2    NaN
3      g
dtype: object

In [18]: s2.str.split('_').str[1]
Out[18]:
0      b
1      d
2    NaN
3      g
dtype: object
```

It is easy to expand this to return a DataFrame using `expand`.

```
In [19]: s2.str.split('_', expand=True)
Out[19]:
     0    1    2
0    a    b    c
1    c    d    e
2  NaN  NaN  NaN
3    f    g    h
```

It is also possible to limit the number of splits:

```
In [20]: s2.str.split('_', expand=True, n=1)
Out[20]:
     0    1
0    a  b_c
1    c  d_e
2  NaN  NaN
3    f  g_h
```

`rsplit` is similar to `split` except it works in the reverse direction, i.e., from the end of the string to the beginning of the string:

```
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
     0    1
0  a_b    c
1  c_d    e
2  NaN  NaN
3  f_g    h
```

`replace` by default replaces regular expressions:

```
In [22]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca',
   ....:                 '', np.nan, 'CABA', 'dog', 'cat'])
   ....:
```

```
In [23]: s3
Out[23]:
0       A
1       B
2       C
3    Aaba
4    Baca
5
6     NaN
7    CABA
8     dog
9     cat
dtype: object

In [24]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
Out[24]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6         NaN
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: object
```

Some caution must be taken to keep regular expressions in mind! For example, the following code will cause trouble because of the regular expression meaning of *$*:

```
# Consider the following badly formatted financial data
In [25]: dollars = pd.Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
Out[26]:
0        12
1       -10
2    10,000
dtype: object

# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
Out[27]:
0         12
1       -$10
2    $10,000
dtype: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'-\$', '-')
Out[28]:
0         12
```

```
1        -10
2    $10,000
dtype: object
```

New in version 0.23.0.

If you do want literal replacement of a string (equivalent to `str.replace()`), you can set the optional `regex` parameter to `False`, rather than escaping each character. In this case both `pat` and `repl` must be strings:

```
# These lines are equivalent
In [29]: dollars.str.replace(r'-\$', '-')
Out[29]:
0         12
1        -10
2    $10,000
dtype: object

In [30]: dollars.str.replace('-$', '-', regex=False)
Out[30]:
0         12
1        -10
2    $10,000
dtype: object
```

New in version 0.20.0.

The `replace` method can also take a callable as replacement. It is called on every `pat` using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

```
# Reverse every lowercase alphabetic word
In [31]: pat = r'[a-z]+'

In [32]: def repl(m):
   ....:     return m.group(0)[::-1]
   ....:

In [33]: pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(pat, repl)
Out[33]:
0    oof 123
1    rab zab
2        NaN
dtype: object

# Using regex groups
In [34]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"

In [35]: def repl(m):
   ....:     return m.group('two').swapcase()
   ....:

In [36]: pd.Series(['Foo Bar Baz', np.nan]).str.replace(pat, repl)
Out[36]:
0    bAR
1    NaN
dtype: object
```

New in version 0.20.0.

The `replace` method also accepts a compiled regular expression object from `re.compile()` as a pattern. All flags should be included in the compiled regular expression object.

```
In [37]: import re

In [38]: regex_pat = re.compile(r'^.a|dog', flags=re.IGNORECASE)

In [39]: s3.str.replace(regex_pat, 'XX-XX ')
Out[39]: 
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
```

Including a flags argument when calling replace with a compiled regular expression object will raise a ValueError.

```
In [40]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

4.6.2 Concatenation

There are several ways to concatenate a Series or Index, either with itself or others, all based on cat(), resp. Index.str.cat.

Concatenating a single Series into a string

The content of a Series (or Index) can be concatenated:

```
In [41]: s = pd.Series(['a', 'b', 'c', 'd'])

In [42]: s.str.cat(sep=',')
Out[42]: 'a,b,c,d'
```

If not specified, the keyword sep for the separator defaults to the empty string, sep='':

```
In [43]: s.str.cat()
Out[43]: 'abcd'
```

By default, missing values are ignored. Using na_rep, they can be given a representation:

```
In [44]: t = pd.Series(['a', 'b', np.nan, 'd'])

In [45]: t.str.cat(sep=',')
Out[45]: 'a,b,d'

In [46]: t.str.cat(sep=',', na_rep='-')
Out[46]: 'a,b,-,d'
```

Concatenating a Series and something list-like into a Series

The first argument to `cat()` can be a list-like object, provided that it matches the length of the calling `Series` (or `Index`).

```
In [47]: s.str.cat(['A', 'B', 'C', 'D'])
Out[47]:
0    aA
1    bB
2    cC
3    dD
dtype: object
```

Missing values on either side will result in missing values in the result as well, *unless* `na_rep` is specified:

```
In [48]: s.str.cat(t)
Out[48]:
0     aa
1     bb
2    NaN
3     dd
dtype: object

In [49]: s.str.cat(t, na_rep='-')
Out[49]:
0    aa
1    bb
2    c-
3    dd
dtype: object
```

Concatenating a Series and something array-like into a Series

New in version 0.23.0.

The parameter `others` can also be two-dimensional. In this case, the number or rows must match the lengths of the calling `Series` (or `Index`).

```
In [50]: d = pd.concat([t, s], axis=1)

In [51]: s
Out[51]:
0    a
1    b
2    c
3    d
dtype: object

In [52]: d
Out[52]:
     0  1
0    a  a
1    b  b
2  NaN  c
3    d  d
```

```
In [53]: s.str.cat(d, na_rep='-')
Out[53]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and an indexed object into a Series, with alignment

New in version 0.23.0.

For concatenation with a `Series` or `DataFrame`, it is possible to align the indexes before concatenation by setting the `join`-keyword.

```
In [54]: u = pd.Series(['b', 'd', 'a', 'c'], index=[1, 3, 0, 2])

In [55]: s
Out[55]:
0    a
1    b
2    c
3    d
dtype: object

In [56]: u
Out[56]:
1    b
3    d
0    a
2    c
dtype: object

In [57]: s.str.cat(u)
Out[57]:
0    ab
1    bd
2    ca
3    dc
dtype: object

In [58]: s.str.cat(u, join='left')
Out[58]:
0    aa
1    bb
2    cc
3    dd
dtype: object
```

Warning: If the `join` keyword is not passed, the method `cat()` will currently fall back to the behavior before version 0.23.0 (i.e. no alignment), but a `FutureWarning` will be raised if any of the involved indexes differ, since this default will change to `join='left'` in a future version.

The usual options are available for `join` (one of `'left', 'outer', 'inner', 'right'`). In particular, alignment also means that the different lengths do not need to coincide anymore.

```
In [59]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4])

In [60]: s
Out[60]:
0    a
1    b
2    c
3    d
dtype: object

In [61]: v
Out[61]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: object

In [62]: s.str.cat(v, join='left', na_rep='-')
Out[62]:
0    aa
1    bb
2    c-
3    dd
dtype: object

In [63]: s.str.cat(v, join='outer', na_rep='-')
Out[63]:
-1    -z
 0    aa
 1    bb
 2    c-
 3    dd
 4    -e
dtype: object
```

The same alignment can be used when `others` is a `DataFrame`:

```
In [64]: f = d.loc[[3, 2, 1, 0], :]

In [65]: s
Out[65]:
0    a
1    b
2    c
3    d
dtype: object

In [66]: f
Out[66]:
     0  1
```

```
3    d  d
2  NaN  c
1    b  b
0    a  a

In [67]: s.str.cat(f, join='left', na_rep='-')
Out[67]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and many objects into a Series

Several array-like items (specifically: `Series`, `Index`, and 1-dimensional variants of `np.ndarray`) can be combined in a list-like container (including iterators, `dict`-views, etc.).

```
In [68]: s
Out[68]:
0    a
1    b
2    c
3    d
dtype: object

In [69]: u
Out[69]:
1    b
3    d
0    a
2    c
dtype: object

In [70]: s.str.cat([u, u.to_numpy()], join='left')
Out[70]:
0    aab
1    bbd
2    cca
3    ddc
dtype: object
```

All elements without an index (e.g. `np.ndarray`) within the passed list-like must match in length to the calling `Series` (or `Index`), but `Series` and `Index` may have arbitrary length (as long as alignment is not disabled with `join=None`):

```
In [71]: v
Out[71]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: object
```

```
In [72]: s.str.cat([v, u, u.to_numpy()], join='outer', na_rep='-')
Out[72]:
-1    -z--
 0    aaab
 1    bbbd
 2    c-ca
 3    dddc
 4    -e--
dtype: object
```

If using `join='right'` on a list-like of `others` that contains different indexes, the union of these indexes will be used as the basis for the final concatenation:

```
In [73]: u.loc[[3]]
Out[73]:
3    d
dtype: object

In [74]: v.loc[[-1, 0]]
Out[74]:
-1    z
 0    a
dtype: object

In [75]: s.str.cat([u.loc[[3]], v.loc[[-1, 0]]], join='right', na_rep='-')
Out[75]:
-1    --z
 0    a-a
 3    dd-
dtype: object
```

4.6.3 Indexing with `.str`

You can use `[]` notation to directly index by position locations. If you index past the end of the string, the result will be a `NaN`.

```
In [76]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
   ....:                 'CABA', 'dog', 'cat'])
   ....:

In [77]: s.str[0]
Out[77]:
0      A
1      B
2      C
3      A
4      B
5    NaN
6      C
7      d
8      c
dtype: object

In [78]: s.str[1]
Out[78]:
```

```
0    NaN
1    NaN
2    NaN
3      a
4      a
5    NaN
6      A
7      o
8      a
dtype: object
```

4.6.4 Extracting substrings

Extract first match in each subject (extract)

Warning: In version 0.18.0, `extract` gained the `expand` argument. When `expand=False` it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0). When `expand=True` it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user. `expand=True` is the default since version 0.23.0.

The `extract` method accepts a regular expression with at least one capture group.

Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [79]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'([ab])(\d)', expand=False)
Out[79]: 
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

Elements that do not match return a row filled with `NaN`. Thus, a Series of messy strings can be converted into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects. The dtype of the result is always object, even if no match is found and the result only contains `NaN`.

Named groups like

```
In [80]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'(?P<letter>[ab])(?P<digit>\d)',
   ....:                                            expand=False)
   ....: 
Out[80]: 
  letter digit
0      a     1
1      b     2
2    NaN   NaN
```

and optional groups like

```
In [81]: pd.Series(['a1', 'b2', '3']).str.extract(r'([ab])?(\d)', expand=False)
Out[81]: 
     0  1
0    a  1
```

(continues on next page)

(continued from previous page)

```
1    b  2
2  NaN  3
```

can also be used. Note that any capture group names in the regular expression will be used for column names; otherwise capture group numbers will be used.

Extracting a regular expression with one group returns a DataFrame with one column if expand=True.

```
In [82]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=True)
Out[82]:
     0
0    1
1    2
2  NaN
```

It returns a Series if expand=False.

```
In [83]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=False)
Out[83]:
0      1
1      2
2    NaN
dtype: object
```

Calling on an Index with a regex with exactly one capture group returns a DataFrame with one column if expand=True.

```
In [84]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])

In [85]: s
Out[85]:
A11    a1
B22    b2
C33    c3
dtype: object

In [86]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[86]:
  letter
0      A
1      B
2      C
```

It returns an Index if expand=False.

```
In [87]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[87]: Index(['A', 'B', 'C'], dtype='object', name='letter')
```

Calling on an Index with a regex with more than one capture group returns a DataFrame if expand=True.

```
In [88]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[88]:
  letter   1
0      A  11
1      B  22
2      C  33
```

It raises `ValueError` if `expand=False`.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

The table below summarizes the behavior of `extract(expand=False)` (input subject in first column, number of groups in regex in first row)

	1 group	>1 group
Index	Index	ValueError
Series	Series	DataFrame

Extract all matches in each subject (extractall)

New in version 0.18.0.

Unlike `extract` (which returns only the first match),

```
In [89]: s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])

In [90]: s
Out[90]:
A    a1a2
B      b1
C      c1
dtype: object

In [91]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'

In [92]: s.str.extract(two_groups, expand=True)
Out[92]:
  letter digit
A      a     1
B      b     1
C      c     1
```

the `extractall` method returns every match. The result of `extractall` is always a `DataFrame` with a `MultiIndex` on its rows. The last level of the `MultiIndex` is named `match` and indicates the order in the subject.

```
In [93]: s.str.extractall(two_groups)
Out[93]:
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
C 0          c     1
```

When each subject string in the Series has exactly one match,

```
In [94]: s = pd.Series(['a3', 'b3', 'c2'])

In [95]: s
Out[95]:
0    a3
```

(continues on next page)

(continued from previous page)

```
1    b3
2    c2
dtype: object
```

then `extractall(pat).xs(0, level='match')` gives the same result as `extract(pat)`.

```
In [96]: extract_result = s.str.extract(two_groups, expand=True)

In [97]: extract_result
Out[97]:
  letter digit
0      a     3
1      b     3
2      c     2

In [98]: extractall_result = s.str.extractall(two_groups)

In [99]: extractall_result
Out[99]:
        letter digit
  match
0 0          a     3
1 0          b     3
2 0          c     2

In [100]: extractall_result.xs(0, level="match")
Out[100]:
  letter digit
0      a     3
1      b     3
2      c     2
```

`Index` also supports `.str.extractall`. It returns a `DataFrame` which has the same result as a `Series.str.extractall` with a default index (starts from 0).

New in version 0.19.0.

```
In [101]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[101]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1

In [102]: pd.Series(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[102]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1
```

4.6.5 Testing for Strings that match or contain a pattern

You can check whether elements contain a pattern:

```
In [103]: pattern = r'[0-9][a-z]'

In [104]: pd.Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
Out[104]:
0    False
1    False
2     True
3     True
4     True
dtype: bool
```

Or whether elements match a pattern:

```
In [105]: pd.Series(['1', '2', '3a', '3b', '03c']).str.match(pattern)
Out[105]:
0    False
1    False
2     True
3     True
4    False
dtype: bool
```

The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match`, while `contains` relies on `re.search`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```
In [106]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat
↪'])

In [107]: s4.str.contains('A', na=False)
Out[107]:
0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
dtype: bool
```

4.6.6 Creating indicator variables

You can extract dummy variables from string columns. For example if they are separated by a `'|'`:

```
In [108]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])

In [109]: s.str.get_dummies(sep='|')
Out[109]:
```

(continues on next page)

(continued from previous page)

```
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1
```

String Index also supports get_dummies which returns a MultiIndex.

New in version 0.18.1.

```
In [110]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])

In [111]: idx.str.get_dummies(sep='|')
Out[111]:
MultiIndex([(1, 0, 0),
            (1, 1, 0),
            (0, 0, 0),
            (1, 0, 1)],
           names=['a', 'b', 'c'])
```

See also *get_dummies()*.

4.6.7 Method summary

Method	Description
cat()	Concatenate strings
split()	Split strings on delimiter
rsplit()	Split strings on delimiter working from the end of the string
get()	Index into each element (retrieve i-th element)
join()	Join strings in each element of the Series with passed separator
get_dummies()	Split strings on the delimiter returning DataFrame of dummy variables
contains()	Return boolean array if each string contains pattern/regex
replace()	Replace occurrences of pattern/regex/string with some other string or the return value of a callable given the occurrence
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 str.center
ljust()	Equivalent to str.ljust
rjust()	Equivalent to str.rjust
zfill()	Equivalent to str.zfill
wrap()	Split long strings into lines with length less than a given width
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
endswith()	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
extract()	Call re.search on each element, returning DataFrame with one row for each element and one column for each regex capture group

Continued on next page

Table 2 – continued from previous page

Method	Description
extractall()	Call re.findall on each element, returning DataFrame with one row for each match and one column for each regex capture group
len()	Compute string lengths
strip()	Equivalent to str.strip
rstrip()	Equivalent to str.rstrip
lstrip()	Equivalent to str.lstrip
partition()	Equivalent to str.partition
rpartition()	Equivalent to str.rpartition
lower()	Equivalent to str.lower
casefold()	Equivalent to str.casefold
upper()	Equivalent to str.upper
find()	Equivalent to str.find
rfind()	Equivalent to str.rfind
index()	Equivalent to str.index
rindex()	Equivalent to str.rindex
capitalize()	Equivalent to str.capitalize
swapcase()	Equivalent to str.swapcase
normalize()	Return Unicode normal form. Equivalent to unicodedata.normalize
translate()	Equivalent to str.translate
isalnum()	Equivalent to str.isalnum
isalpha()	Equivalent to str.isalpha
isdigit()	Equivalent to str.isdigit
isspace()	Equivalent to str.isspace
islower()	Equivalent to str.islower
isupper()	Equivalent to str.isupper
istitle()	Equivalent to str.istitle
isnumeric()	Equivalent to str.isnumeric
isdecimal()	Equivalent to str.isdecimal

{{ header }}

4.7 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.

4.7.1 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 not available or NA.

Note: If you want to consider `inf` and `-inf` to be NA in computations, you can set `pandas.options.mode.use_inf_as_na = True`.

```
In [1]: df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   ...:                    columns=['one', 'two', 'three'])
   ...: 

In [2]: df['four'] = 'bar'

In [3]: df['five'] = df['one'] > 0

In [4]: df
Out[4]: 
        one       two     three four  five
a  0.070821 -0.093641  0.014099  bar  True
c  0.702961  0.870484 -1.521966  bar  True
e  0.219802  1.546457 -0.174262  bar  True
f  0.831417  1.332054  0.438532  bar  True
h  0.786117 -0.304618  0.956171  bar  True

In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
Out[6]: 
        one       two     three four  five
a  0.070821 -0.093641  0.014099  bar  True
b       NaN       NaN       NaN  NaN   NaN
c  0.702961  0.870484 -1.521966  bar  True
d       NaN       NaN       NaN  NaN   NaN
e  0.219802  1.546457 -0.174262  bar  True
f  0.831417  1.332054  0.438532  bar  True
g       NaN       NaN       NaN  NaN   NaN
h  0.786117 -0.304618  0.956171  bar  True
```

To make detecting missing values easier (and across different array dtypes), pandas provides the `isna()` and `notna()` functions, which are also methods on Series and DataFrame objects:

```
In [7]: df2['one']
Out[7]: 
a    0.070821
b         NaN
c    0.702961
d         NaN
e    0.219802
f    0.831417
g         NaN
h    0.786117
Name: one, dtype: float64

In [8]: pd.isna(df2['one'])
Out[8]: 
a    False
b     True
c    False
d     True
```

```
e    False
f    False
g     True
h    False
Name: one, dtype: bool

In [9]: df2['four'].notna()
Out[9]:
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: four, dtype: bool

In [10]: df2.isna()
Out[10]:
     one    two  three   four   five
a  False  False  False  False  False
b   True   True   True   True   True
c  False  False  False  False  False
d   True   True   True   True   True
e  False  False  False  False  False
f  False  False  False  False  False
g   True   True   True   True   True
h  False  False  False  False  False
```

Warning: One has to be mindful that in Python (and NumPy), the `nan's` dont compare equal, but `None's` **do**. Note that pandas/NumPy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`.

```
In [11]: None == None                                                    #␣
↪noqa: E711
Out[11]: True

In [12]: np.nan == np.nan
Out[12]: False
```

So as compared to above, a scalar equality comparison versus a `None/np.nan` doesnt provide useful information.

```
In [13]: df2['one'] == np.nan
Out[13]:
a    False
b    False
c    False
d    False
e    False
f    False
g    False
h    False
Name: one, dtype: bool
```

Integer dtypes and missing data

Because `NaN` is a float, a column of integers with even one missing values is cast to floating-point dtype (see *Support for integer NA* for more). Pandas provides a nullable integer array, which can be used by explicitly requesting the dtype:

```
In [14]: pd.Series([1, 2, np.nan, 4], dtype=pd.Int64Dtype())
Out[14]:
0      1
1      2
2    NaN
3      4
dtype: Int64
```

Alternatively, the string alias `dtype='Int64'` (note the capital `"I"`) can be used.

See *Nullable integer data type* for more.

Datetimes

For datetime64[ns] types, `NaT` represents missing values. This is a pseudo-native sentinel value that can be represented by NumPy in a singular dtype (datetime64[ns]). pandas objects provide compatibility between `NaT` and `NaN`.

```
In [15]: df2 = df.copy()

In [16]: df2['timestamp'] = pd.Timestamp('20120101')

In [17]: df2
Out[17]:
        one       two     three four  five  timestamp
a  0.070821 -0.093641  0.014099  bar  True 2012-01-01
c  0.702961  0.870484 -1.521966  bar  True 2012-01-01
e  0.219802  1.546457 -0.174262  bar  True 2012-01-01
f  0.831417  1.332054  0.438532  bar  True 2012-01-01
h  0.786117 -0.304618  0.956171  bar  True 2012-01-01

In [18]: df2.loc[['a', 'c', 'h'], ['one', 'timestamp']] = np.nan

In [19]: df2
Out[19]:
        one       two     three four  five  timestamp
a       NaN -0.093641  0.014099  bar  True        NaT
c       NaN  0.870484 -1.521966  bar  True        NaT
e  0.219802  1.546457 -0.174262  bar  True 2012-01-01
f  0.831417  1.332054  0.438532  bar  True 2012-01-01
h       NaN -0.304618  0.956171  bar  True        NaT

In [20]: df2.dtypes.value_counts()
Out[20]:
float64           3
datetime64[ns]    1
object            1
bool              1
dtype: int64
```

Inserting missing data

You can insert missing values by simply assigning to containers. The actual missing value used will be chosen based on the dtype.

For example, numeric containers will always use NaN regardless of the missing value type chosen:

```
In [21]: s = pd.Series([1, 2, 3])

In [22]: s.loc[0] = None

In [23]: s
Out[23]: 
0    NaN
1    2.0
2    3.0
dtype: float64
```

Likewise, datetime containers will always use NaT.

For object containers, pandas will use the value given:

```
In [24]: s = pd.Series(["a", "b", "c"])

In [25]: s.loc[0] = None

In [26]: s.loc[1] = np.nan

In [27]: s
Out[27]: 
0    None
1     NaN
2       c
dtype: object
```

Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [28]: a
Out[28]: 
        one       two
a       NaN -0.093641
c       NaN  0.870484
e  0.219802  1.546457
f  0.831417  1.332054
h  0.831417 -0.304618

In [29]: b
Out[29]: 
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.870484 -1.521966
e  0.219802  1.546457 -0.174262
f  0.831417  1.332054  0.438532
h       NaN -0.304618  0.956171
```

```
In [30]: a + b
Out[30]:
        one  three       two
a       NaN    NaN -0.187283
c       NaN    NaN  1.740968
e  0.439605    NaN  3.092914
f  1.662833    NaN  2.664108
h       NaN    NaN -0.609235
```

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 0.
- Cumulative methods like `cumsum()` and `cumprod()` ignore NA values by default, but preserve them in the resulting arrays. To override this behaviour and include NA values, use `skipna=False`.

```
In [31]: df
Out[31]:
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.870484 -1.521966
e  0.219802  1.546457 -0.174262
f  0.831417  1.332054  0.438532
h       NaN -0.304618  0.956171

In [32]: df['one'].sum()
Out[32]: 1.0512188858617544

In [33]: df.mean(1)
Out[33]:
a   -0.039771
c   -0.325741
e    0.530666
f    0.867334
h    0.325777
dtype: float64

In [34]: df.cumsum()
Out[34]:
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.776843 -1.507866
e  0.219802  2.323300 -1.682128
f  1.051219  3.655354 -1.243596
h       NaN  3.350736 -0.287425

In [35]: df.cumsum(skipna=False)
Out[35]:
   one       two     three
a  NaN -0.093641  0.014099
c  NaN  0.776843 -1.507866
e  NaN  2.323300 -1.682128
```

```
f  NaN  3.655354 -1.243596
h  NaN  3.350736 -0.287425
```

4.7.2 Sum/prod of empties/nans

> **Warning:** This behavior is now standard as of v0.22.0 and is consistent with the default in `numpy`; previously sum/prod of all-NA or empty Series/DataFrames would return NaN. See *v0.22.0 whatsnew* for more.

The sum of an empty or all-NA Series or column of a DataFrame is 0.

```
In [36]: pd.Series([np.nan]).sum()
Out[36]: 0.0

In [37]: pd.Series([]).sum()
Out[37]: 0.0
```

The product of an empty or all-NA Series or column of a DataFrame is 1.

```
In [38]: pd.Series([np.nan]).prod()
Out[38]: 1.0

In [39]: pd.Series([]).prod()
Out[39]: 1.0
```

4.7.3 NA values in GroupBy

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

```
In [40]: df
Out[40]:
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.870484 -1.521966
e  0.219802  1.546457 -0.174262
f  0.831417  1.332054  0.438532
h       NaN -0.304618  0.956171

In [41]: df.groupby('one').mean()
Out[41]:
               two     three
one
0.219802  1.546457 -0.174262
0.831417  1.332054  0.438532
```

See the groupby section *here* for more information.

Cleaning / filling missing data

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

4.7.4 Filling missing values: fillna

`fillna()` can fill in NA values with non-NA data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [42]: df2
Out[42]:
        one       two     three four  five  timestamp
a       NaN -0.093641  0.014099  bar  True        NaT
c       NaN  0.870484 -1.521966  bar  True        NaT
e  0.219802  1.546457 -0.174262  bar  True 2012-01-01
f  0.831417  1.332054  0.438532  bar  True 2012-01-01
h       NaN -0.304618  0.956171  bar  True        NaT

In [43]: df2.fillna(0)
Out[43]:
        one       two     three four  five            timestamp
a  0.000000 -0.093641  0.014099  bar  True                    0
c  0.000000  0.870484 -1.521966  bar  True                    0
e  0.219802  1.546457 -0.174262  bar  True  2012-01-01 00:00:00
f  0.831417  1.332054  0.438532  bar  True  2012-01-01 00:00:00
h  0.000000 -0.304618  0.956171  bar  True                    0

In [44]: df2['one'].fillna('missing')
Out[44]:
a     missing
c     missing
e    0.219802
f    0.831417
h     missing
Name: one, dtype: object
```

Fill gaps forward or backward

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

```
In [45]: df
Out[45]:
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.870484 -1.521966
e  0.219802  1.546457 -0.174262
f  0.831417  1.332054  0.438532
h       NaN -0.304618  0.956171

In [46]: df.fillna(method='pad')
Out[46]:
        one       two     three
a       NaN -0.093641  0.014099
c       NaN  0.870484 -1.521966
e  0.219802  1.546457 -0.174262
f  0.831417  1.332054  0.438532
h  0.831417 -0.304618  0.956171
```

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 [47]: df
Out[47]:
   one       two     three
a  NaN -0.093641  0.014099
c  NaN  0.870484 -1.521966
e  NaN       NaN       NaN
f  NaN       NaN       NaN
h  NaN -0.304618  0.956171

In [48]: df.fillna(method='pad', limit=1)
Out[48]:
   one       two     three
a  NaN -0.093641  0.014099
c  NaN  0.870484 -1.521966
e  NaN  0.870484 -1.521966
f  NaN       NaN       NaN
h  NaN -0.304618  0.956171
```

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.

`ffill()` is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

4.7.5 Filling with a PandasObject

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```
In [49]: dff = pd.DataFrame(np.random.randn(10, 3), columns=list('ABC'))

In [50]: dff.iloc[3:5, 0] = np.nan

In [51]: dff.iloc[4:6, 1] = np.nan

In [52]: dff.iloc[5:8, 2] = np.nan

In [53]: dff
Out[53]:
          A         B         C
0  0.632107 -0.123200  0.579811
1 -0.617833  0.730289 -1.930809
2 -1.795903 -2.012672 -0.884042
3       NaN  0.809658  0.727889
4       NaN       NaN  1.683552
5 -1.134942       NaN       NaN
6 -1.654372 -0.175245       NaN
7  0.332654 -1.208013       NaN
```

```
8  0.028692  0.139178  0.877023
9  0.780292  0.682156  0.993475

In [54]: dff.fillna(dff.mean())
Out[54]: 
          A         B         C
0  0.632107 -0.123200  0.579811
1 -0.617833  0.730289 -1.930809
2 -1.795903 -2.012672 -0.884042
3 -0.428663  0.809658  0.727889
4 -0.428663 -0.144731  1.683552
5 -1.134942 -0.144731  0.292414
6 -1.654372 -0.175245  0.292414
7  0.332654 -1.208013  0.292414
8  0.028692  0.139178  0.877023
9  0.780292  0.682156  0.993475

In [55]: dff.fillna(dff.mean()['B':'C'])
Out[55]: 
          A         B         C
0  0.632107 -0.123200  0.579811
1 -0.617833  0.730289 -1.930809
2 -1.795903 -2.012672 -0.884042
3       NaN  0.809658  0.727889
4       NaN -0.144731  1.683552
5 -1.134942 -0.144731  0.292414
6 -1.654372 -0.175245  0.292414
7  0.332654 -1.208013  0.292414
8  0.028692  0.139178  0.877023
9  0.780292  0.682156  0.993475
```

Same result as above, but is aligning the fill value which is a Series in this case.

```
In [56]: dff.where(pd.notna(dff), dff.mean(), axis='columns')
Out[56]: 
          A         B         C
0  0.632107 -0.123200  0.579811
1 -0.617833  0.730289 -1.930809
2 -1.795903 -2.012672 -0.884042
3 -0.428663  0.809658  0.727889
4 -0.428663 -0.144731  1.683552
5 -1.134942 -0.144731  0.292414
6 -1.654372 -0.175245  0.292414
7  0.332654 -1.208013  0.292414
8  0.028692  0.139178  0.877023
9  0.780292  0.682156  0.993475
```

4.7.6 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 `dropna()`:

```
In [57]: df
Out[57]: 
   one       two     three
a  NaN -0.093641  0.014099
```

```
c  NaN  0.870484 -1.521966
e  NaN  0.000000  0.000000
f  NaN  0.000000  0.000000
h  NaN -0.304618  0.956171

In [58]: df.dropna(axis=0)
Out[58]:
Empty DataFrame
Columns: [one, two, three]
Index: []

In [59]: df.dropna(axis=1)
Out[59]:
        two     three
a -0.093641  0.014099
c  0.870484 -1.521966
e  0.000000  0.000000
f  0.000000  0.000000
h -0.304618  0.956171

In [60]: df['one'].dropna()
Out[60]: Series([], Name: one, dtype: float64)
```

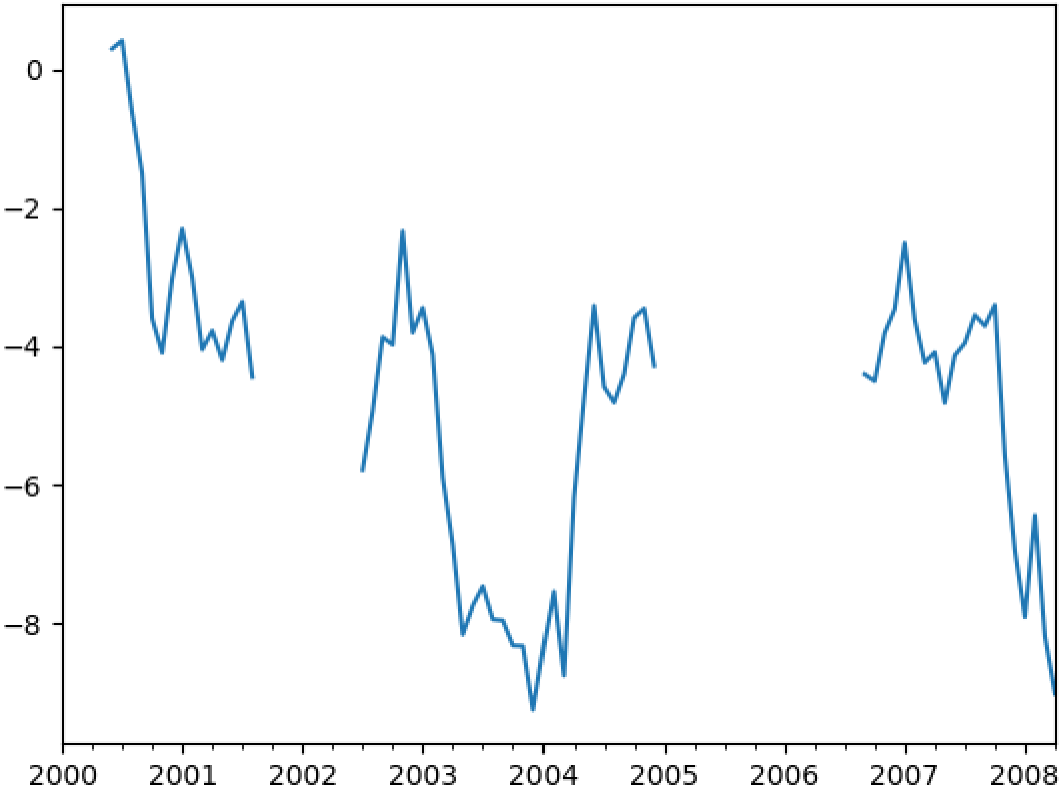
An equivalent `dropna()` is available for Series. DataFrame.dropna has considerably more options than Series.dropna, which can be examined *in the API*.

4.7.7 Interpolation

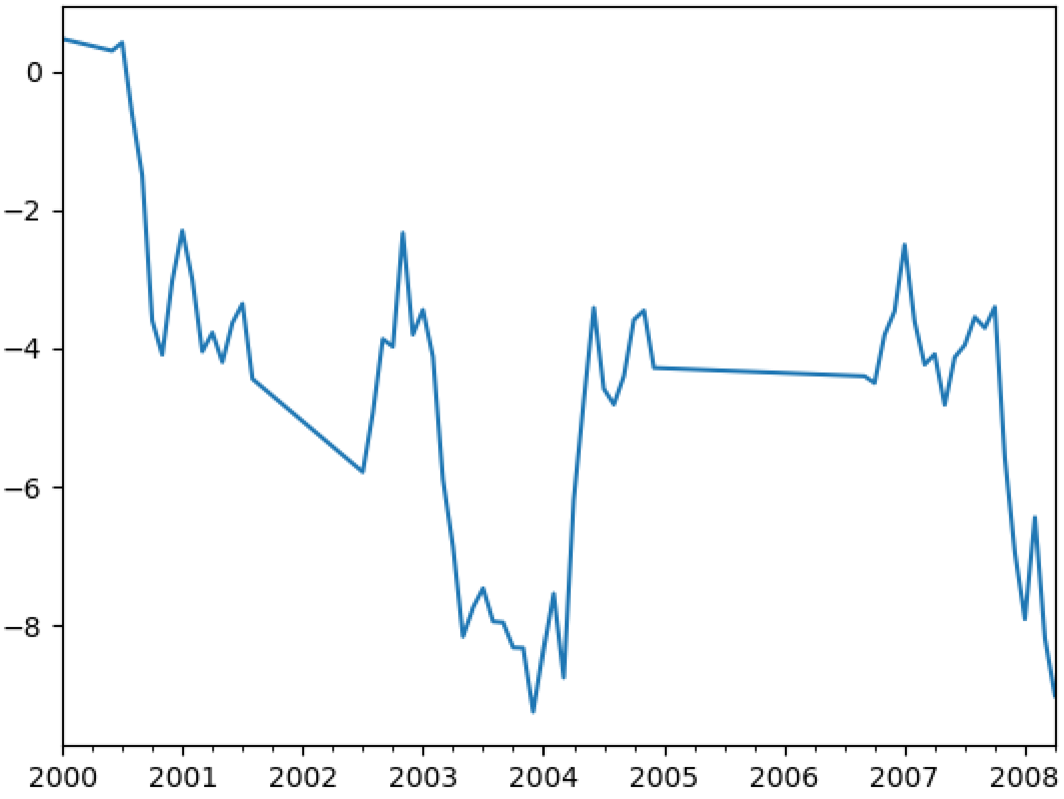
New in version 0.23.0: The `limit_area` keyword argument was added.

Both Series and DataFrame objects have `interpolate()` that, by default, performs linear interpolation at missing data points.

```
In [61]: ts
Out[61]:
2000-01-31    0.469112
2000-02-29         NaN
2000-03-31         NaN
2000-04-28         NaN
2000-05-31         NaN
                ...
2007-12-31   -6.950267
2008-01-31   -7.904475
2008-02-29   -6.441779
2008-03-31   -8.184940
2008-04-30   -9.011531
Freq: BM, Length: 100, dtype: float64

In [62]: ts.count()
Out[62]: 66

In [63]: ts.plot()
Out[63]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3f238410>
```



```
In [64]: ts.interpolate()
Out[64]:
2000-01-31    0.469112
2000-02-29    0.434469
2000-03-31    0.399826
2000-04-28    0.365184
2000-05-31    0.330541
                ...
2007-12-31   -6.950267
2008-01-31   -7.904475
2008-02-29   -6.441779
2008-03-31   -8.184940
2008-04-30   -9.011531
Freq: BM, Length: 100, dtype: float64

In [65]: ts.interpolate().count()
Out[65]: 100

In [66]: ts.interpolate().plot()
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3b392d90>
```



Index aware interpolation is available via the `method` keyword:

```
In [67]: ts2
Out[67]:
2000-01-31    0.469112
2000-02-29         NaN
2002-07-31   -5.785037
2005-01-31         NaN
2008-04-30   -9.011531
dtype: float64

In [68]: ts2.interpolate()
Out[68]:
2000-01-31    0.469112
2000-02-29   -2.657962
2002-07-31   -5.785037
2005-01-31   -7.398284
2008-04-30   -9.011531
dtype: float64

In [69]: ts2.interpolate(method='time')
Out[69]:
2000-01-31    0.469112
2000-02-29    0.270241
```

```
2002-07-31   -5.785037
2005-01-31   -7.190866
2008-04-30   -9.011531
dtype: float64
```

For a floating-point index, use `method='values'`:

```
In [70]: ser
Out[70]:
0.0      0.0
1.0      NaN
10.0    10.0
dtype: float64

In [71]: ser.interpolate()
Out[71]:
0.0      0.0
1.0      5.0
10.0    10.0
dtype: float64

In [72]: ser.interpolate(method='values')
Out[72]:
0.0      0.0
1.0      1.0
10.0    10.0
dtype: float64
```

You can also interpolate with a DataFrame:

```
In [73]: df = pd.DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
   ....:                    'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
   ....:

In [74]: df
Out[74]:
     A      B
0  1.0   0.25
1  2.1    NaN
2  NaN    NaN
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40

In [75]: df.interpolate()
Out[75]:
     A      B
0  1.0   0.25
1  2.1   1.50
2  3.4   2.75
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
```

The `method` argument gives access to fancier interpolation methods. If you have scipy installed, you can pass the name of a 1-d interpolation routine to `method`. Youll want to consult the full scipy interpolation documentation and reference guide for details. The appropriate interpolation method will depend on the type of data you are working with.

- If you are dealing with a time series that is growing at an increasing rate, `method='quadratic'` may be appropriate.
- If you have values approximating a cumulative distribution function, then `method='pchip'` should work well.
- To fill missing values with goal of smooth plotting, consider `method='akima'`.

> **Warning:** These methods require `scipy`.

```
In [76]: df.interpolate(method='barycentric')
Out[76]: 
      A       B
0  1.00   0.250
1  2.10  -7.660
2  3.53  -4.515
3  4.70   4.000
4  5.60  12.200
5  6.80  14.400

In [77]: df.interpolate(method='pchip')
Out[77]: 
         A          B
0  1.00000   0.250000
1  2.10000   0.672808
2  3.43454   1.928950
3  4.70000   4.000000
4  5.60000  12.200000
5  6.80000  14.400000

In [78]: df.interpolate(method='akima')
Out[78]: 
          A          B
0  1.000000   0.250000
1  2.100000  -0.873316
2  3.406667   0.320034
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

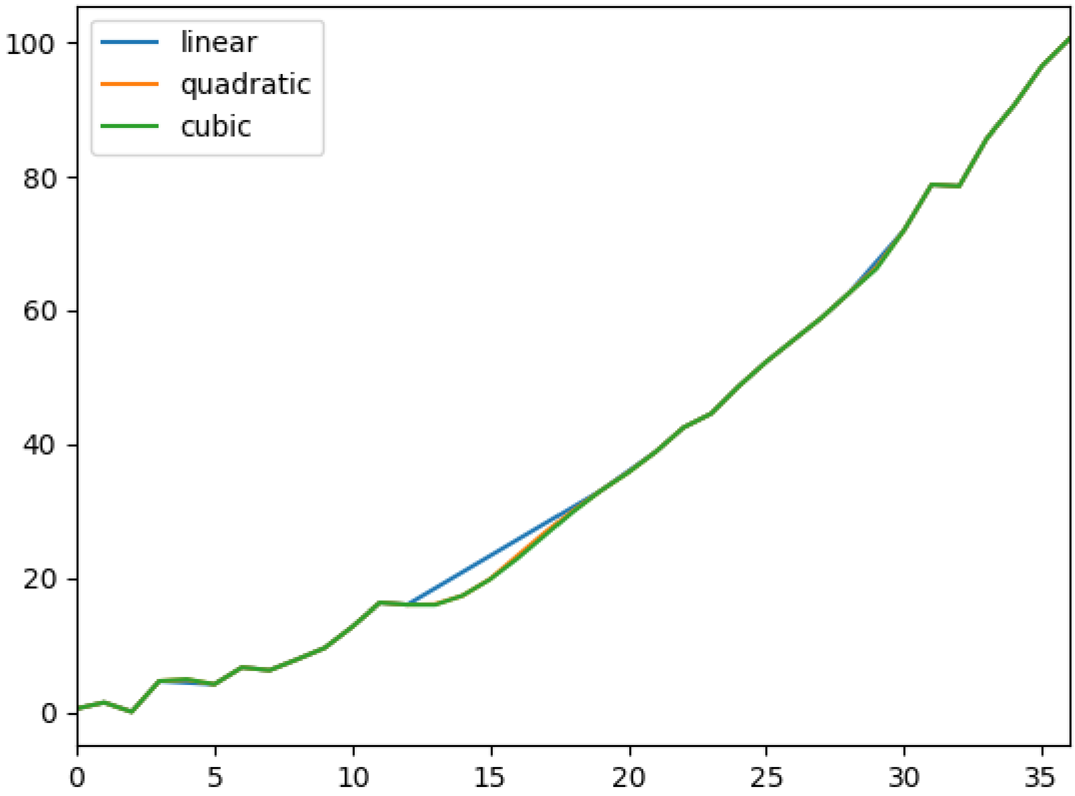
When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [79]: df.interpolate(method='spline', order=2)
Out[79]: 
          A          B
0  1.000000   0.250000
1  2.100000  -0.428598
2  3.404545   1.206900
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

```
In [80]: df.interpolate(method='polynomial', order=2)
Out[80]: 
          A          B
0  1.000000   0.250000
1  2.100000  -2.703846
2  3.451351  -1.453846
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

Compare several methods:

```
In [81]: np.random.seed(2)

In [82]: ser = pd.Series(np.arange(1, 10.1, .25) ** 2 + np.random.randn(37))

In [83]: missing = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])

In [84]: ser[missing] = np.nan

In [85]: methods = ['linear', 'quadratic', 'cubic']

In [86]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})

In [87]: df.plot()
Out[87]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3f319490>
```



Another use case is interpolation at *new* values. Suppose you have 100 observations from some distribution. And lets suppose that youre particularly interested in whats happening around the middle. You can mix pandas `reindex` and `interpolate` methods to interpolate at the new values.

```
In [88]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [89]: new_index = ser.index | pd.Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [90]: interp_s = ser.reindex(new_index).interpolate(method='pchip')

In [91]: interp_s[49:51]
Out[91]:
49.00    0.471410
49.25    0.476841
49.50    0.481780
49.75    0.485998
50.00    0.489266
50.25    0.491814
50.50    0.493995
50.75    0.495763
51.00    0.497074
dtype: float64
```

Interpolation limits

Like other pandas fill methods, `interpolate()` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive `NaN` values filled since the last valid observation:

```
In [92]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan,
   ....:                  np.nan, 13, np.nan, np.nan])
   ....:

In [93]: ser
Out[93]:
0     NaN
1     NaN
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
8     NaN
dtype: float64

# fill all consecutive values in a forward direction
In [94]: ser.interpolate()
Out[94]:
0     NaN
1     NaN
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
8    13.0
dtype: float64

# fill one consecutive value in a forward direction
In [95]: ser.interpolate(limit=1)
Out[95]:
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5     NaN
6    13.0
7    13.0
8     NaN
dtype: float64
```

By default, `NaN` values are filled in a `forward` direction. Use `limit_direction` parameter to fill `backward` or from `both` directions.

```
# fill one consecutive value backwards
In [96]: ser.interpolate(limit=1, limit_direction='backward')
```

```
Out[96]:
0     NaN
1     5.0
2     5.0
3     NaN
4     NaN
5    11.0
6    13.0
7     NaN
8     NaN
dtype: float64

# fill one consecutive value in both directions
In [97]: ser.interpolate(limit=1, limit_direction='both')
Out[97]:
0     NaN
1     5.0
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7    13.0
8     NaN
dtype: float64

# fill all consecutive values in both directions
In [98]: ser.interpolate(limit_direction='both')
Out[98]:
0     5.0
1     5.0
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
8    13.0
dtype: float64
```

By default, `NaN` values are filled whether they are inside (surrounded by) existing valid values, or outside existing valid values. Introduced in v0.23 the `limit_area` parameter restricts filling to either inside or outside values.

```
# fill one consecutive inside value in both directions
In [99]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
Out[99]:
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7     NaN
8     NaN
```

```
dtype: float64

# fill all consecutive outside values backward
In [100]: ser.interpolate(limit_direction='backward', limit_area='outside')
Out[100]:
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
8     NaN
dtype: float64

# fill all consecutive outside values in both directions
In [101]: ser.interpolate(limit_direction='both', limit_area='outside')
Out[101]:
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7    13.0
8    13.0
dtype: float64
```

4.7.8 Replacing generic values

Often times we want to replace arbitrary values with other values.

`replace()` in Series and `replace()` in DataFrame 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 [102]: ser = pd.Series([0., 1., 2., 3., 4.])

In [103]: ser.replace(0, 5)
Out[103]:
0    5.0
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [104]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[104]:
0    4.0
```

(continues on next page)

(continued from previous page)

```
1    3.0
2    2.0
3    1.0
4    0.0
dtype: float64
```

You can also specify a mapping dict:

```
In [105]: ser.replace({0: 10, 1: 100})
Out[105]:
0     10.0
1    100.0
2      2.0
3      3.0
4      4.0
dtype: float64
```

For a DataFrame, you can specify individual values by column:

```
In [106]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [107]: df.replace({'a': 0, 'b': 5}, 100)
Out[107]:
     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 [108]: ser.replace([1, 2, 3], method='pad')
Out[108]:
0    0.0
1    0.0
2    0.0
3    0.0
4    4.0
dtype: float64
```

4.7.9 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 [109]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}

In [110]: df = pd.DataFrame(d)
```

(continues on next page)

(continued from previous page)

```
In [111]: df.replace('.', np.nan)
Out[111]:
   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 [112]: df.replace(r'\s*\.\s*', np.nan, regex=True)
Out[112]:
   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 [113]: df.replace(['a', '.'], ['b', np.nan])
Out[113]:
   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 [114]: df.replace([r'\.', r'(a)'], ['dot', r'\1stuff'], regex=True)
Out[114]:
   a       b       c
0  0  astuff  astuff
1  1       b       b
2  2     dot     NaN
3  3     dot       d
```

Only search in column `'b'` (dict -> dict):

```
In [115]: df.replace({'b': '.'}, {'b': np.nan})
Out[115]:
   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 [116]: df.replace({'b': r'\s*\.\s*'}, {'b': np.nan}, regex=True)
Out[116]:
   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 [117]: df.replace({'b': {'b': r''}}, regex=True)
Out[117]:
   a  b    c
0  0  a    a
1  1       b
2  2  .  NaN
3  3  .    d
```

Alternatively, you can pass the nested dictionary like so:

```
In [118]: df.replace(regex={'b': {r'\s*\.\s*': np.nan}})
Out[118]:
   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 [119]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
Out[119]:
   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 [120]: df.replace([r'\s*\.\s*', r'a|b'], np.nan, regex=True)
Out[120]:
   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 explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be:

```
In [121]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=np.nan)
Out[121]:
   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.

4.7.10 Numeric replacement

`replace()` is similar to `fillna()`.

```
In [122]: df = pd.DataFrame(np.random.randn(10, 2))

In [123]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5

In [124]: df.replace(1.5, np.nan)
Out[124]: 
          0         1
0 -0.844214 -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4       NaN       NaN
5       NaN       NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9       NaN       NaN
```

Replacing more than one value is possible by passing a list.

```
In [125]: df00 = df.iloc[0, 0]

In [126]: df.replace([1.5, df00], [np.nan, 'a'])
Out[126]: 
          0         1
0         a  -1.02141
1  0.432396  -0.32358
2  0.423825   0.79918
3   1.26261  0.751965
4       NaN       NaN
5       NaN       NaN
6 -0.498174   -1.0608
7  0.591667 -0.183257
8   1.01985  -1.48247
9       NaN       NaN

In [127]: df[1].dtype
Out[127]: dtype('float64')
```

You can also operate on the DataFrame in place:

```
In [128]: df.replace(1.5, np.nan, inplace=True)
```

Warning: When replacing multiple `bool` or `datetime64` objects, the first argument to `replace` (`to_replace`) must match the type of the value being replaced. For example,

```
>>> s = pd.Series([True, False, True])
>>> s.replace({'a string': 'new value', True: False})  # raises
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

will raise a TypeError because one of the dict keys is not of the correct type for replacement.

However, when replacing a *single* object such as,

```
In [129]: s = pd.Series([True, False, True])

In [130]: s.replace('a string', 'another string')
Out[130]:
0     True
1    False
2     True
dtype: bool
```

the original NDFrame object will be returned untouched. Were working on unifying this API, but for backwards compatibility reasons we cannot break the latter behavior. See GH6354 for more details.

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, weve established some casting rules. When a reindexing operation introduces missing data, the Series will be cast according to the rules introduced in the table below.

data type	Cast to
integer	float
boolean	object
float	no cast
object	no cast

For example:

```
In [131]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [132]: s > 0
Out[132]:
0    True
2    True
4    True
6    True
7    True
dtype: bool

In [133]: (s > 0).dtype
Out[133]: dtype('bool')

In [134]: crit = (s > 0).reindex(list(range(8)))

In [135]: crit
Out[135]:
0    True
1     NaN
2    True
3     NaN
4    True
5     NaN
```

```
6    True
7    True
dtype: object

In [136]: crit.dtype
Out[136]: 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 [137]: reindexed = s.reindex(list(range(8))).fillna(0)

In [138]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-138-0dac417a4890> in <module>
----> 1 reindexed[crit]

~/sandbox/pandas-release/pandas/pandas/core/series.py in __getitem__(self, key)
   1108             key = list(key)
   1109 
-> 1110         if com.is_bool_indexer(key):
   1111             key = check_bool_indexer(self.index, key)
   1112 

~/sandbox/pandas-release/pandas/pandas/core/common.py in is_bool_indexer(key)
    128         if not lib.is_bool_array(key):
    129             if isna(key).any():
--> 130                 raise ValueError(na_msg)
    131             return False
    132         return True

ValueError: cannot index with vector containing NA / NaN values
```

However, these can be filled in using `fillna()` and it will work fine:

```
In [139]: reindexed[crit.fillna(False)]
Out[139]:
0    0.126504
2    0.696198
4    0.697416
6    0.601516
7    0.003659
dtype: float64

In [140]: reindexed[crit.fillna(True)]
Out[140]:
0    0.126504
1    0.000000
2    0.696198
3    0.000000
4    0.697416
5    0.000000
6    0.601516
7    0.003659
dtype: float64
```

Pandas provides a nullable integer dtype, but you must explicitly request it when creating the series or column. Notice that we use a capital I in the `dtype="Int64"`.

```
In [141]: s = pd.Series([0, 1, np.nan, 3, 4], dtype="Int64")

In [142]: s
Out[142]: 
0      0
1      1
2    NaN
3      3
4      4
dtype: Int64
```

See *Nullable integer data type* for more.   {{ header }}

4.8 Categorical data

This is an introduction to pandas categorical data type, including a short comparison with Rs `factor`.

Categoricals are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (*categories*; *levels* in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

In contrast to statistical categorical variables, categorical data might have an order (e.g. strongly agree vs agree or first observation vs. second observation), but numerical operations (additions, divisions,) are not possible.

All values of categorical data are either in *categories* or *np.nan*. Order is defined by the order of *categories*, not lexical order of the values. Internally, the data structure consists of a *categories* array and an integer array of *codes* which point to the real value in the *categories* array.

The categorical data type is useful in the following cases:

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory, see *here*.
- The lexical order of a variable is not the same as the logical order (one, two, three). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order, see *here*.
- As a signal to other Python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

See also the *API docs on categoricals*.

4.8.1 Object creation

Series creation

Categorical `Series` or columns in a `DataFrame` can be created in several ways:

By specifying `dtype="category"` when constructing a `Series`:

```
In [1]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [2]: s
```

(continues on next page)

(continued from previous page)

```
Out[2]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]
```

By converting an existing `Series` or column to a `category` dtype:

```
In [3]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [4]: df["B"] = df["A"].astype('category')

In [5]: df
Out[5]:
   A  B
0  a  a
1  b  b
2  c  c
3  a  a
```

By using special functions, such as `cut()`, which groups data into discrete bins. See the *example on tiling* in the docs.

```
In [6]: df = pd.DataFrame({'value': np.random.randint(0, 100, 20)})

In [7]: labels = ["{0} - {1}".format(i, i + 9) for i in range(0, 100, 10)]

In [8]: df['group'] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)

In [9]: df.head(10)
Out[9]:
   value    group
0     82  80 - 89
1      8    0 - 9
2     67  60 - 69
3      2    0 - 9
4     41  40 - 49
5     44  40 - 49
6     79  70 - 79
7     66  60 - 69
8      4    0 - 9
9     62  60 - 69
```

By passing a `pandas.Categorical` object to a `Series` or assigning it to a `DataFrame`.

```
In [10]: raw_cat = pd.Categorical(["a", "b", "c", "a"], categories=["b", "c", "d"],
   ....:                          ordered=False)
   ....:

In [11]: s = pd.Series(raw_cat)

In [12]: s
Out[12]:
0    NaN
```

(continues on next page)

(continued from previous page)

```
1      b
2      c
3    NaN
dtype: category
Categories (3, object): [b, c, d]

In [13]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [14]: df["B"] = raw_cat

In [15]: df
Out[15]:
   A    B
0  a  NaN
1  b    b
2  c    c
3  a  NaN
```

Categorical data has a specific `category` *dtype*:

```
In [16]: df.dtypes
Out[16]:
A      object
B    category
dtype: object
```

DataFrame creation

Similar to the previous section where a single column was converted to categorical, all columns in a `DataFrame` can be batch converted to categorical either during or after construction.

This can be done during construction by specifying `dtype="category"` in the `DataFrame` constructor:

```
In [17]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')}, dtype="category")

In [18]: df.dtypes
Out[18]:
A    category
B    category
dtype: object
```

Note that the categories present in each column differ; the conversion is done column by column, so only labels present in a given column are categories:

```
In [19]: df['A']
Out[19]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [20]: df['B']
Out[20]:
```

```
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

New in version 0.23.0.

Analogously, all columns in an existing `DataFrame` can be batch converted using `DataFrame.astype()`:

```
In [21]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [22]: df_cat = df.astype('category')

In [23]: df_cat.dtypes
Out[23]:
A    category
B    category
dtype: object
```

This conversion is likewise done column by column:

```
In [24]: df_cat['A']
Out[24]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [25]: df_cat['B']
Out[25]:
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

Controlling behavior

In the examples above where we passed `dtype='category'`, we used the default behavior:

1. Categories are inferred from the data.
2. Categories are unordered.

To control those behaviors, instead of passing `'category'`, use an instance of `CategoricalDtype`.

```
In [26]: from pandas.api.types import CategoricalDtype

In [27]: s = pd.Series(["a", "b", "c", "a"])

In [28]: cat_type = CategoricalDtype(categories=["b", "c", "d"],
   ....:                             ordered=True)
```

(continues on next page)

(continued from previous page)

```
   ....:

In [29]: s_cat = s.astype(cat_type)

In [30]: s_cat
Out[30]:
0    NaN
1      b
2      c
3    NaN
dtype: category
Categories (3, object): [b < c < d]
```

Similarly, a `CategoricalDtype` can be used with a `DataFrame` to ensure that categories are consistent among all columns.

```
In [31]: from pandas.api.types import CategoricalDtype

In [32]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [33]: cat_type = CategoricalDtype(categories=list('abcd'),
   ....:                             ordered=True)
   ....:

In [34]: df_cat = df.astype(cat_type)

In [35]: df_cat['A']
Out[35]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (4, object): [a < b < c < d]

In [36]: df_cat['B']
Out[36]:
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (4, object): [a < b < c < d]
```

Note: To perform table-wise conversion, where all labels in the entire `DataFrame` are used as categories for each column, the `categories` parameter can be determined programmatically by `categories = pd.unique(df.to_numpy().ravel())`.

If you already have `codes` and `categories`, you can use the *`from_codes()`* constructor to save the factorize step during normal constructor mode:

```
In [37]: splitter = np.random.choice([0, 1], 5, p=[0.5, 0.5])
```

(continues on next page)

(continued from previous page)

```
In [38]: s = pd.Series(pd.Categorical.from_codes(splitter,
   ....:                                          categories=["train", "test"]))
   ....:
```

Regaining original data

To get back to the original `Series` or NumPy array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:

```
In [39]: s = pd.Series(["a", "b", "c", "a"])

In [40]: s
Out[40]: 
0    a
1    b
2    c
3    a
dtype: object

In [41]: s2 = s.astype('category')

In [42]: s2
Out[42]: 
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [43]: s2.astype(str)
Out[43]: 
0    a
1    b
2    c
3    a
dtype: object

In [44]: np.asarray(s2)
Out[44]: array(['a', 'b', 'c', 'a'], dtype=object)
```

Note: In contrast to Rs *factor* function, categorical data is not converting input values to strings; categories will end up the same data type as the original values.

Note: In contrast to Rs *factor* function, there is currently no way to assign/change labels at creation time. Use *categories* to change the categories after creation time.

4.8.2 CategoricalDtype

Changed in version 0.21.0.

A categoricals type is fully described by

1. `categories`: a sequence of unique values and no missing values
2. `ordered`: a boolean

This information can be stored in a `CategoricalDtype`. The `categories` argument is optional, which implies that the actual categories should be inferred from whatever is present in the data when the *`pandas.Categorical`* is created. The categories are assumed to be unordered by default.

```
In [45]: from pandas.api.types import CategoricalDtype

In [46]: CategoricalDtype(['a', 'b', 'c'])
Out[46]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=None)

In [47]: CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[47]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [48]: CategoricalDtype()
Out[48]: CategoricalDtype(categories=None, ordered=None)
```

A `CategoricalDtype` can be used in any place pandas expects a *dtype*. For example *`pandas.read_csv()`*, *`pandas.DataFrame.astype()`*, or in the `Series` constructor.

Note: As a convenience, you can use the string `'category'` in place of a `CategoricalDtype` when you want the default behavior of the categories being unordered, and equal to the set values present in the array. In other words, `dtype='category'` is equivalent to `dtype=CategoricalDtype()`.

Equality semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and order. When comparing two unordered categoricals, the order of the `categories` is not considered.

```
In [49]: c1 = CategoricalDtype(['a', 'b', 'c'], ordered=False)

# Equal, since order is not considered when ordered=False
In [50]: c1 == CategoricalDtype(['b', 'c', 'a'], ordered=False)
Out[50]: True

# Unequal, since the second CategoricalDtype is ordered
In [51]: c1 == CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[51]: False
```

All instances of `CategoricalDtype` compare equal to the string `'category'`.

```
In [52]: c1 == 'category'
Out[52]: True
```

Warning: Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances `CategoricalDtype` compare equal to `'category'`, all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of `categories` or `ordered`.

4.8.3 Description

Using `describe()` on categorical data will produce similar output to a `Series` or `DataFrame` of type `string`.

```
In [53]: cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a",␣
↪"c"])

In [54]: df = pd.DataFrame({"cat": cat, "s": ["a", "c", "c", np.nan]})

In [55]: df.describe()
Out[55]: 
       cat  s
count    3  3
unique   2  2
top      c  c
freq     2  2

In [56]: df["cat"].describe()
Out[56]: 
count     3
unique    2
top       c
freq      2
Name: cat, dtype: object
```

4.8.4 Working with categories

Categorical data has a *categories* and a *ordered* property, which list their possible values and whether the ordering matters or not. These properties are exposed as `s.cat.categories` and `s.cat.ordered`. If you dont manually specify categories and ordering, they are inferred from the passed arguments.

```
In [57]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [58]: s.cat.categories
Out[58]: Index(['a', 'b', 'c'], dtype='object')

In [59]: s.cat.ordered
Out[59]: False
```

Its also possible to pass in the categories in a specific order:

```
In [60]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"],
   ....:                  categories=["c", "b", "a"]))
   ....: 

In [61]: s.cat.categories
Out[61]: Index(['c', 'b', 'a'], dtype='object')

In [62]: s.cat.ordered
```

```
Out[62]: False
```

Note: New categorical data are **not** automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered `Categorical`.

Note: The result of `unique()` is not always the same as `Series.cat.categories`, because `Series.unique()` has a couple of guarantees, namely that it returns categories in the order of appearance, and it only includes values that are actually present.

```
In [63]: s = pd.Series(list('babc')).astype(CategoricalDtype(list('abcd')))

In [64]: s
Out[64]:
0    b
1    a
2    b
3    c
dtype: category
Categories (4, object): [a, b, c, d]

# categories
In [65]: s.cat.categories
Out[65]: Index(['a', 'b', 'c', 'd'], dtype='object')

# uniques
In [66]: s.unique()
Out[66]:
[b, a, c]
Categories (3, object): [b, a, c]
```

Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `rename_categories()` method:

```
In [67]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [68]: s
Out[68]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [69]: s.cat.categories = ["Group %s" % g for g in s.cat.categories]

In [70]: s
Out[70]:
0    Group a
1    Group b
```

(continues on next page)

(continued from previous page)

```
2    Group c
3    Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]

In [71]: s = s.cat.rename_categories([1, 2, 3])

In [72]: s
Out[72]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [1, 2, 3]

# You can also pass a dict-like object to map the renaming
In [73]: s = s.cat.rename_categories({1: 'x', 2: 'y', 3: 'z'})

In [74]: s
Out[74]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Note: In contrast to Rs *factor*, categorical data can have categories of other types than string.

Note: Be aware that assigning new categories is an inplace operation, while most other operations under `Series.cat` per default return a new `Series` of dtype *category*.

Categories must be unique or a *ValueError* is raised:

```
In [75]: try:
   ....:     s.cat.categories = [1, 1, 1]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorical categories must be unique
```

Categories must also not be `NaN` or a *ValueError* is raised:

```
In [76]: try:
   ....:     s.cat.categories = [1, 2, np.nan]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorial categories cannot be null
```

Appending new categories

Appending categories can be done by using the `add_categories()` method:

```
In [77]: s = s.cat.add_categories([4])

In [78]: s.cat.categories
Out[78]: Index(['x', 'y', 'z', 4], dtype='object')

In [79]: s
Out[79]:
0    x
1    y
2    z
3    x
dtype: category
Categories (4, object): [x, y, z, 4]
```

Removing categories

Removing categories can be done by using the `remove_categories()` method. Values which are removed are replaced by `np.nan`.:

```
In [80]: s = s.cat.remove_categories([4])

In [81]: s
Out[81]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Removing unused categories

Removing unused categories can also be done:

```
In [82]: s = pd.Series(pd.Categorical(["a", "b", "a"],
   ....:               categories=["a", "b", "c", "d"]))
   ....:

In [83]: s
Out[83]:
0    a
1    b
2    a
dtype: category
Categories (4, object): [a, b, c, d]

In [84]: s.cat.remove_unused_categories()
Out[84]:
0    a
1    b
```

```
2    a
dtype: category
Categories (2, object): [a, b]
```

Setting categories

If you want to do remove and add new categories in one step (which has some speed advantage), or simply set the categories to a predefined scale, use `set_categories()`.

```
In [85]: s = pd.Series(["one", "two", "four", "-"], dtype="category")

In [86]: s
Out[86]: 
0     one
1     two
2    four
3       -
dtype: category
Categories (4, object): [-, four, one, two]

In [87]: s = s.cat.set_categories(["one", "two", "three", "four"])

In [88]: s
Out[88]: 
0     one
1     two
2    four
3     NaN
dtype: category
Categories (4, object): [one, two, three, four]
```

Note: Be aware that `Categorical.set_categories()` cannot know whether some category is omitted intentionally or because it is misspelled or (under Python3) due to a type difference (e.g., NumPy S1 dtype and Python strings). This can result in surprising behaviour!

4.8.5 Sorting and order

If categorical data is ordered (`s.cat.ordered == True`), then the order of the categories has a meaning and certain operations are possible. If the categorical is unordered, `.min()/.max()` will raise a `TypeError`.

```
In [89]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], ordered=False))

In [90]: s.sort_values(inplace=True)

In [91]: s = pd.Series(["a", "b", "c", "a"]).astype(
   ....:     CategoricalDtype(ordered=True)
   ....: )
   ....: 

In [92]: s.sort_values(inplace=True)

In [93]: s
Out[93]: 
```

```
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [94]: s.min(), s.max()
Out[94]: ('a', 'c')
```

You can set categorical data to be ordered by using `as_ordered()` or unordered by using `as_unordered()`. These will by default return a *new* object.

```
In [95]: s.cat.as_ordered()
Out[95]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [96]: s.cat.as_unordered()
Out[96]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a, b, c]
```

Sorting will use the order defined by categories, not any lexical order present on the data type. This is even true for strings and numeric data:

```
In [97]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [98]: s = s.cat.set_categories([2, 3, 1], ordered=True)

In [99]: s
Out[99]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [100]: s.sort_values(inplace=True)

In [101]: s
Out[101]:
1    2
2    3
0    1
3    1
dtype: category
```

```
Categories (3, int64): [2 < 3 < 1]

In [102]: s.min(), s.max()
Out[102]: (2, 1)
```

Reordering

Reordering the categories is possible via the `Categorical.reorder_categories()` and the `Categorical.set_categories()` methods. For `Categorical.reorder_categories()`, all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [103]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [104]: s = s.cat.reorder_categories([2, 3, 1], ordered=True)

In [105]: s
Out[105]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [106]: s.sort_values(inplace=True)

In [107]: s
Out[107]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [108]: s.min(), s.max()
Out[108]: (2, 1)
```

Note: Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the `Series`, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the `Series` are changed.

Note: If the `Categorical` is not ordered, `Series.min()` and `Series.max()` will raise `TypeError`. Numeric operations like `+`, `-`, `*`, `/` and operations based on them (e.g. `Series.median()`, which would need to compute the mean between two values if the length of an array is even) do not work and raise a `TypeError`.

Multi column sorting

A categorical dtyped column will participate in a multi-column sort in a similar manner to other columns. The ordering of the categorical is determined by the `categories` of that column.

```
In [109]: dfs = pd.DataFrame({'A': pd.Categorical(list('bbeebbaa'),
   .....:                                         categories=['e', 'a', 'b'],
   .....:                                         ordered=True),
   .....:                     'B': [1, 2, 1, 2, 2, 1, 2, 1]})
   .....:

In [110]: dfs.sort_values(by=['A', 'B'])
Out[110]:
   A  B
2  e  1
3  e  2
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
```

Reordering the `categories` changes a future sort.

```
In [111]: dfs['A'] = dfs['A'].cat.reorder_categories(['a', 'b', 'e'])

In [112]: dfs.sort_values(by=['A', 'B'])
Out[112]:
   A  B
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
2  e  1
3  e  2
```

4.8.6 Comparisons

Comparing categorical data with other objects is possible in three cases:

- Comparing equality (== and `!=`) to a list-like object (list, Series, array,) of the same length as the categorical data.
- All comparisons (==, `!=`, >, >=, <, and <=) of categorical data to another categorical Series, when `ordered==True` and the *categories* are the same.
- All comparisons of a categorical data to a scalar.

All other comparisons, especially non-equality comparisons of two categoricals with different categories or a categorical with any list-like object, will raise a `TypeError`.

Note: Any non-equality comparisons of categorical data with a `Series`, `np.array`, `list` or categorical data with different categories or ordering will raise a `TypeError` because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [113]: cat = pd.Series([1, 2, 3]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
   .....:

In [114]: cat_base = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
   .....:

In [115]: cat_base2 = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype(ordered=True)
   .....: )
   .....:

In [116]: cat
Out[116]:
0    1
1    2
2    3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [117]: cat_base
Out[117]:
0    2
1    2
2    2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [118]: cat_base2
Out[118]:
0    2
1    2
2    2
dtype: category
Categories (1, int64): [2]
```

Comparing to a categorical with the same categories and ordering or to a scalar works:

```
In [119]: cat > cat_base
Out[119]:
0     True
1    False
2    False
dtype: bool

In [120]: cat > 2
Out[120]:
0     True
1    False
```

```
2    False
dtype: bool
```

Equality comparisons work with any list-like object of same length and scalars:

```
In [121]: cat == cat_base
Out[121]:
0    False
1     True
2    False
dtype: bool

In [122]: cat == np.array([1, 2, 3])
Out[122]:
0    True
1    True
2    True
dtype: bool

In [123]: cat == 2
Out[123]:
0    False
1     True
2    False
dtype: bool
```

This doesnt work because the categories are not the same:

```
In [124]: try:
   .....:     cat > cat_base2
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categoricals can only be compared if 'categories' are the same. Categories␣
↪are different lengths
```

If you want to do a non-equality comparison of a categorical series with a list-like object which is not categorical data, you need to be explicit and convert the categorical data back to the original values:

```
In [125]: base = np.array([1, 2, 3])

In [126]: try:
   .....:     cat > base
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Cannot compare a Categorical for op __gt__ with type <class 'numpy.
↪ndarray'>.
If you want to compare values, use 'np.asarray(cat) <op> other'.

In [127]: np.asarray(cat) > base
Out[127]: array([False, False, False])
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```
In [128]: c1 = pd.Categorical(['a', 'b'], categories=['a', 'b'], ordered=False)
```

(continues on next page)

(continued from previous page)

```
In [129]: c2 = pd.Categorical(['a', 'b'], categories=['b', 'a'], ordered=False)

In [130]: c1 == c2
Out[130]: array([ True,  True])
```

4.8.7 Operations

Apart from `Series.min()`, `Series.max()` and `Series.mode()`, the following operations are possible with categorical data:

`Series` methods like `Series.value_counts()` will use all categories, even if some categories are not present in the data:

```
In [131]: s = pd.Series(pd.Categorical(["a", "b", "c", "c"],
   .....:                categories=["c", "a", "b", "d"]))
   .....:

In [132]: s.value_counts()
Out[132]:
c    2
b    1
a    1
d    0
dtype: int64
```

Groupby will also show unused categories:

```
In [133]: cats = pd.Categorical(["a", "b", "b", "b", "c", "c", "c"],
   .....:                        categories=["a", "b", "c", "d"])
   .....:

In [134]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 2, 3, 4, 5]})

In [135]: df.groupby("cats").mean()
Out[135]:
      values
cats
a        1.0
b        2.0
c        4.0
d        NaN

In [136]: cats2 = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [137]: df2 = pd.DataFrame({"cats": cats2,
   .....:                     "B": ["c", "d", "c", "d"],
   .....:                     "values": [1, 2, 3, 4]})
   .....:

In [138]: df2.groupby(["cats", "B"]).mean()
Out[138]:
        values
cats B
a    c     1.0
     d     2.0
```

(continues on next page)

(continued from previous page)

```
b    c      3.0
     d      4.0
c    c      NaN
     d      NaN
```

Pivot tables:

```
In [139]: raw_cat = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [140]: df = pd.DataFrame({"A": raw_cat,
   .....:                    "B": ["c", "d", "c", "d"],
   .....:                    "values": [1, 2, 3, 4]})
   .....:

In [141]: pd.pivot_table(df, values='values', index=['A', 'B'])
Out[141]:
     values
A B
a c       1
  d       2
b c       3
  d       4
```

4.8.8 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.at`, and `.iat`, work as normal. The only difference is the return type (for getting) and that only values already in *categories* can be assigned.

Getting

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.

```
In [142]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [143]: cats = pd.Series(["a", "b", "b", "b", "c", "c", "c"],
   .....:                  dtype="category", index=idx)
   .....:

In [144]: values = [1, 2, 2, 2, 3, 4, 5]

In [145]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [146]: df.iloc[2:4, :]
Out[146]:
  cats  values
j    b       2
k    b       2

In [147]: df.iloc[2:4, :].dtypes
Out[147]:
cats      category
values       int64
dtype: object
```

```
In [148]: df.loc["h":"j", "cats"]
Out[148]:
h    a
i    b
j    b
Name: cats, dtype: category
Categories (3, object): [a, b, c]

In [149]: df[df["cats"] == "b"]
Out[149]:
  cats  values
i    b       2
j    b       2
k    b       2
```

An example where the category type is not preserved is if you take one single row: the resulting `Series` is of dtype `object`:

```
# get the complete "h" row as a Series
In [150]: df.loc["h", :]
Out[150]:
cats      a
values    1
Name: h, dtype: object
```

Returning a single item from categorical data will also return the value, not a categorical of length 1.

```
In [151]: df.iat[0, 0]
Out[151]: 'a'

In [152]: df["cats"].cat.categories = ["x", "y", "z"]

In [153]: df.at["h", "cats"]  # returns a string
Out[153]: 'x'
```

Note: The is in contrast to Rs *factor* function, where `factor(c(1,2,3))[1]` returns a single value *factor*.

To get a single value `Series` of type `category`, you pass in a list with a single value:

```
In [154]: df.loc[["h"], "cats"]
Out[154]:
h    x
Name: cats, dtype: category
Categories (3, object): [x, y, z]
```

String and datetime accessors

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```
In [155]: str_s = pd.Series(list('aabb'))

In [156]: str_cat = str_s.astype('category')
In [157]: str_cat
Out[157]:
0    a
1    a
2    b
3    b
dtype: category
Categories (2, object): [a, b]

In [158]: str_cat.str.contains("a")
Out[158]:
0     True
1     True
2    False
3    False
dtype: bool

In [159]: date_s = pd.Series(pd.date_range('1/1/2015', periods=5))

In [160]: date_cat = date_s.astype('category')

In [161]: date_cat
Out[161]:
0   2015-01-01
1   2015-01-02
2   2015-01-03
3   2015-01-04
4   2015-01-05
dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-
↪04, 2015-01-05]

In [162]: date_cat.dt.day
Out[162]:
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Note: The returned `Series` (or `DataFrame`) is of the same type as if you used the `.str.<method>` / `.dt.<method>` on a `Series` of that type (and not of type `category`!).

That means, that the returned values from methods and properties on the accessors of a `Series` and the returned values from methods and properties on the accessors of this `Series` transformed to one of type *category* will be equal:

```
In [163]: ret_s = str_s.str.contains("a")

In [164]: ret_cat = str_cat.str.contains("a")

In [165]: ret_s.dtype == ret_cat.dtype
```

```
Out[165]: True

In [166]: ret_s == ret_cat
Out[166]: 
0    True
1    True
2    True
3    True
dtype: bool
```

Note: The work is done on the `categories` and then a new `Series` is constructed. This has some performance implication if you have a `Series` of type string, where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`). In this case it can be faster to convert the original `Series` to one of type `category` and use `.str.<method>` or `.dt.<property>` on that.

Setting

Setting values in a categorical column (or `Series`) works as long as the value is included in the *categories*:

```
In [167]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [168]: cats = pd.Categorical(["a", "a", "a", "a", "a", "a", "a"],
   .....:                       categories=["a", "b"])
   .....: 

In [169]: values = [1, 1, 1, 1, 1, 1, 1]

In [170]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [171]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

In [172]: df
Out[172]: 
  cats  values
h    a       1
i    a       1
j    b       2
k    b       2
l    a       1
m    a       1
n    a       1

In [173]: try:
   .....:     df.iloc[2:4, :] = [["c", 3], ["c", 3]]
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....: 
ValueError: Cannot setitem on a Categorical with a new category, set the␣
↪categories first
```

Setting values by assigning categorical data will also check that the *categories* match:

```
In [174]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"],␣
↪categories=["a", "b"])

In [175]: df
Out[175]:
  cats  values
h    a       1
i    a       1
j    a       2
k    a       2
l    a       1
m    a       1
n    a       1

In [176]: try:
   .....:     df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"],
   .....:                                              categories=["a", "b",␣
↪"c"])
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
ValueError: Cannot set a Categorical with another, without identical␣
↪categories
```

Assigning a `Categorical` to parts of a column of other types will use the values:

```
In [177]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a",␣
↪"a"]})

In [178]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [179]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [180]: df
Out[180]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a

In [181]: df.dtypes
Out[181]:
a    object
b    object
dtype: object
```

Merging

You can concat two `DataFrames` containing categorical data together, but the categories of these categoricals need to be the same:

```
In [182]: cat = pd.Series(["a", "b"], dtype="category")

In [183]: vals = [1, 2]
In [184]: df = pd.DataFrame({"cats": cat, "vals": vals})

In [185]: res = pd.concat([df, df])

In [186]: res
Out[186]:
  cats  vals
0    a     1
1    b     2
0    a     1
1    b     2

In [187]: res.dtypes
Out[187]:
cats    category
vals       int64
dtype: object
```

In this case the categories are not the same, and therefore an error is raised:

```
In [188]: df_different = df.copy()

In [189]: df_different["cats"].cat.categories = ["c", "d"]

In [190]: try:
   .....:     pd.concat([df, df_different])
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
```

The same applies to `df.append(df_different)`.

See also the section on *merge dtypes* for notes about preserving merge dtypes and performance.

Unioning

New in version 0.19.0.

If you want to combine categoricals that do not necessarily have the same categories, the *union_categoricals()* function will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
In [191]: from pandas.api.types import union_categoricals

In [192]: a = pd.Categorical(["b", "c"])

In [193]: b = pd.Categorical(["a", "b"])

In [194]: union_categoricals([a, b])
Out[194]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the data. If you want the categories to be lexsorted, use `sort_categories=True` argument.

```
In [195]: union_categoricals([a, b], sort_categories=True)
Out[195]: 
[b, c, a, b]
Categories (3, object): [a, b, c]
```

union_categoricals also works with the easy case of combining two categoricals of the same categories and order information (e.g. what you could also append for).

```
In [196]: a = pd.Categorical(["a", "b"], ordered=True)

In [197]: b = pd.Categorical(["a", "b", "a"], ordered=True)

In [198]: union_categoricals([a, b])
Out[198]: 
[a, b, a, b, a]
Categories (2, object): [a < b]
```

The below raises TypeError because the categories are ordered and not identical.

```
In [1]: a = pd.Categorical(["a", "b"], ordered=True)
In [2]: b = pd.Categorical(["a", "b", "c"], ordered=True)
In [3]: union_categoricals([a, b])
Out[3]:
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0.

Ordered categoricals with different categories or orderings can be combined by using the ignore_ordered=True argument.

```
In [199]: a = pd.Categorical(["a", "b", "c"], ordered=True)

In [200]: b = pd.Categorical(["c", "b", "a"], ordered=True)

In [201]: union_categoricals([a, b], ignore_order=True)
Out[201]: 
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

union_categoricals() also works with a CategoricalIndex, or Series containing categorical data, but note that the resulting array will always be a plain Categorical:

```
In [202]: a = pd.Series(["b", "c"], dtype='category')

In [203]: b = pd.Series(["a", "b"], dtype='category')

In [204]: union_categoricals([a, b])
Out[204]: 
[b, c, a, b]
Categories (3, object): [b, c, a]
```

Note: union_categoricals may recode the integer codes for categories when combining categoricals. This is likely what you want, but if you are relying on the exact numbering of the categories, be aware.

```
In [205]: c1 = pd.Categorical(["b", "c"])

In [206]: c2 = pd.Categorical(["a", "b"])
In [207]: c1
Out[207]:
[b, c]
Categories (2, object): [b, c]

# "b" is coded to 0
In [208]: c1.codes
Out[208]: array([0, 1], dtype=int8)

In [209]: c2
Out[209]:
[a, b]
Categories (2, object): [a, b]

# "b" is coded to 1
In [210]: c2.codes
Out[210]: array([0, 1], dtype=int8)

In [211]: c = union_categoricals([c1, c2])

In [212]: c
Out[212]:
[b, c, a, b]
Categories (3, object): [b, c, a]

# "b" is coded to 0 throughout, same as c1, different from c2
In [213]: c.codes
Out[213]: array([0, 1, 2, 0], dtype=int8)
```

Concatenation

This section describes concatenations specific to `category` dtype. See *Concatenating objects* for general description.

By default, `Series` or `DataFrame` concatenation which contains the same categories results in `category` dtype, otherwise results in `object` dtype. Use `.astype` or `union_categoricals` to get `category` result.

```
# same categories
In [214]: s1 = pd.Series(['a', 'b'], dtype='category')

In [215]: s2 = pd.Series(['a', 'b', 'a'], dtype='category')

In [216]: pd.concat([s1, s2])
Out[216]:
0    a
1    b
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]

# different categories
```

```
In [217]: s3 = pd.Series(['b', 'c'], dtype='category')

In [218]: pd.concat([s1, s3])
Out[218]:
0    a
1    b
0    b
1    c
dtype: object

In [219]: pd.concat([s1, s3]).astype('category')
Out[219]:
0    a
1    b
0    b
1    c
dtype: category
Categories (3, object): [a, b, c]

In [220]: union_categoricals([s1.array, s3.array])
Out[220]:
[a, b, b, c]
Categories (3, object): [a, b, c]
```

Following table summarizes the results of `Categoricals` related concatenations.

arg1	arg2	result
category	category (identical categories)	category
category	category (different categories, both not ordered)	object (dtype is inferred)
category	category (different categories, either one is ordered)	object (dtype is inferred)
category	not category	object (dtype is inferred)

4.8.9 Getting data in/out

You can write data that contains `category` dtypes to a `HDFStore`. See *here* for an example and caveats.

It is also possible to write data to and reading data from *Stata* format files. See *here* for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to *category* and assign the right categories and categories ordering.

```
In [221]: import io

In [222]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))

# rename the categories
In [223]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [224]: s = s.cat.set_categories(["very bad", "bad", "medium", "good",␣
 ↪"very good"])

In [225]: df = pd.DataFrame({"cats": s, "vals": [1, 2, 3, 4, 5, 6]})
In [226]: csv = io.StringIO()

In [227]: df.to_csv(csv)

In [228]: df2 = pd.read_csv(io.StringIO(csv.getvalue()))

In [229]: df2.dtypes
Out[229]:
Unnamed: 0     int64
cats          object
vals           int64
dtype: object

In [230]: df2["cats"]
Out[230]:
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: object

# Redo the category
In [231]: df2["cats"] = df2["cats"].astype("category")

In [232]: df2["cats"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                  "good", "very good"],
   .....:                                 inplace=True)
   .....:

In [233]: df2.dtypes
Out[233]:
Unnamed: 0       int64
cats          category
vals             int64
dtype: object

In [234]: df2["cats"]
Out[234]:
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

The same holds for writing to a SQL database with `to_sql`.

4.8.10 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*.

Missing values should **not** be included in the Categoricals `categories`, only in the `values`. Instead, it is understood that NaN is different, and is always a possibility. When working with the Categoricals `codes`, missing values will always have a code of `-1`.

```
In [235]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")

# only two categories
In [236]: s
Out[236]:
0      a
1      b
2    NaN
3      a
dtype: category
Categories (2, object): [a, b]

In [237]: s.cat.codes
Out[237]:
0    0
1    1
2   -1
3    0
dtype: int8
```

Methods for working with missing data, e.g. `isna()`, `fillna()`, `dropna()`, all work normally:

```
In [238]: s = pd.Series(["a", "b", np.nan], dtype="category")

In [239]: s
Out[239]:
0      a
1      b
2    NaN
dtype: category
Categories (2, object): [a, b]

In [240]: pd.isna(s)
Out[240]:
0    False
1    False
2     True
dtype: bool

In [241]: s.fillna("a")
Out[241]:
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]
```

4.8.11 Differences to Rs *factor*

The following differences to Rs factor functions can be observed:

- Rs *levels* are named *categories*.
- Rs *levels* are always of type string, while *categories* in pandas can be of any dtype.
- Its not possible to specify labels at creation time. Use `s.cat.rename_categories(new_labels)` afterwards.
- In contrast to Rs *factor* function, using categorical data as the sole input to create a new categorical series will *not* remove unused categories but create a new categorical series which is equal to the passed in one!
- R allows for missing values to be included in its *levels* (pandas *categories*). Pandas does not allow *NaN* categories, but missing values can still be in the *values*.

4.8.12 Gotchas

Memory usage

The memory usage of a `Categorical` is proportional to the number of categories plus the length of the data. In contrast, an `object` dtype is a constant times the length of the data.

```
In [242]: s = pd.Series(['foo', 'bar'] * 1000)

# object dtype
In [243]: s.nbytes
Out[243]: 16000

# category dtype
In [244]: s.astype('category').nbytes
Out[244]: 2016
```

Note: If the number of categories approaches the length of the data, the `Categorical` will use nearly the same or more memory than an equivalent `object` dtype representation.

```
In [245]: s = pd.Series(['foo%04d' % i for i in range(2000)])

# object dtype
In [246]: s.nbytes
Out[246]: 16000

# category dtype
In [247]: s.astype('category').nbytes
Out[247]: 20000
```

Categorical is not a *numpy* array

Currently, categorical data and the underlying `Categorical` is implemented as a Python object and not as a low-level NumPy array dtype. This leads to some problems.

NumPy itself doesnt know about the new *dtype*:

```
In [248]: try:
   .....:     np.dtype("category")
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type "category" not understood

In [249]: dtype = pd.Categorical(["a"]).dtype

In [250]: try:
   .....:     np.dtype(dtype)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type not understood
```

Dtype comparisons work:

```
In [251]: dtype == np.str_
Out[251]: False

In [252]: np.str_ == dtype
Out[252]: False
```

To check if a Series contains Categorical data, use `hasattr(s, 'cat')`:

```
In [253]: hasattr(pd.Series(['a'], dtype='category'), 'cat')
Out[253]: True

In [254]: hasattr(pd.Series(['a']), 'cat')
Out[254]: False
```

Using NumPy functions on a `Series` of type `category` should not work as *Categoricals* are not numeric data (even in the case that `.categories` is numeric).

```
In [255]: s = pd.Series(pd.Categorical([1, 2, 3, 4]))

In [256]: try:
   .....:     np.sum(s)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categorical cannot perform the operation sum
```

Note: If such a function works, please file a bug at https://github.com/pandas-dev/pandas!

dtype in apply

Pandas currently does not preserve the dtype in apply functions: If you apply along rows you get a *Series* of `object` *dtype* (same as getting a row -> getting one element will return a basic type) and applying along columns will also convert to object. `NaN` values are unaffected. You can use `fillna` to handle missing values before applying a function.

```
In [257]: df = pd.DataFrame({"a": [1, 2, 3, 4],
   .....:                    "b": ["a", "b", "c", "d"],
   .....:                     "cats": pd.Categorical([1, 2, 3, 2])})
   .....:

In [258]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[258]:
0    <class 'int'>
1    <class 'int'>
2    <class 'int'>
3    <class 'int'>
dtype: object

In [259]: df.apply(lambda col: col.dtype, axis=0)
Out[259]:
a          int64
b         object
cats    category
dtype: object
```

Categorical index

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements. See the *advanced indexing docs* for a more detailed explanation.

Setting the index will create a `CategoricalIndex`:

```
In [260]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])

In [261]: strings = ["a", "b", "c", "d"]

In [262]: values = [4, 2, 3, 1]

In [263]: df = pd.DataFrame({"strings": strings, "values": values},␣
↪index=cats)

In [264]: df.index
Out[264]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1],␣
↪ordered=False, dtype='category')

# This now sorts by the categories order
In [265]: df.sort_index()
Out[265]:
  strings  values
4       d       1
2       b       2
3       c       3
1       a       4
```

Side effects

Constructing a `Series` from a `Categorical` will not copy the input `Categorical`. This means that changes to the `Series` will in most cases change the original `Categorical`:

```
In [266]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [267]: s = pd.Series(cat, name="cat")

In [268]: cat
Out[268]: 
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [269]: s.iloc[0:2] = 10

In [270]: cat
Out[270]: 
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [271]: df = pd.DataFrame(s)

In [272]: df["cat"].cat.categories = [1, 2, 3, 4, 5]

In [273]: cat
Out[273]: 
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]
```

Use `copy=True` to prevent such a behaviour or simply dont reuse `Categoricals`:

```
In [274]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [275]: s = pd.Series(cat, name="cat", copy=True)

In [276]: cat
Out[276]: 
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [277]: s.iloc[0:2] = 10

In [278]: cat
Out[278]: 
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
```

Note: This also happens in some cases when you supply a NumPy array instead of a `Categorical`: using an int array (e.g. `np.array([1,2,3,4])`) will exhibit the same behavior, while using a string array (e.g. `np.array(["a","b","c","a"])`) will not.

{{ header }}

4.9 Nullable integer data type

New in version 0.24.0.

Note: IntegerArray is currently experimental. Its API or implementation may change without warning.

In Missing Data, we saw that pandas primarily uses `NaN` to represent missing data. Because `NaN` is a float, this forces an array of integers with any missing values to become floating point. In some cases, this may not matter much. But if your integer column is, say, an identifier, casting to float can be problematic. Some integers cannot even be represented as floating point numbers.

Pandas can represent integer data with possibly missing values using *`arrays.IntegerArray`*. This is an *extension types* implemented within pandas. It is not the default dtype for integers, and will not be inferred; you must explicitly pass the dtype into *`array()`* or *`Series`*:

```
In [1]: arr = pd.array([1, 2, np.nan], dtype=pd.Int64Dtype())

In [2]: arr
Out[2]:
<IntegerArray>
[1, 2, NaN]
Length: 3, dtype: Int64
```

Or the string alias `"Int64"` (note the capital `"I"`, to differentiate from NumPys `'int64'` dtype:

```
In [3]: pd.array([1, 2, np.nan], dtype="Int64")
Out[3]:
<IntegerArray>
[1, 2, NaN]
Length: 3, dtype: Int64
```

This array can be stored in a *`DataFrame`* or *`Series`* like any NumPy array.

```
In [4]: pd.Series(arr)
Out[4]:
0      1
1      2
2    NaN
dtype: Int64
```

You can also pass the list-like object to the *`Series`* constructor with the dtype.

```
In [5]: s = pd.Series([1, 2, np.nan], dtype="Int64")

In [6]: s
Out[6]:
0      1
1      2
2    NaN
dtype: Int64
```

By default (if you dont specify `dtype`), NumPy is used, and youll end up with a `float64` dtype Series:

```
In [7]: pd.Series([1, 2, np.nan])
Out[7]:
0    1.0
1    2.0
2    NaN
dtype: float64
```

Operations involving an integer array will behave similar to NumPy arrays. Missing values will be propagated, and and the data will be coerced to another dtype if needed.

```
# arithmetic
In [8]: s + 1
Out[8]:
0      2
1      3
2    NaN
dtype: Int64

# comparison
In [9]: s == 1
Out[9]:
0     True
1    False
2    False
dtype: bool

# indexing
In [10]: s.iloc[1:3]
Out[10]:
1      2
2    NaN
dtype: Int64

# operate with other dtypes
In [11]: s + s.iloc[1:3].astype('Int8')
Out[11]:
0    NaN
1      4
2    NaN
dtype: Int64

# coerce when needed
In [12]: s + 0.01
Out[12]:
0    1.01
1    2.01
2     NaN
dtype: float64
```

These dtypes can operate as part of of `DataFrame`.

```
In [13]: df = pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab')})

In [14]: df
Out[14]:
     A  B  C
0    1  1  a
1    2  1  a
2  NaN  3  b

In [15]: df.dtypes
Out[15]:
```

```
A      Int64
B      int64
C     object
dtype: object
```

These dtypes can be merged & reshaped & casted.

```
In [16]: pd.concat([df[['A']], df[['B', 'C']]], axis=1).dtypes
Out[16]:
A      Int64
B      int64
C     object
dtype: object

In [17]: df['A'].astype(float)
Out[17]:
0    1.0
1    2.0
2    NaN
Name: A, dtype: float64
```

Reduction and groupby operations such as sum work as well.

```
In [18]: df.sum()
Out[18]:
A      3
B      5
C    aab
dtype: object

In [19]: df.groupby('B').A.sum()
Out[19]:
B
1    3
3    0
Name: A, dtype: Int64
```

{{ header }}

4.10 Visualization

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt

In [2]: plt.close('all')
```

We provide the basics in pandas to easily create decent looking plots. See the *ecosystem* section for visualization libraries that go beyond the basics documented here.

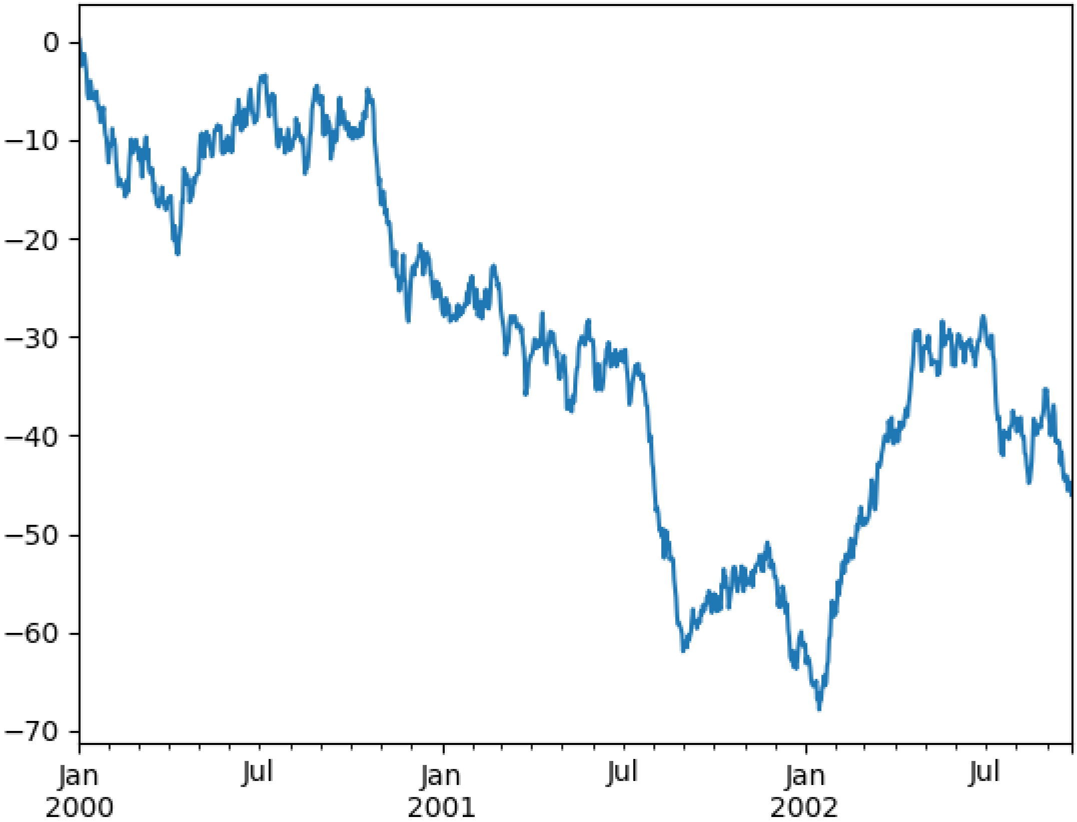
Note: All calls to `np.random` are seeded with 123456.

4.10.1 Basic plotting: `plot`

We will demonstrate the basics, see the *cookbook* for some advanced strategies.

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`:

```
In [3]: ts = pd.Series(np.random.randn(1000),
   ...:                index=pd.date_range('1/1/2000', periods=1000))
   ...:

In [4]: ts = ts.cumsum()

In [5]: ts.plot()
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1c36223c90>
```



If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above.

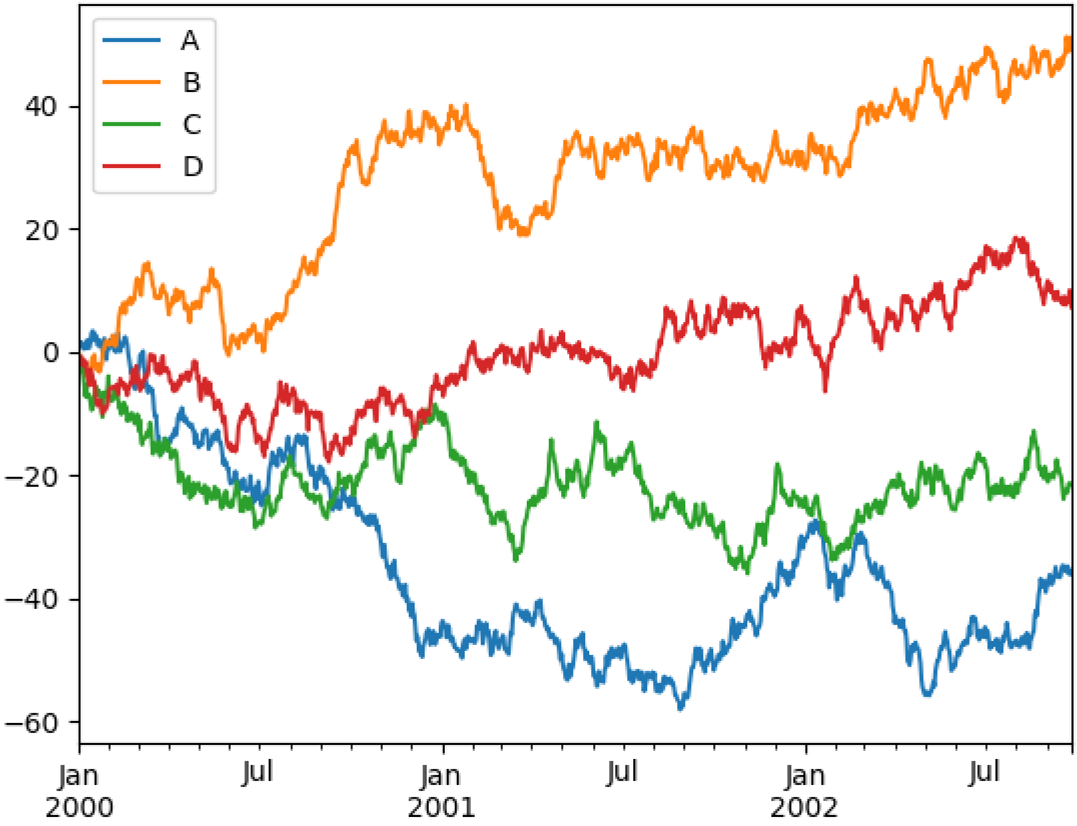
On DataFrame, `plot()` is a convenience to plot all of the columns with labels:

```
In [6]: df = pd.DataFrame(np.random.randn(1000, 4),
   ...:                   index=ts.index, columns=list('ABCD'))
   ...:

In [7]: df = df.cumsum()
```

(continues on next page)

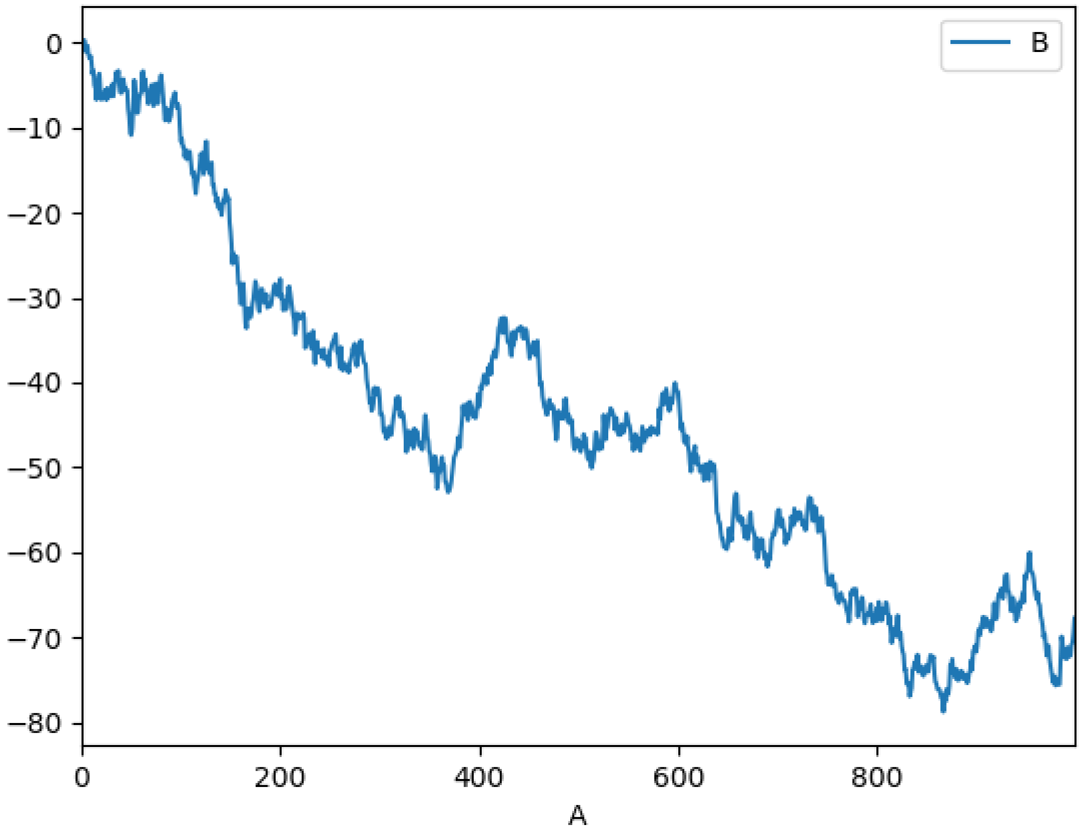
(continued from previous page)

```
In [8]: plt.figure();

In [9]: df.plot();
```



You can plot one column versus another using the *x* and *y* keywords in `plot()`:

```
In [10]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()

In [11]: df3['A'] = pd.Series(list(range(len(df))))

In [12]: df3.plot(x='A', y='B')
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3a20a490>
```



Note: For more formatting and styling options, see *formatting* below.

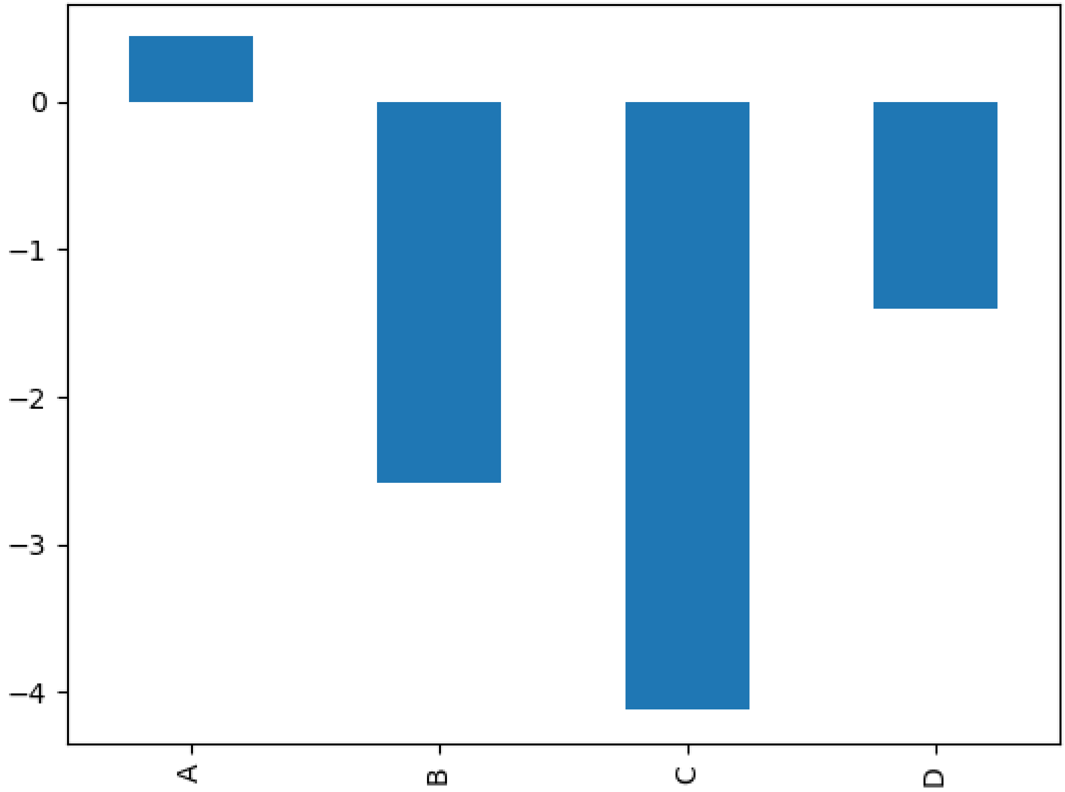
4.10.2 Other plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to `plot()`, and include:

- *bar* or *barh* for bar plots
- *hist* for histogram
- *box* for boxplot
- *kde* or *density* for density plots
- *area* for area plots
- *scatter* for scatter plots
- *hexbin* for hexagonal bin plots
- *pie* for pie plots

For example, a bar plot can be created the following way:

```
In [13]: plt.figure();

In [14]: df.iloc[5].plot(kind='bar');
```



You can also create these other plots using the methods `DataFrame.plot.<kind>` instead of providing the `kind` keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

```
In [15]: df = pd.DataFrame()

In [16]: df.plot.<TAB>  # noqa: E225, E999
df.plot.area     df.plot.barh     df.plot.density  df.plot.hist     df.plot.line     ␣
↪df.plot.scatter
df.plot.bar      df.plot.box      df.plot.hexbin   df.plot.kde      df.plot.pie
```

In addition to these `kind` s, there are the *DataFrame.hist()*, and *DataFrame.boxplot()* methods, which use a separate interface.

Finally, there are several *plotting functions* in `pandas.plotting` that take a `Series` or `DataFrame` as an argument. These include:

- *Scatter Matrix*
- *Andrews Curves*
- *Parallel Coordinates*

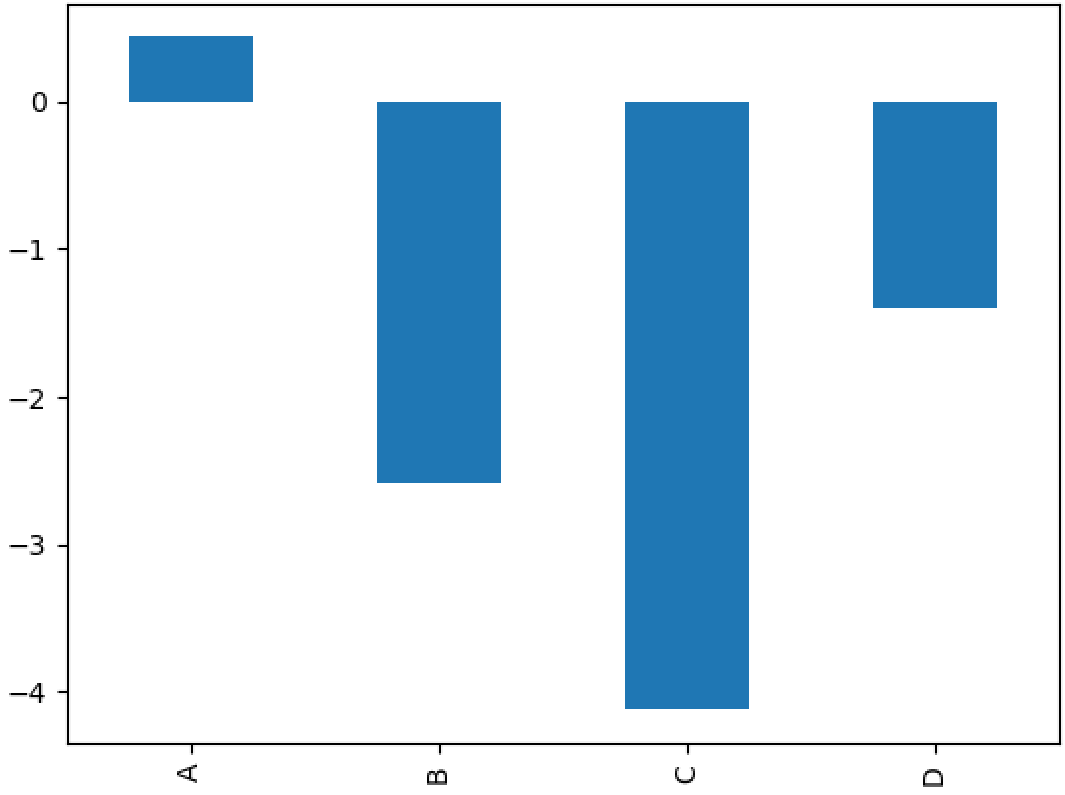
- *Lag Plot*
- *Autocorrelation Plot*
- *Bootstrap Plot*
- *RadViz*

Plots may also be adorned with *errorbars* or *tables*.

Bar plots

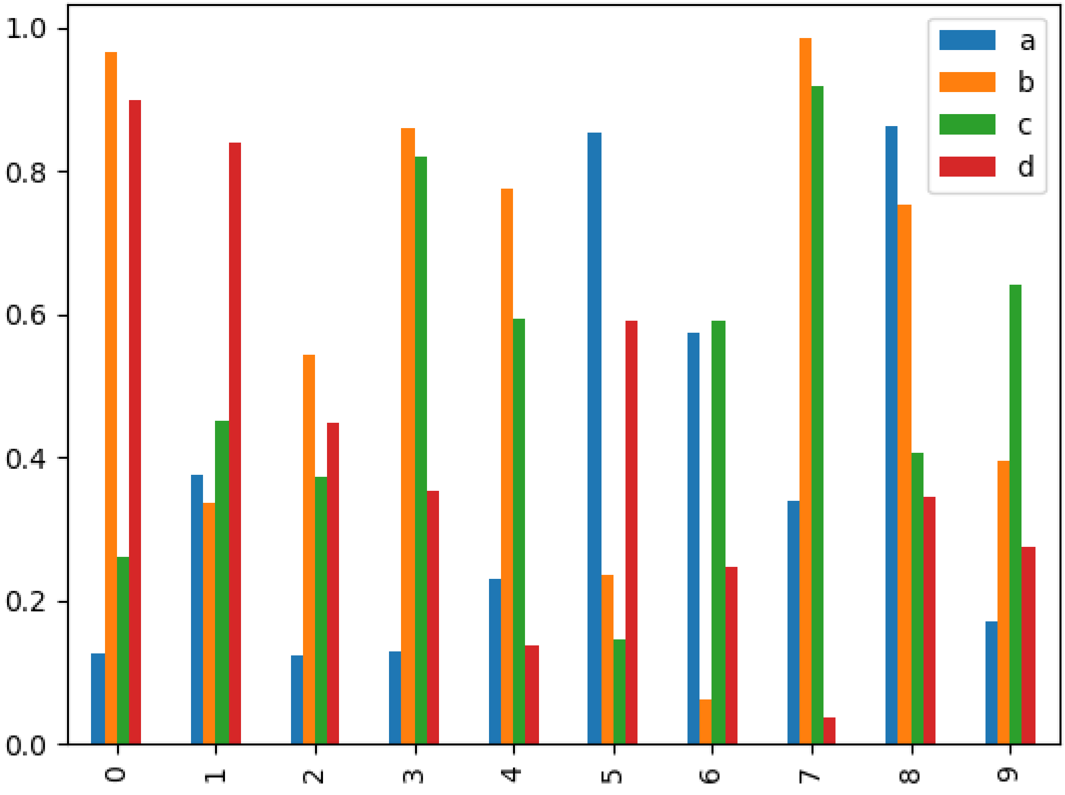
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [17]: plt.figure();

In [18]: df.iloc[5].plot.bar()
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3aa7aa50>

In [19]: plt.axhline(0, color='k');
```



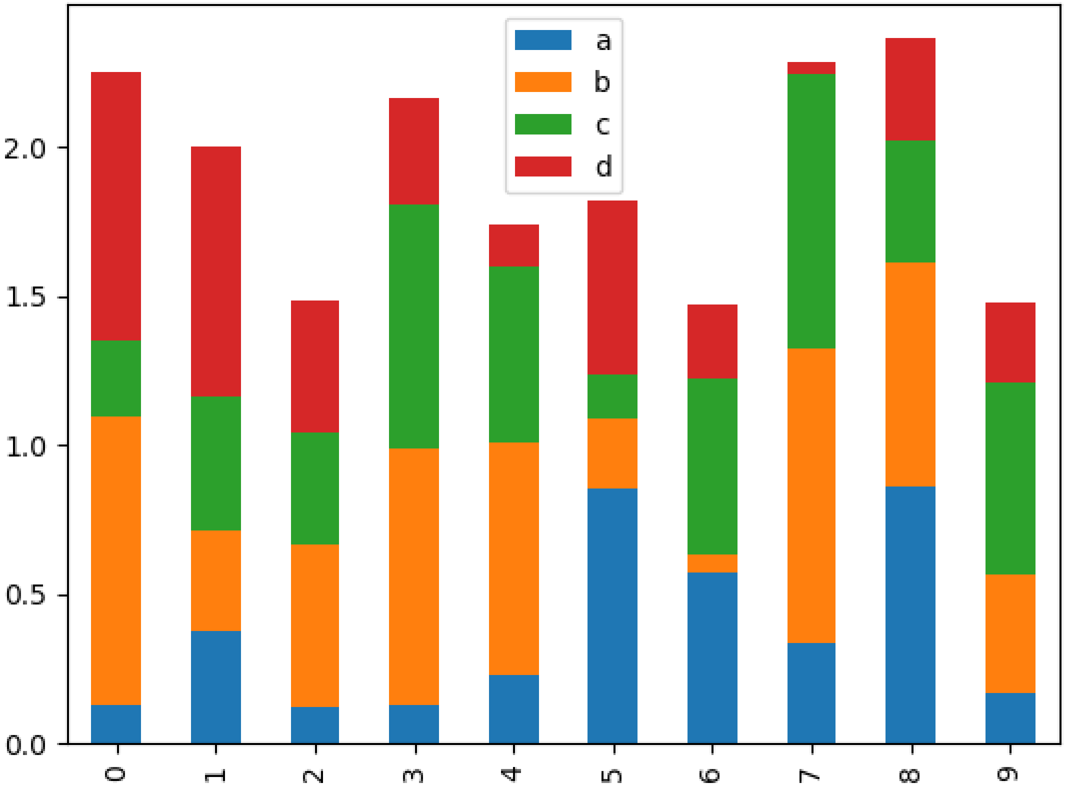
Calling a DataFrames `plot.bar()` method produces a multiple bar plot:

```
In [20]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [21]: df2.plot.bar();
```



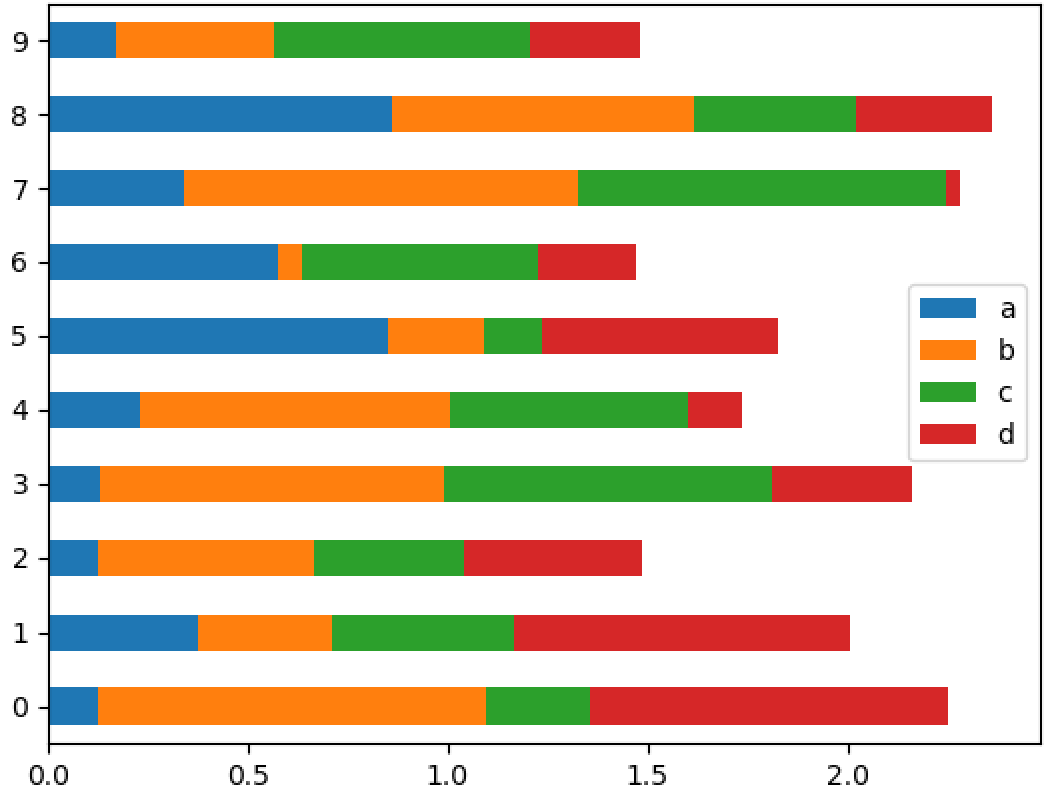
To produce a stacked bar plot, pass `stacked=True`:

```
In [22]: df2.plot.bar(stacked=True);
```



To get horizontal bar plots, use the `barh` method:

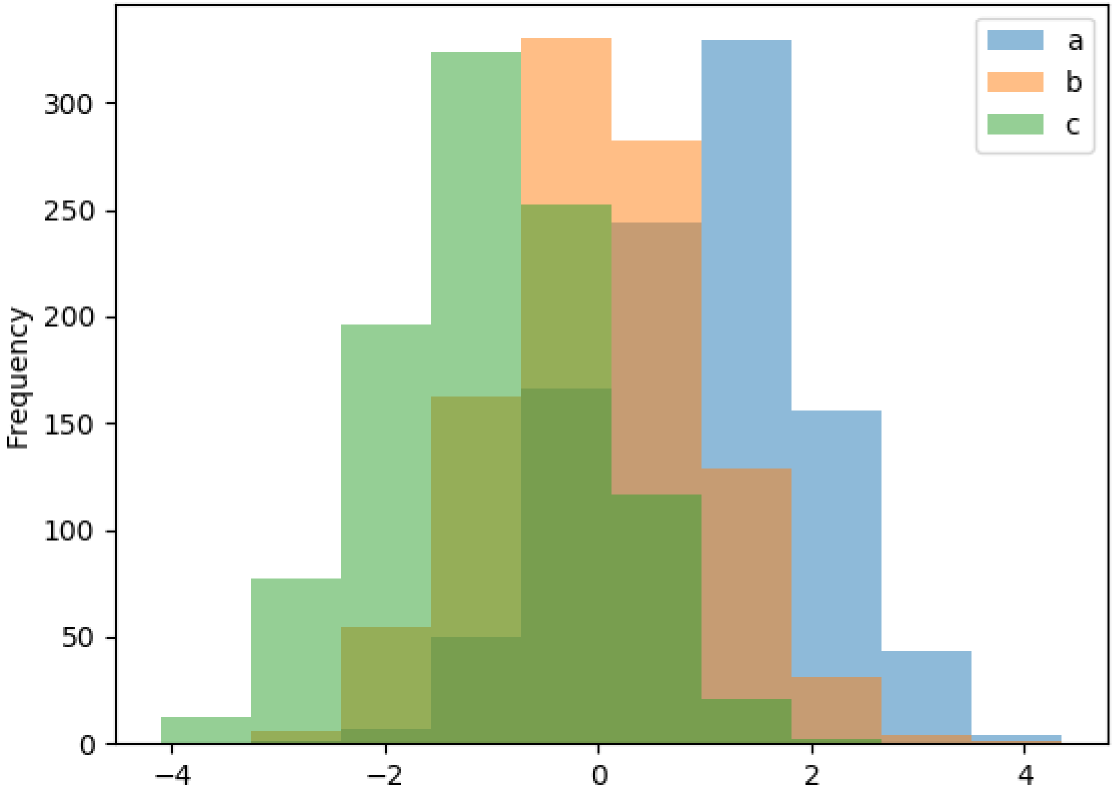
```
In [23]: df2.plot.barh(stacked=True);
```



Histograms

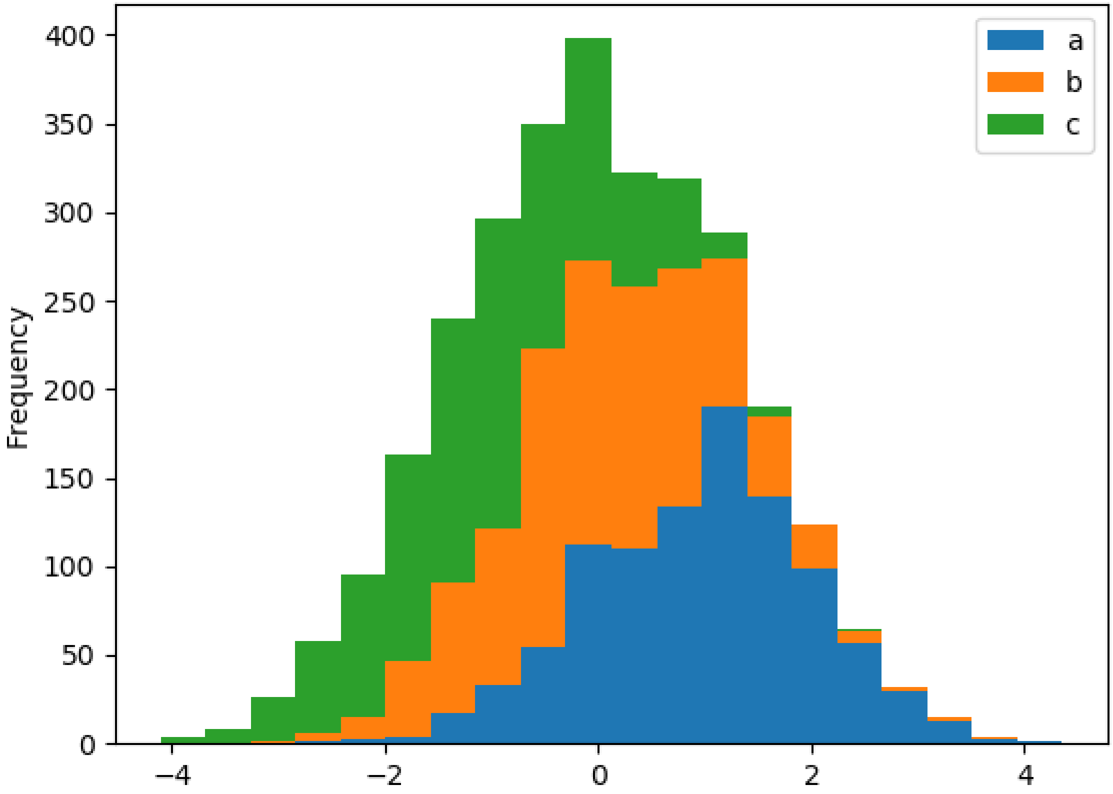
Histograms can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [24]: df4 = pd.DataFrame({'a': np.random.randn(1000) + 1, 'b': np.random.
↪randn(1000),
   ....:                     'c': np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])
   ....:

In [25]: plt.figure();

In [26]: df4.plot.hist(alpha=0.5)
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3af8db50>
```



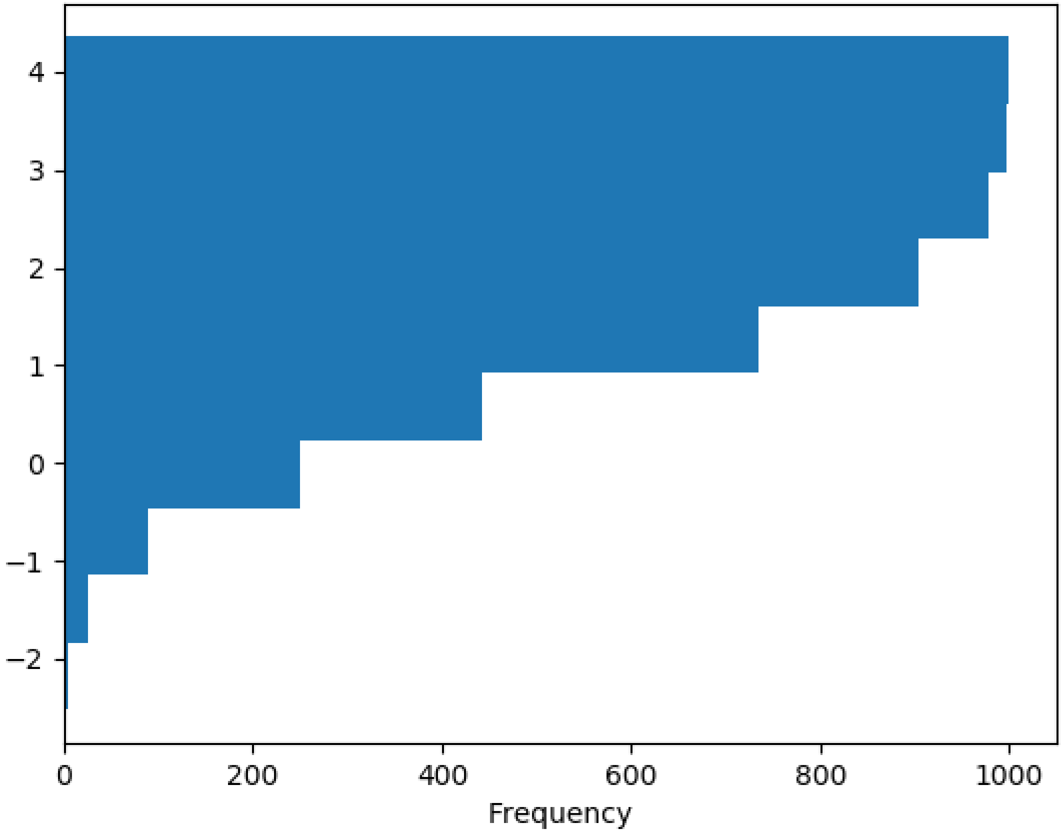
A histogram can be stacked using `stacked=True`. Bin size can be changed using the `bins` keyword.

```
In [27]: plt.figure();

In [28]: df4.plot.hist(stacked=True, bins=20)
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3ae12d90>
```



You can pass other keywords supported by matplotlib `hist`. For example, horizontal and cumulative histograms can be drawn by `orientation='horizontal'` and `cumulative=True`.

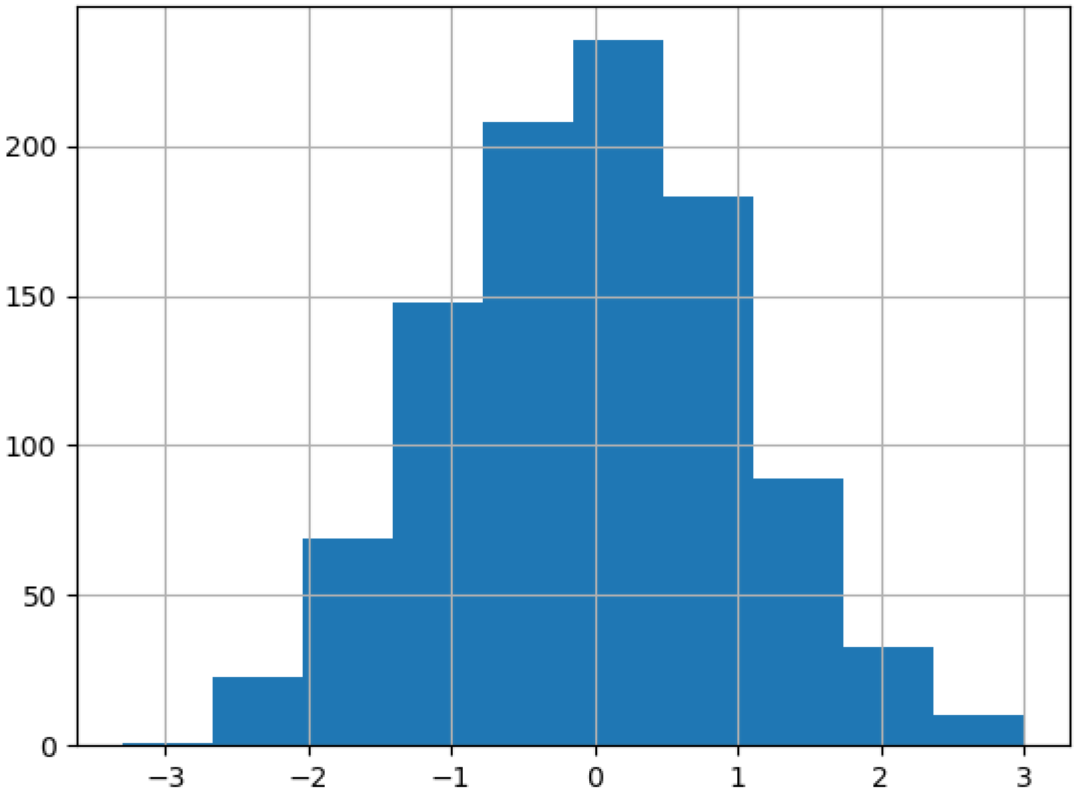
```
In [29]: plt.figure();

In [30]: df4['a'].plot.hist(orientation='horizontal', cumulative=True)
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3a171890>
```



See the `hist` method and the matplotlib hist documentation for more.

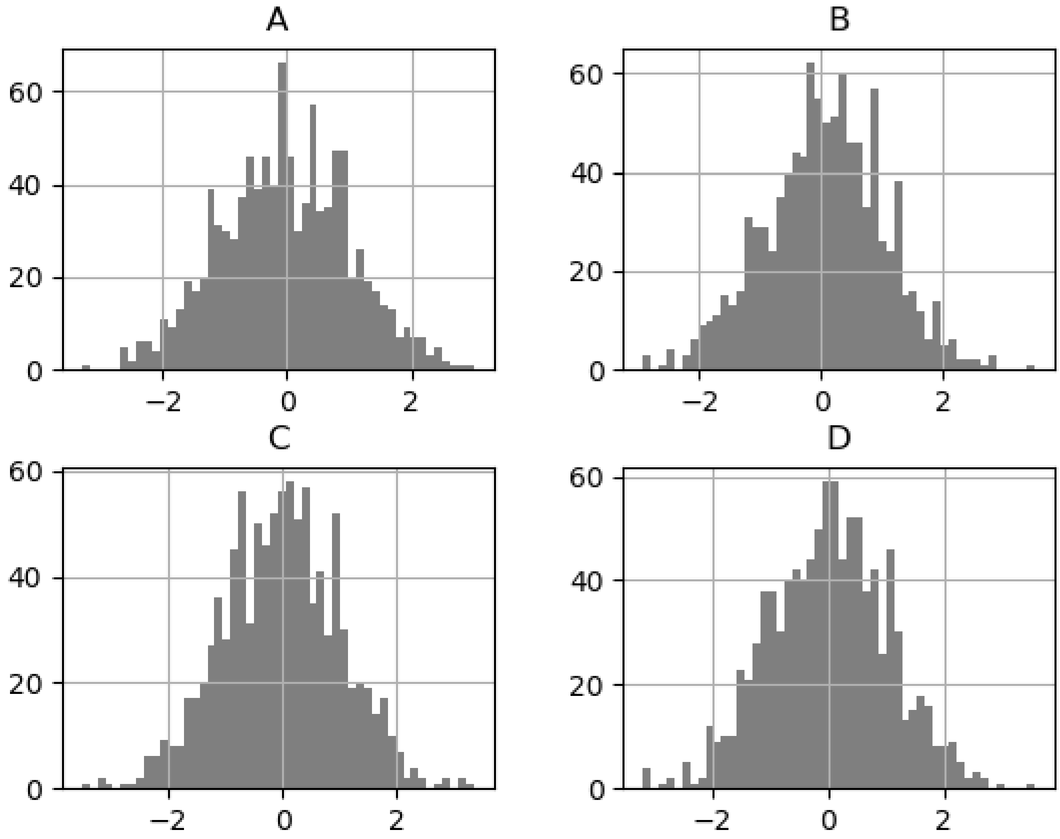
The existing interface `DataFrame.hist` to plot histogram still can be used.

```
In [31]: plt.figure();

In [32]: df['A'].diff().hist()
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3a1822d0>
```



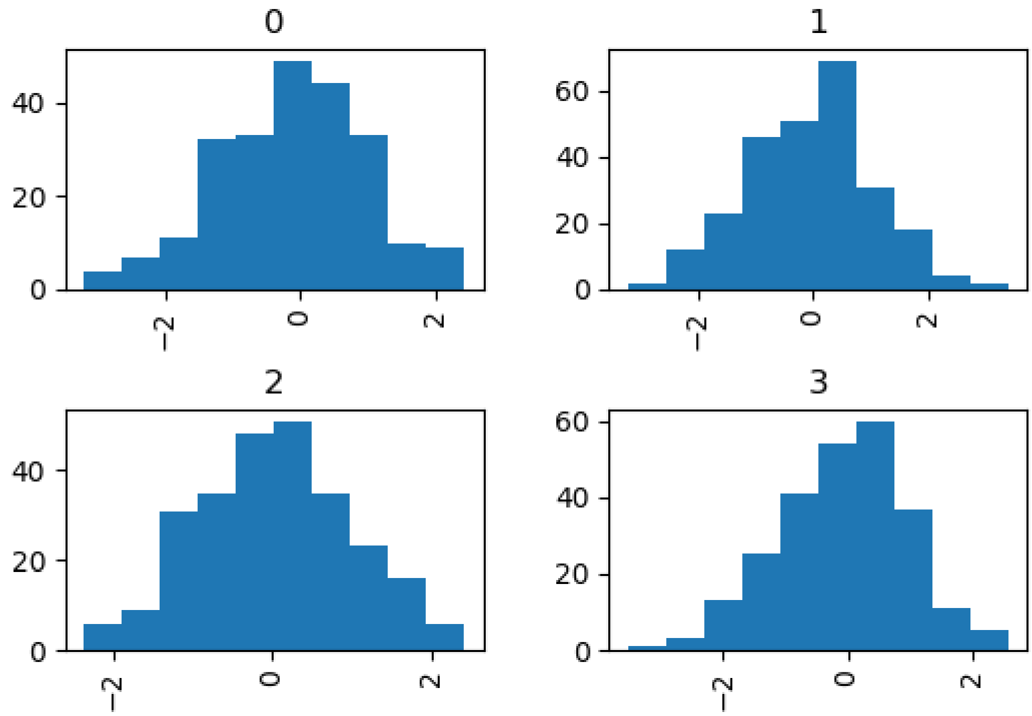
`DataFrame.hist()` plots the histograms of the columns on multiple subplots:

```
In [33]: plt.figure()
Out[33]: <Figure size 640x480 with 0 Axes>

In [34]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[34]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c39ff60d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3a3809d0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c36c54c90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c36b5df90>]],
      dtype=object)
```



The `by` keyword can be specified to plot grouped histograms:

```
In [35]: data = pd.Series(np.random.randn(1000))

In [36]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[36]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c39ff6110>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c4042b810>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c404b3e50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c404ed6d0>]],
      dtype=object)
```

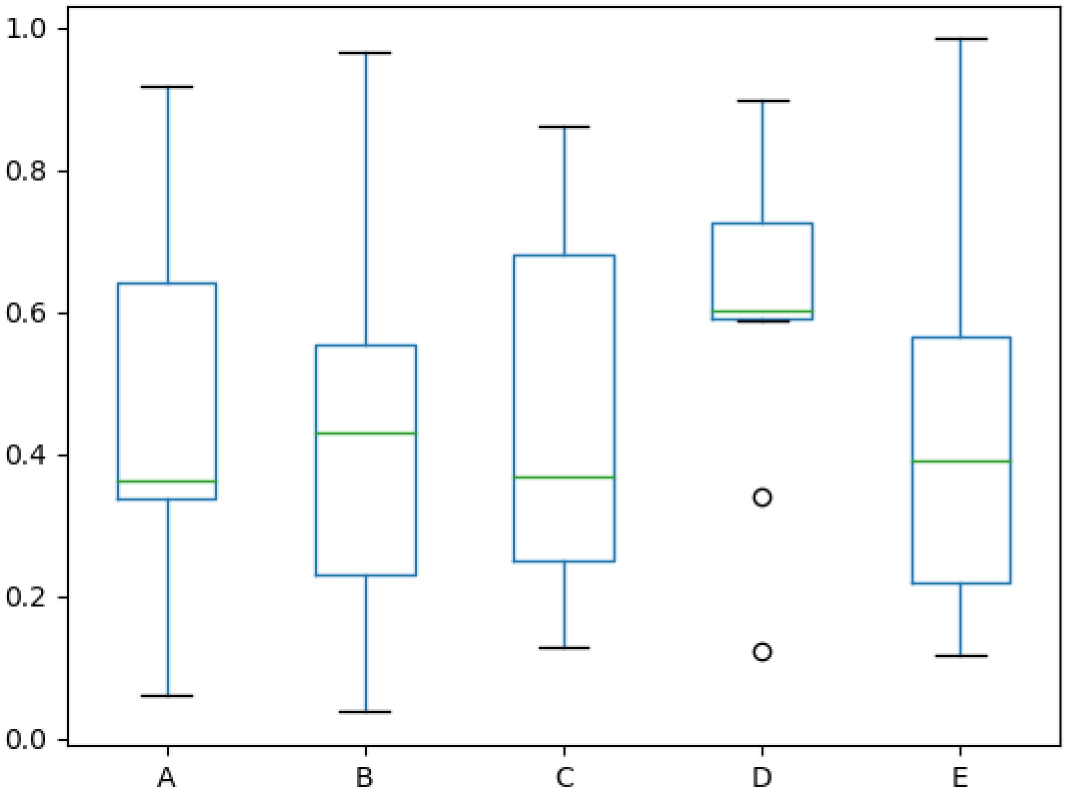


Box plots

Boxplot can be drawn calling `Series.plot.box()` and `DataFrame.plot.box()`, or `DataFrame.boxplot()` 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 [37]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])

In [38]: df.plot.box()
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x1c405fc410>
```

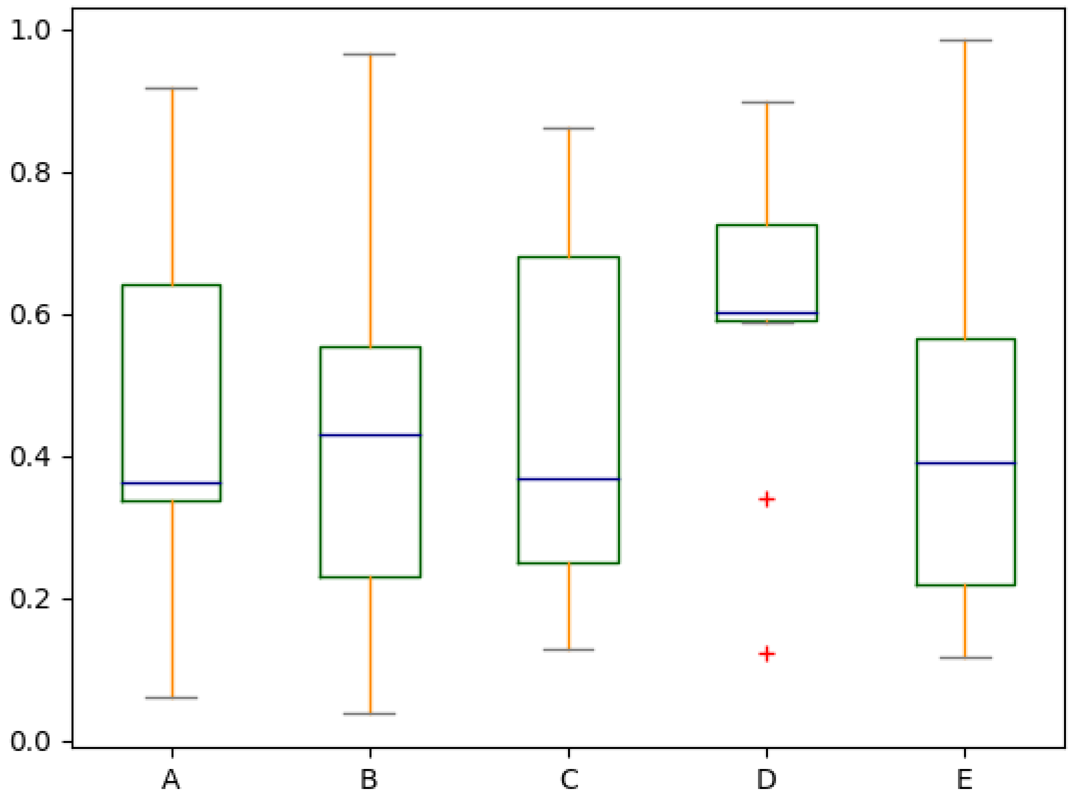


Boxplot can be colorized by passing `color` keyword. You can pass a `dict` whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the `dict`, default colors are used for the corresponding artists. Also, boxplot has `sym` keyword to specify fliers style.

When you pass other type of arguments via `color` keyword, it will be directly passed to matplotlib for all the `boxes`, `whiskers`, `medians` and `caps` colorization.

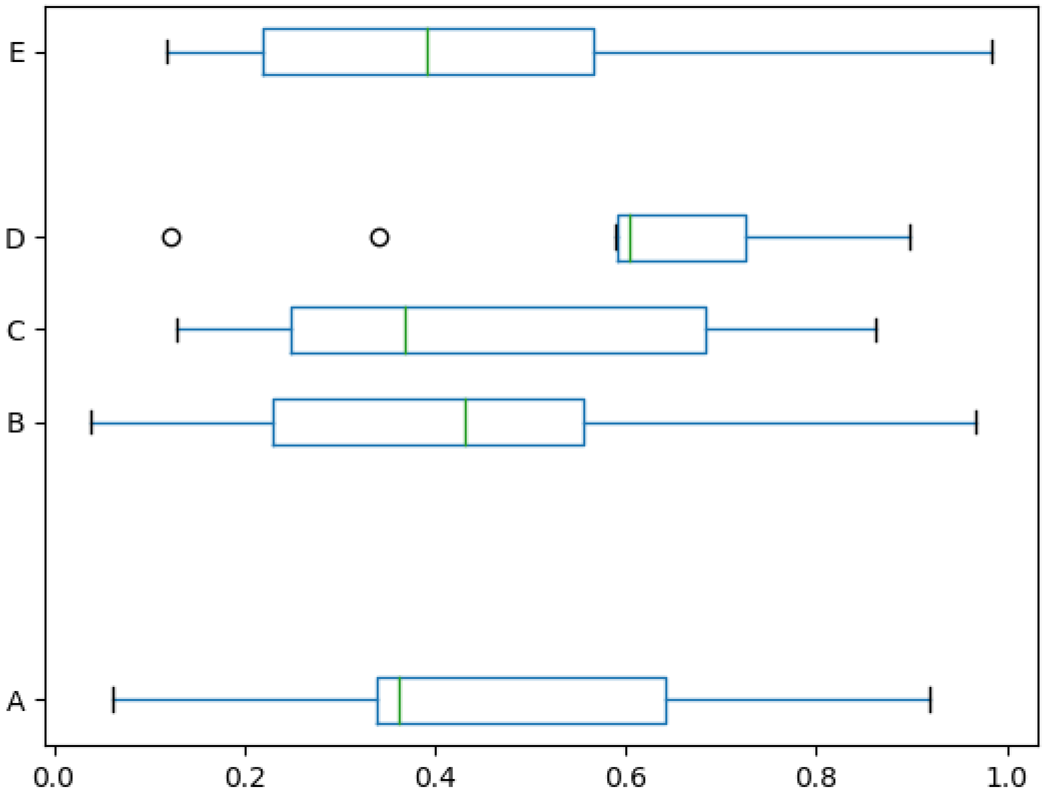
The colors are applied to every boxes to be drawn. If you want more complicated colorization, you can get each drawn artists by passing *return_type*.

```
In [39]: color = {'boxes': 'DarkGreen', 'whiskers': 'DarkOrange',
   ....:          'medians': 'DarkBlue', 'caps': 'Gray'}
   ....:

In [40]: df.plot.box(color=color, sym='r+')
Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4079d710>
```



Also, you can pass other keywords supported by matplotlib `boxplot`. For example, horizontal and custom-positioned boxplot can be drawn by `vert=False` and `positions` keywords.

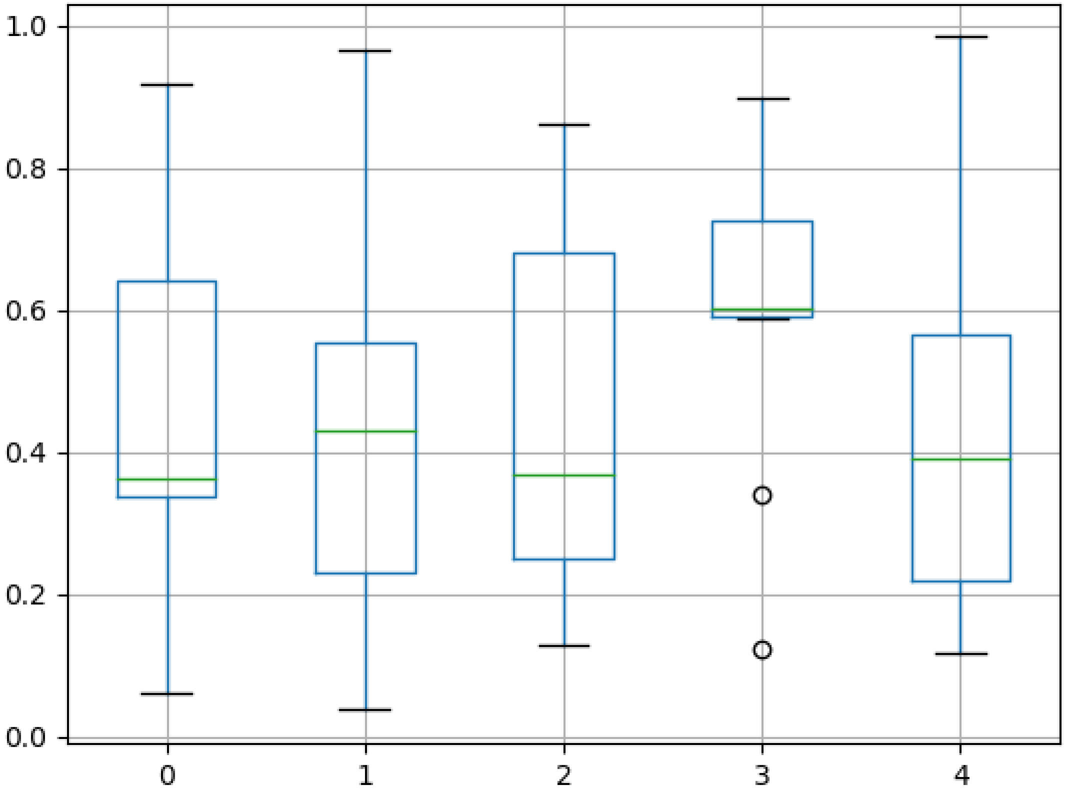
```
In [41]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8])
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4074dbd0>
```



See the boxplot method and the matplotlib boxplot documentation for more.

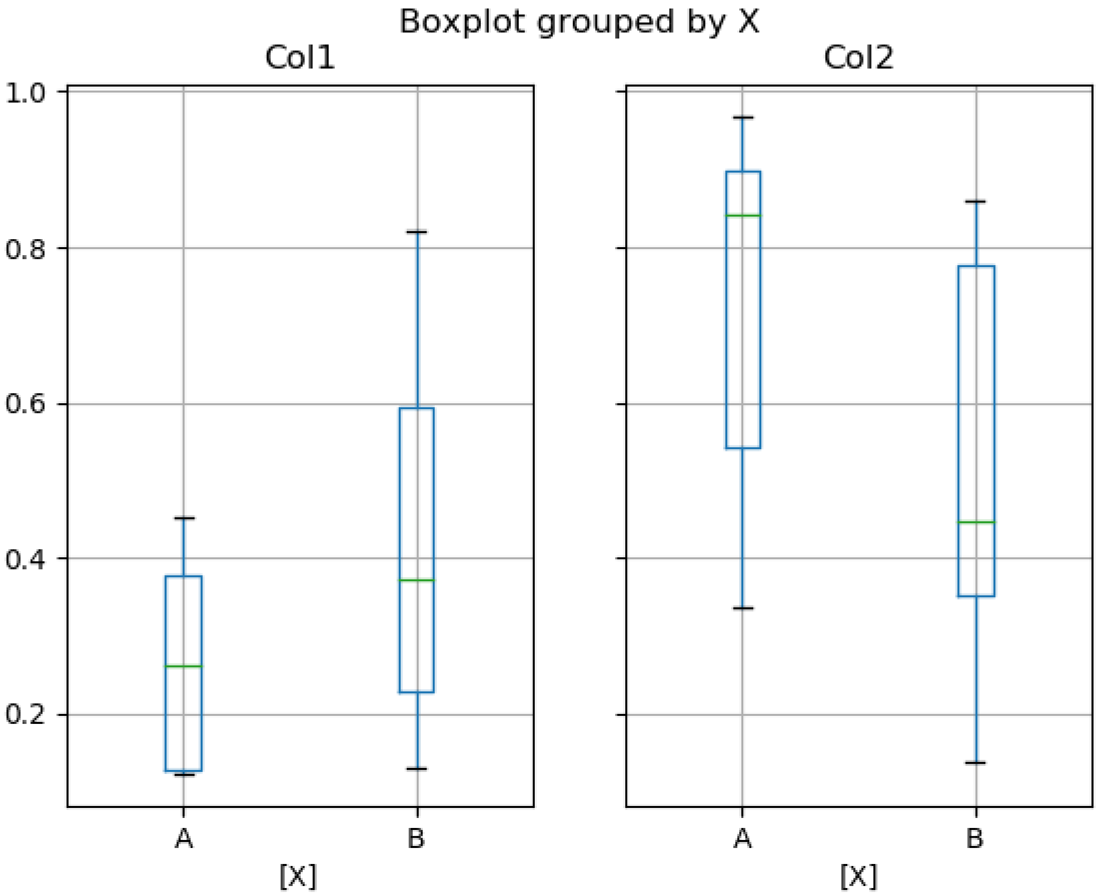
The existing interface DataFrame.boxplot to plot boxplot still can be used.

```
In [42]: df = pd.DataFrame(np.random.rand(10, 5))

In [43]: plt.figure();

In [44]: bp = df.boxplot()
```



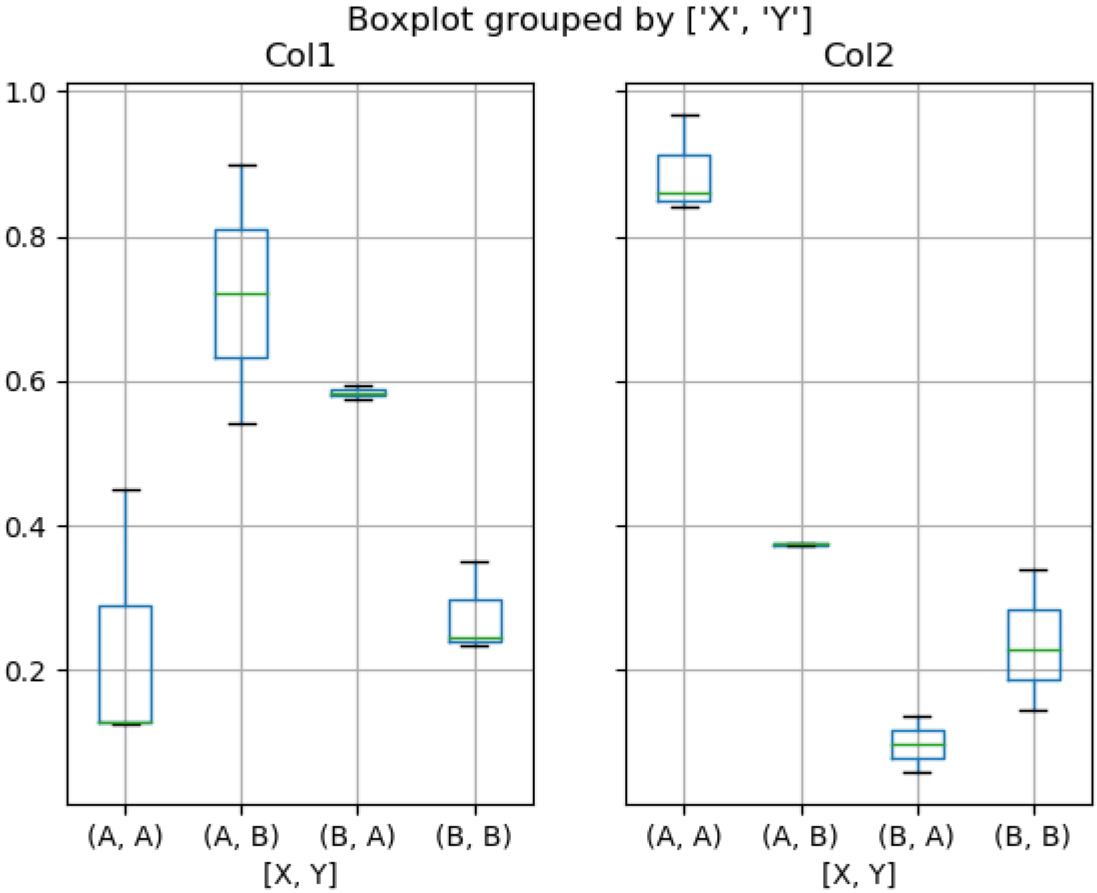
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [45]: df = pd.DataFrame(np.random.rand(10, 2), columns=['Col1', 'Col2'])

In [46]: df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B'])

In [47]: plt.figure();

In [48]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [49]: df = pd.DataFrame(np.random.rand(10, 3), columns=['Col1', 'Col2', 'Col3'])

In [50]: df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B'])

In [51]: df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A', 'B', 'A', 'B', 'A', 'B'])

In [52]: plt.figure();

In [53]: bp = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```



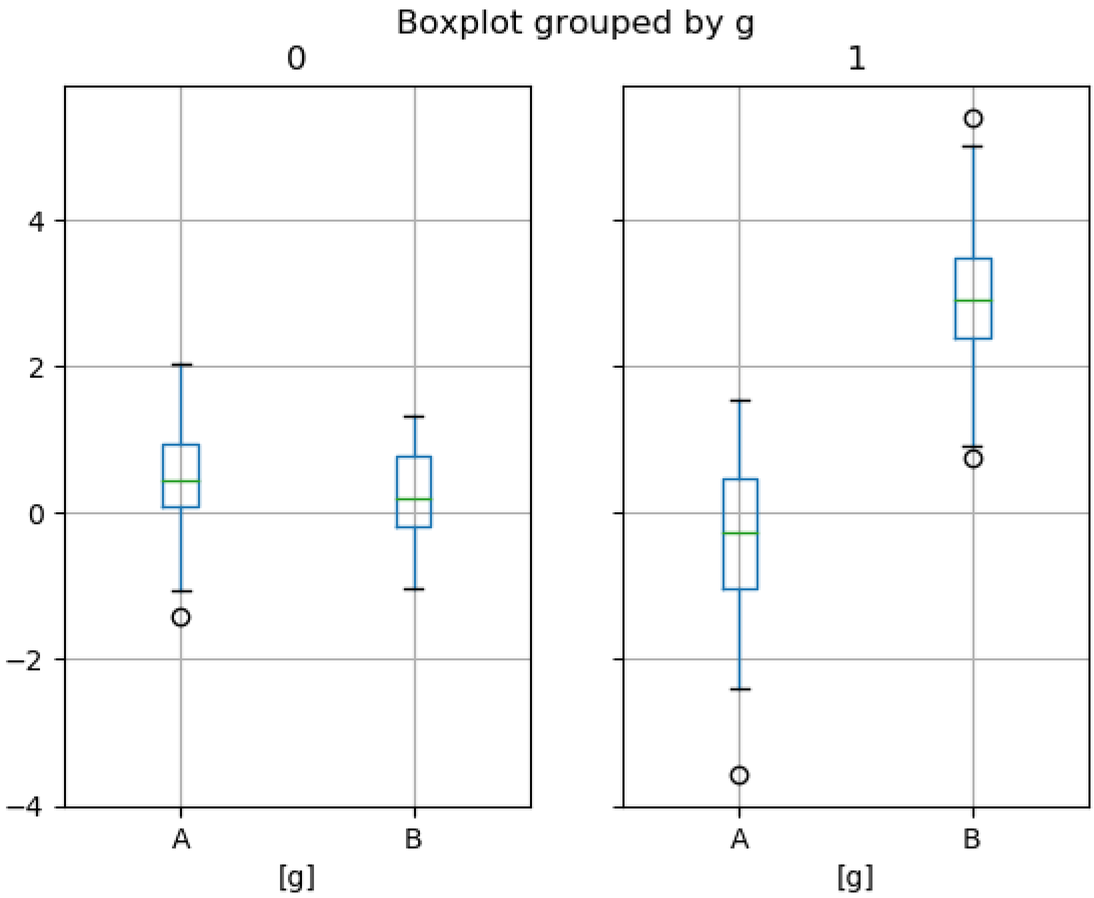
Warning: The default changed from `'dict'` to `'axes'` in version 0.19.0.

In `boxplot`, the return type can be controlled by the `return_type`, keyword. The valid choices are `{"axes", "dict", "both", None}`. Faceting, created by `DataFrame.boxplot` with the `by` keyword, will affect the output type as well:

`return_type=`	Faceted	Output type
`None`	No	axes
`None`	Yes	2-D ndarray of axes
`'axes'`	No	axes
`'axes'`	Yes	Series of axes
`'dict'`	No	dict of artists
`'dict'`	Yes	Series of dicts of artists
`'both'`	No	namedtuple
`'both'`	Yes	Series of namedtuples

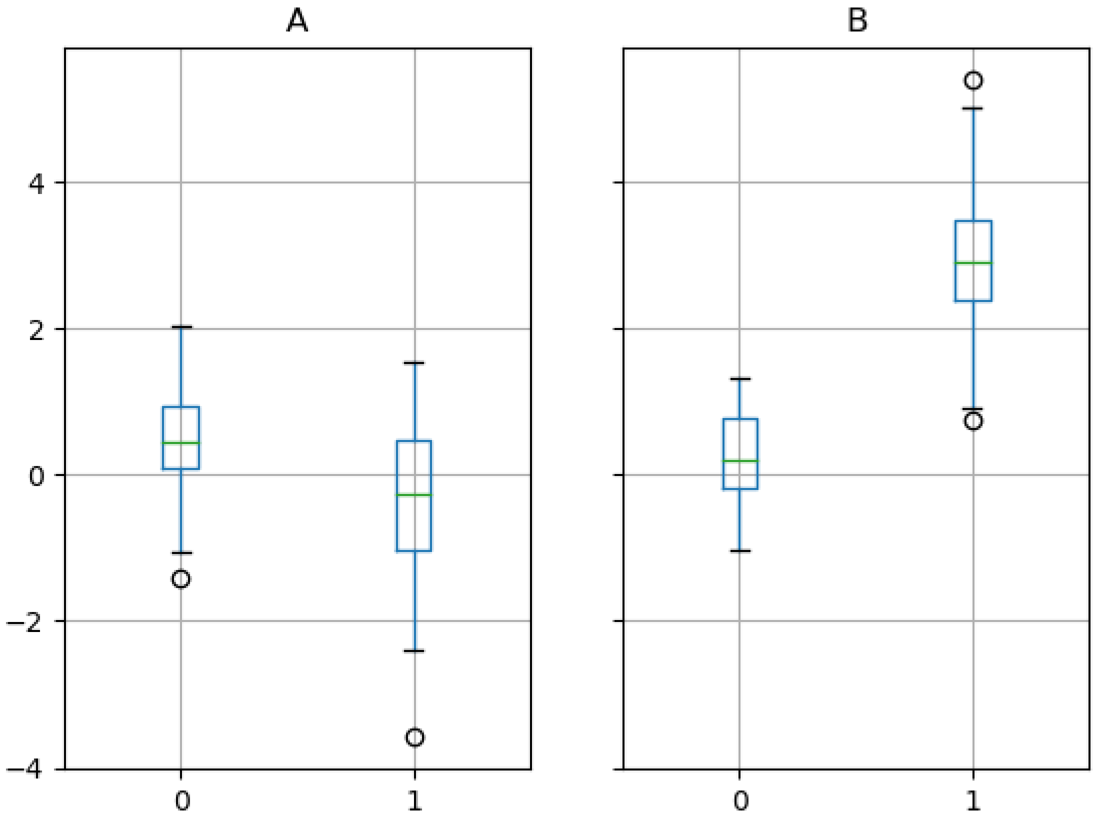
`Groupby.boxplot` always returns a `Series` of `return_type`.

```
In [54]: np.random.seed(1234)

In [55]: df_box = pd.DataFrame(np.random.randn(50, 2))

In [56]: df_box['g'] = np.random.choice(['A', 'B'], size=50)

In [57]: df_box.loc[df_box['g'] == 'B', 1] += 3

In [58]: bp = df_box.boxplot(by='g')
```



The subplots above are split by the numeric columns first, then the value of the g column. Below the subplots are first split by the value of g, then by the numeric columns.

```
In [59]: bp = df_box.groupby('g').boxplot()
```

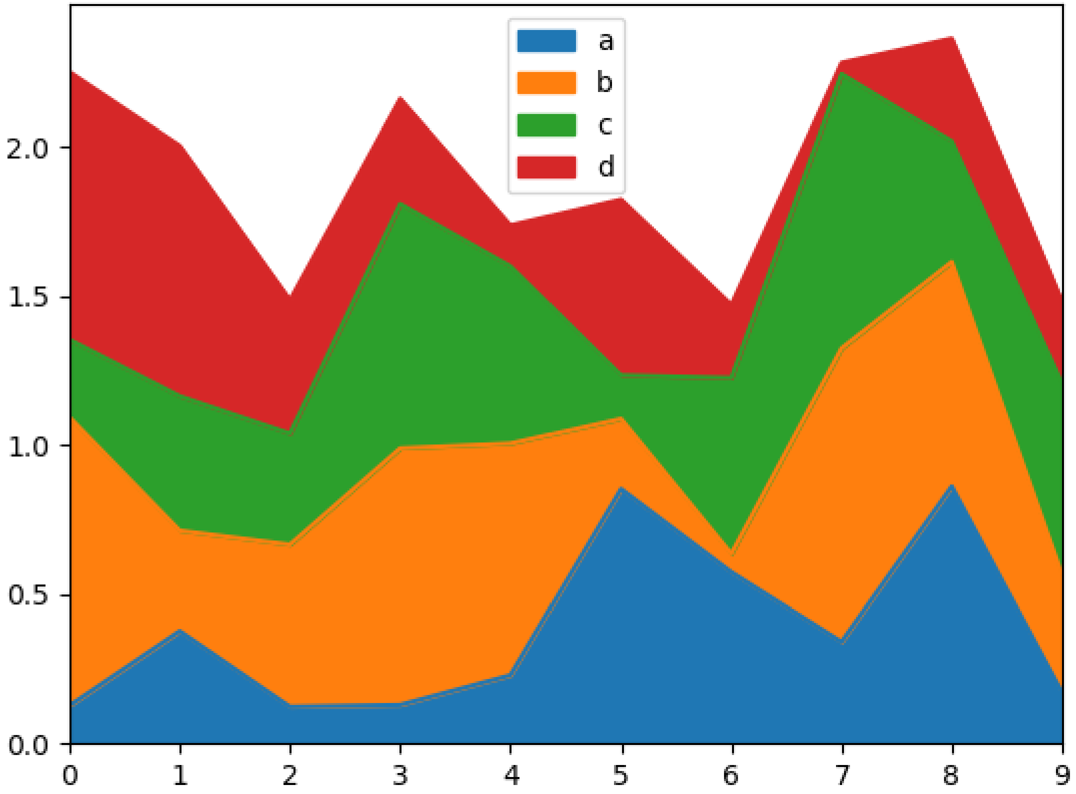


Area plot

You can create area plots with `Series.plot.area()` and `DataFrame.plot.area()`. Area plots are stacked by default. To produce stacked area plot, each column must be either all positive or all negative values.

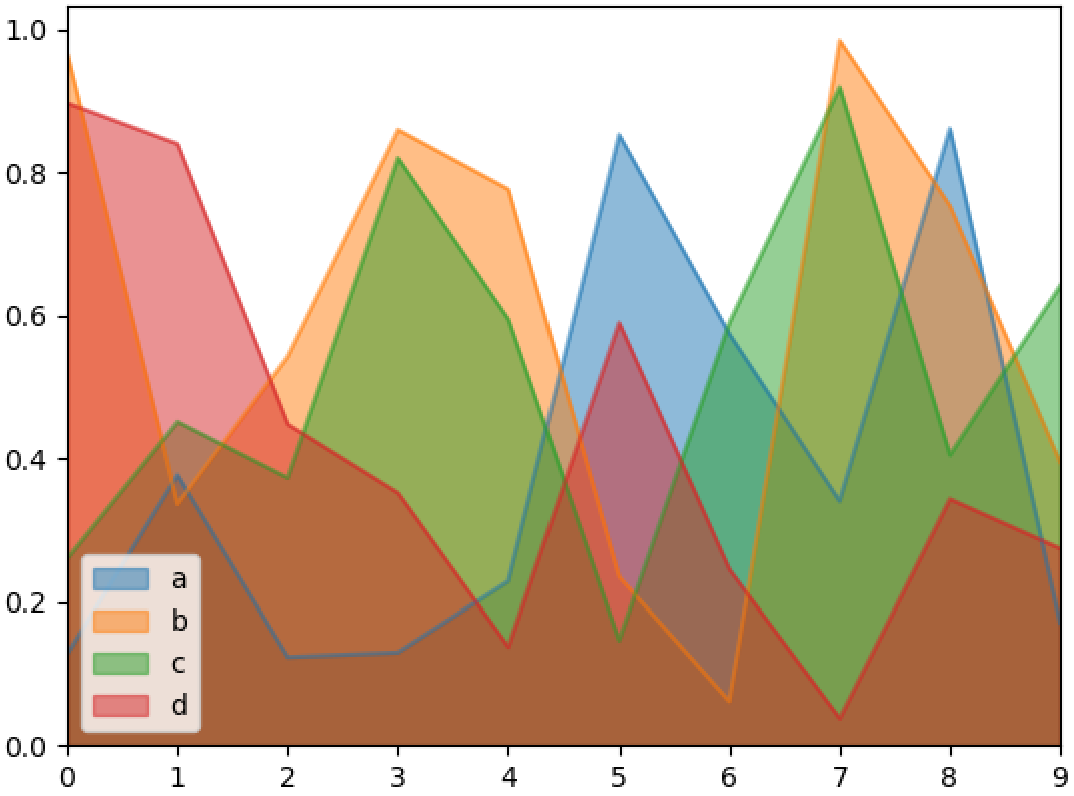
When input data contains *NaN*, it will be automatically filled by 0. If you want to drop or fill by different values, use `dataframe.dropna()` or `dataframe.fillna()` before calling *plot*.

```
In [60]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [61]: df.plot.area();
```



To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

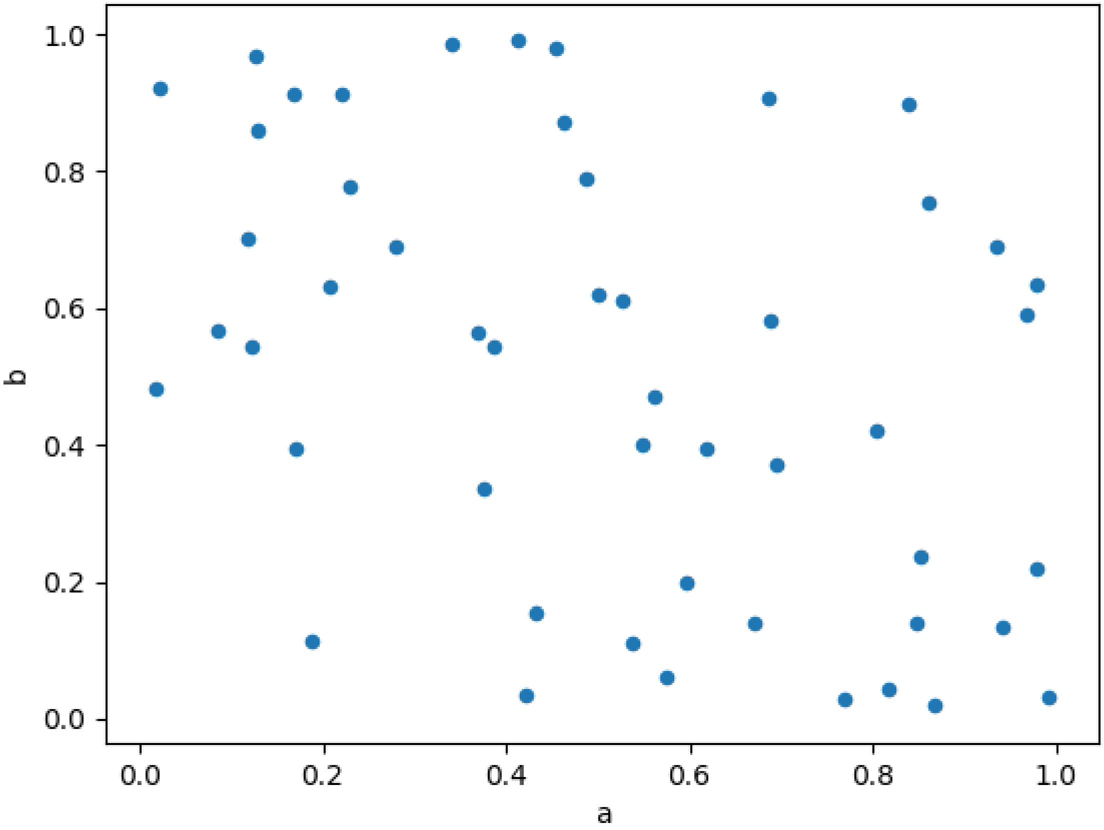
```
In [62]: df.plot.area(stacked=False);
```



Scatter plot

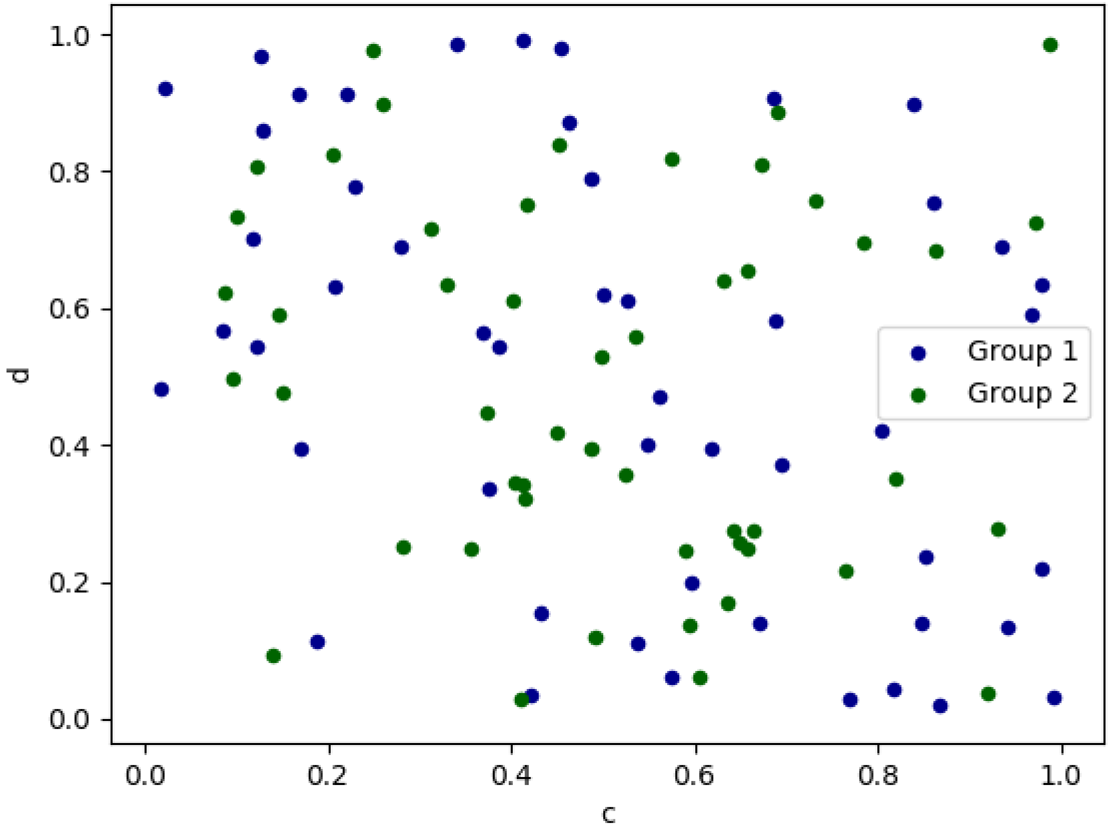
Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for the x and y axes. These can be specified by the `x` and `y` keywords.

```
In [63]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])

In [64]: df.plot.scatter(x='a', y='b');
```



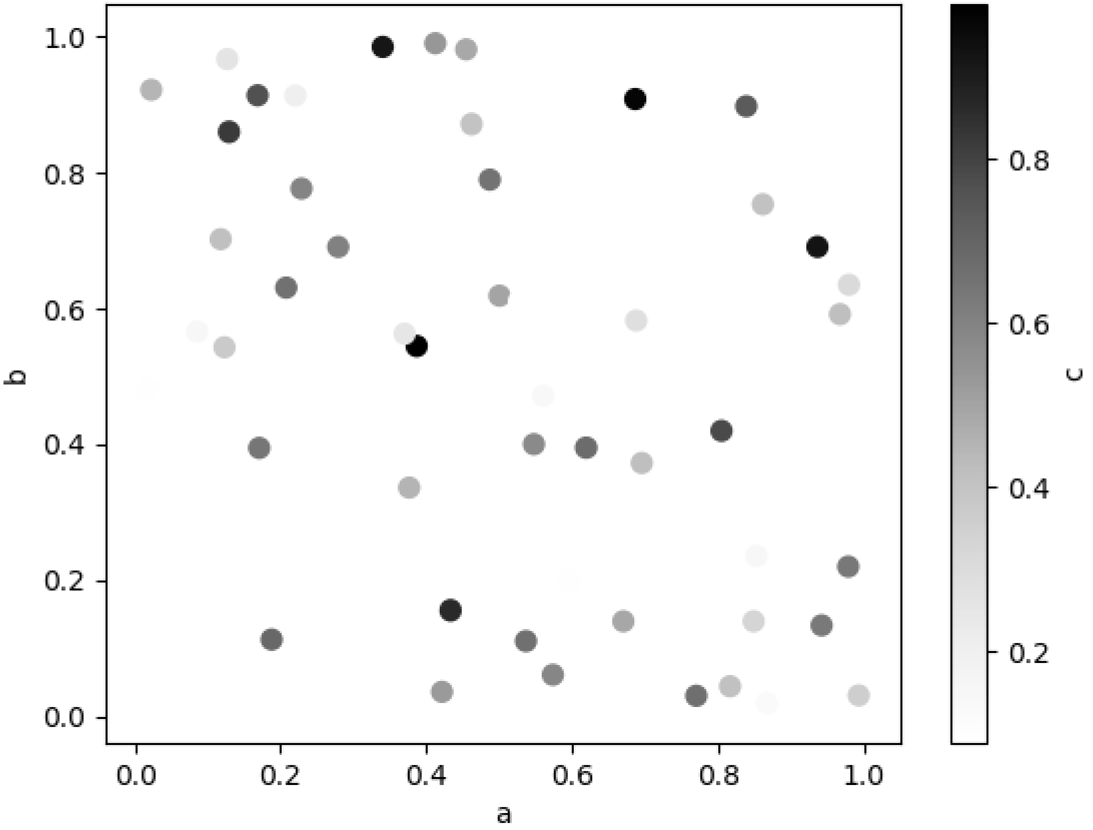
To plot multiple column groups in a single axes, repeat `plot` method specifying target `ax`. It is recommended to specify `color` and `label` keywords to distinguish each groups.

```
In [65]: ax = df.plot.scatter(x='a', y='b', color='DarkBlue', label='Group 1');

In [66]: df.plot.scatter(x='c', y='d', color='DarkGreen', label='Group 2', ax=ax);
```



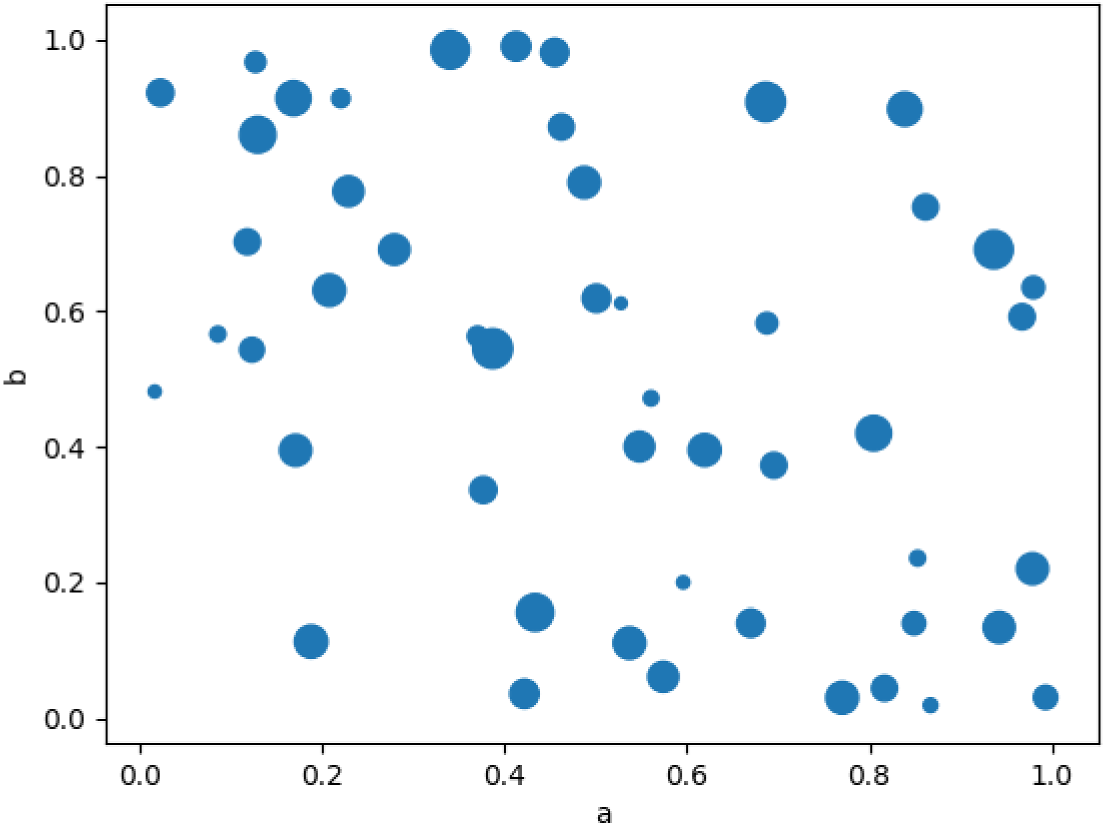
The keyword `c` may be given as the name of a column to provide colors for each point:

```
In [67]: df.plot.scatter(x='a', y='b', c='c', s=50);
```



You can pass other keywords supported by matplotlib `scatter`. The example below shows a bubble chart using a column of the `DataFrame` as the bubble size.

```
In [68]: df.plot.scatter(x='a', y='b', s=df['c'] * 200);
```

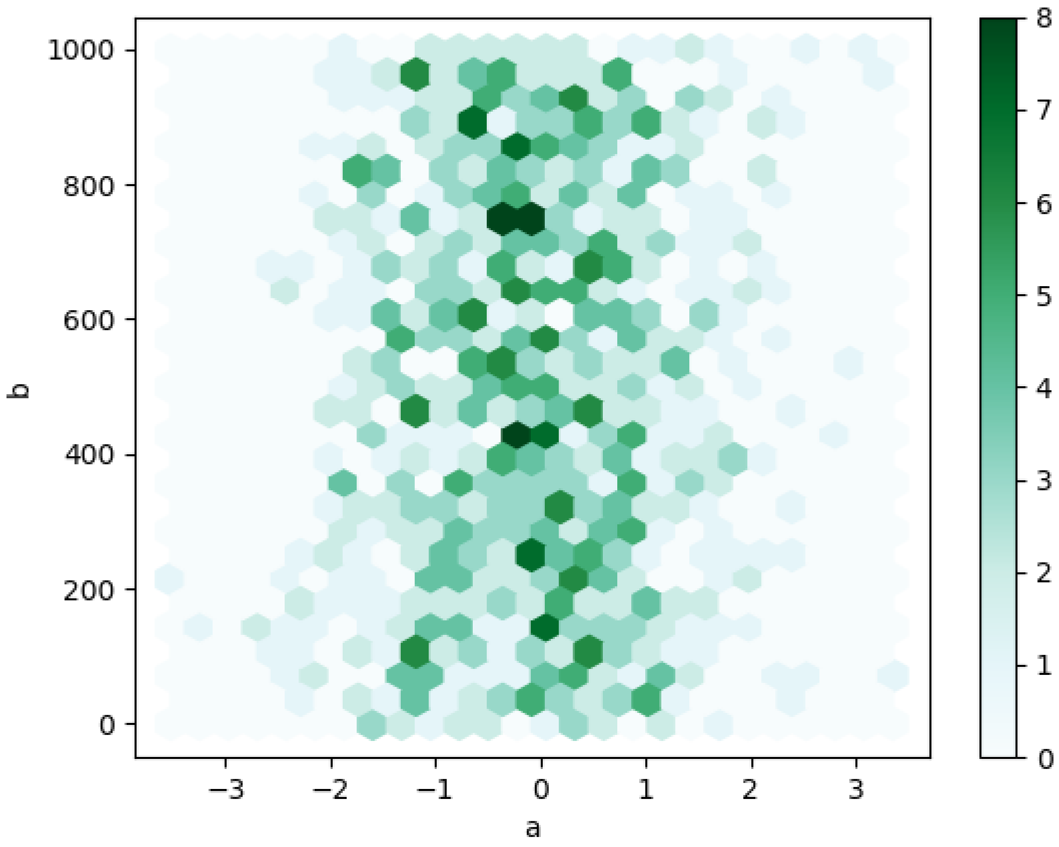


See the scatter method and the matplotlib scatter documentation for more.

Hexagonal bin plot

You can create hexagonal bin plots with DataFrame.plot.hexbin(). Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

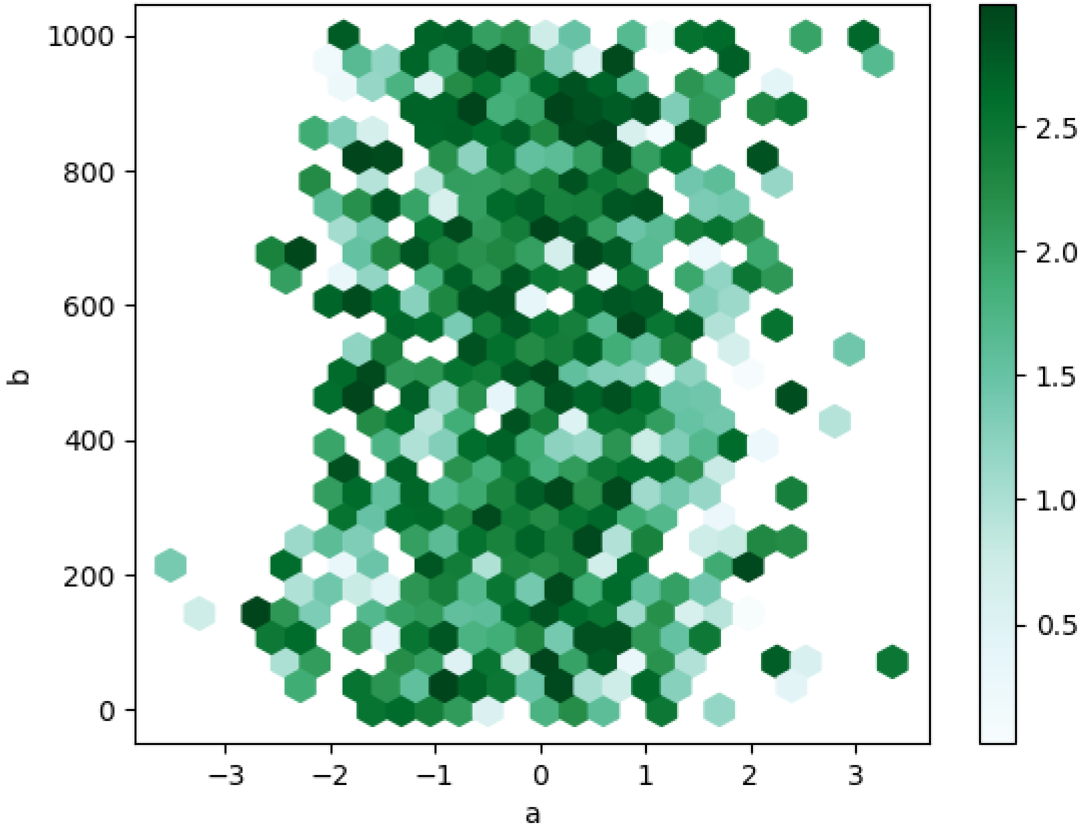
```
In [69]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

In [70]: df['b'] = df['b'] + np.arange(1000)

In [71]: df.plot.hexbin(x='a', y='b', gridsize=25)
Out[71]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3bc43c90>
```



A useful keyword argument is gridsize; it controls the number of hexagons in the x-direction, and defaults to 100. A larger gridsize means more, smaller bins.

By default, a histogram of the counts around each (x, y) point is computed. You can specify alternative aggregations by passing values to the C and reduce_C_function arguments. C specifies the value at each (x, y) point and reduce_C_function is a function of one argument that reduces all the values in a bin to a single number (e.g. mean, max, sum, std). In this example the positions are given by columns a and b, while the value is given by column z. The bins are aggregated with NumPys max function.

```
In [72]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

In [73]: df['b'] = df['b'] = df['b'] + np.arange(1000)

In [74]: df['z'] = np.random.uniform(0, 3, 1000)

In [75]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max, gridsize=25)
Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0x1c39a8cf90>
```

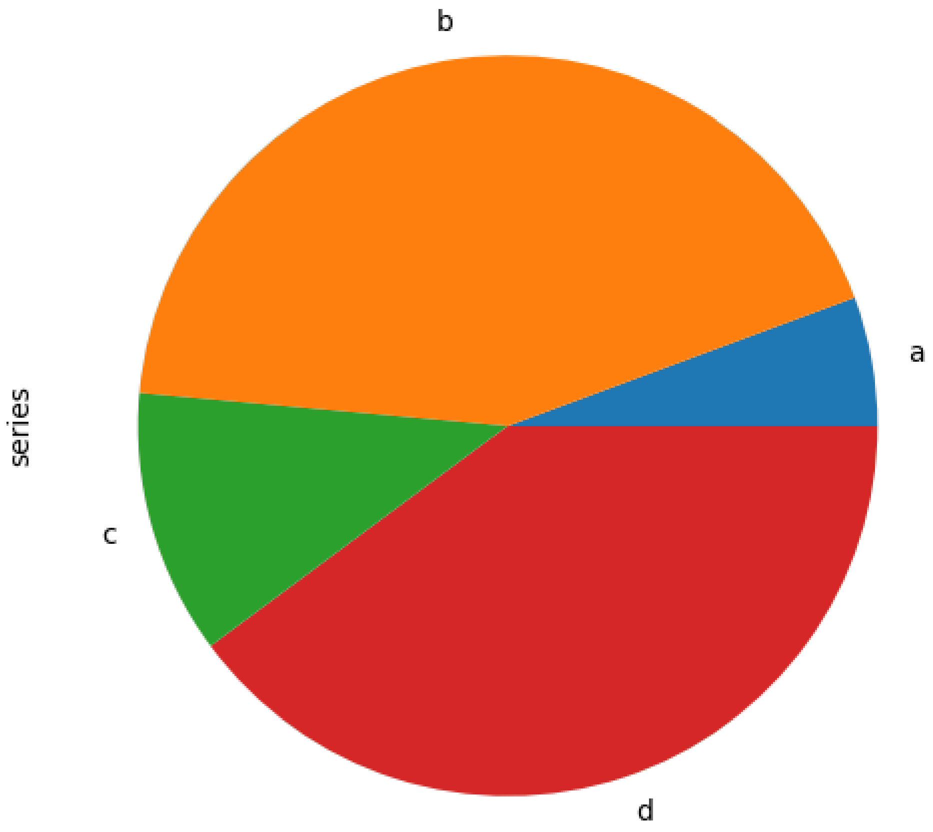


See the hexbin method and the matplotlib hexbin documentation for more.

Pie plot

You can create a pie plot with DataFrame.plot.pie() or Series.plot.pie(). If your data includes any NaN, they will be automatically filled with 0. A ValueError will be raised if there are any negative values in your data.

```
In [76]: series = pd.Series(3 * np.random.rand(4),
   ....:                    index=['a', 'b', 'c', 'd'], name='series')
   ....:

In [77]: series.plot.pie(figsize=(6, 6))
Out[77]: <matplotlib.axes._subplots.AxesSubplot at 0x1257b7a10>
```



For pie plots its best to use square figures, i.e. a figure aspect ratio 1. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling `ax.set_aspect('equal')` on the returned `axes` object.

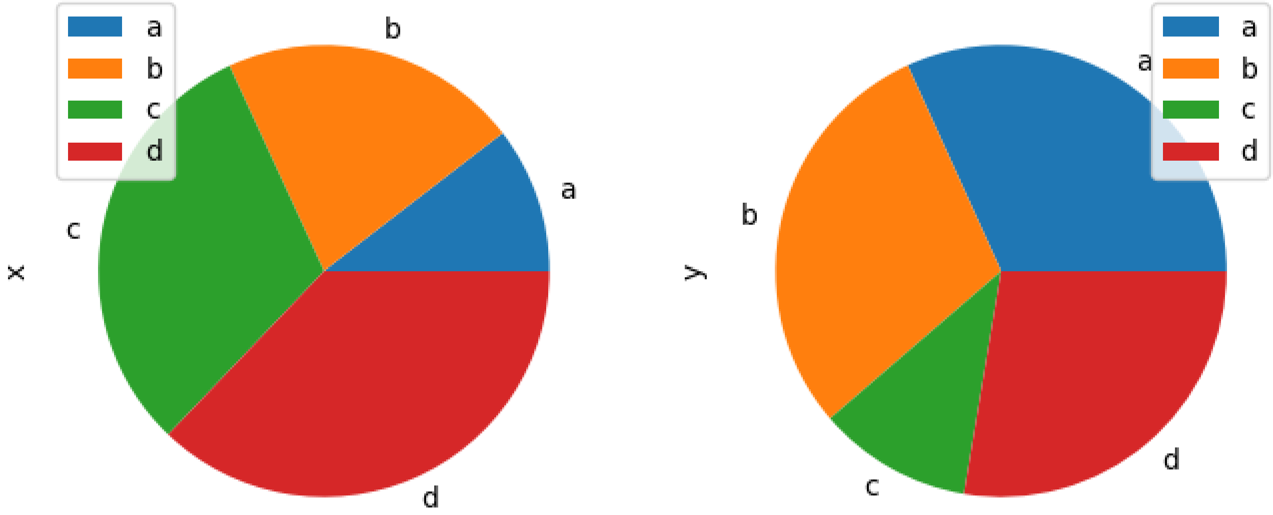
Note that pie plot with `DataFrame` requires that you either specify a target column by the `y` argument or `subplots=True`. When `y` is specified, pie plot of selected column will be drawn. If `subplots=True` is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify `legend=False` to hide it.

```
In [78]: df = pd.DataFrame(3 * np.random.rand(4, 2),
   ....:                   index=['a', 'b', 'c', 'd'], columns=['x', 'y'])
   ....:

In [79]: df.plot.pie(subplots=True, figsize=(8, 4))
Out[79]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x1c39e89a10>,
```

(continues on next page)

(continued from previous page)

```
       <matplotlib.axes._subplots.AxesSubplot object at 0x1c399678d0>],
      dtype=object)
```

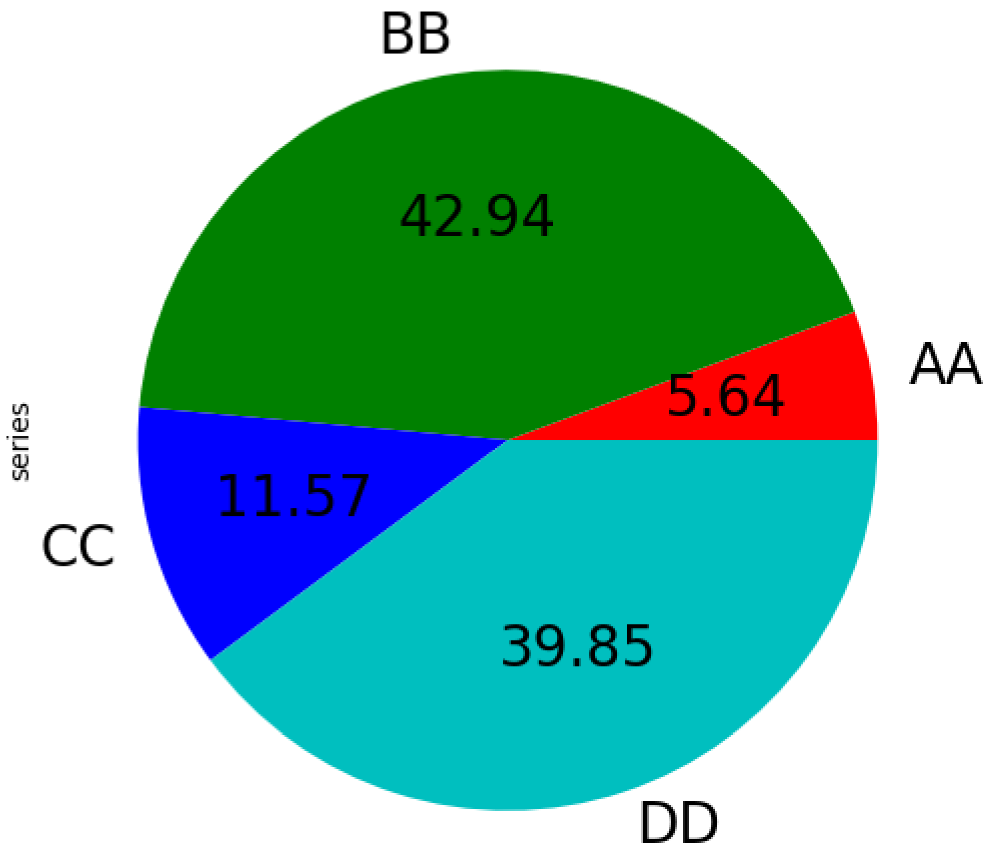


You can use the labels and colors keywords to specify the labels and colors of each wedge.

Warning: Most pandas plots use the label and color arguments (note the lack of s on those). To be consistent with matplotlib.pyplot.pie() you must use labels and colors.

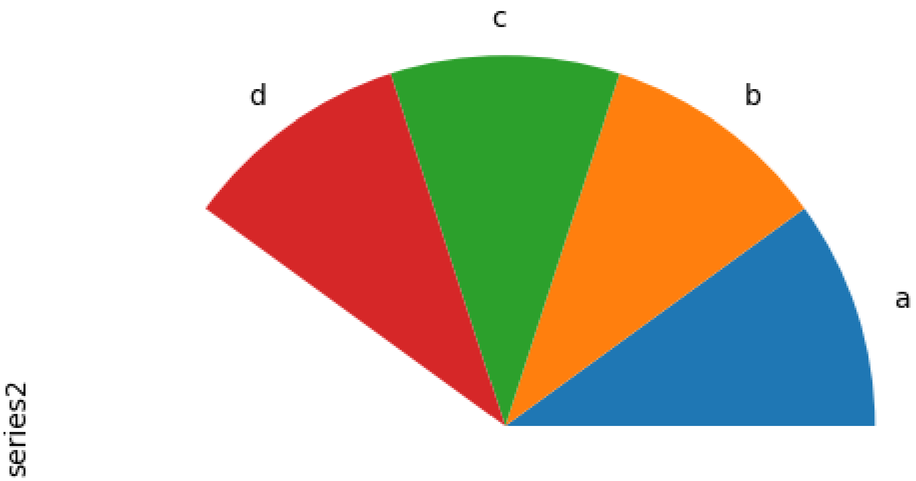
If you want to hide wedge labels, specify labels=None. If fontsize is specified, the value will be applied to wedge labels. Also, other keywords supported by matplotlib.pyplot.pie() can be used.

```
In [80]: series.plot.pie(labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
   ....:                 autopct='%.2f', fontsize=20, figsize=(6, 6))
   ....:
Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2e007310>
```



If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```
In [81]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')

In [82]: series.plot.pie(figsize=(6, 6))
Out[82]: <matplotlib.axes._subplots.AxesSubplot at 0x1c38678390>
```



See the matplotlib pie documentation for more.

4.10.3 Plotting with missing data

Pandas tries to be pragmatic about plotting `DataFrames` or `Series` that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.

Plot Type	NaN Handling
Line	Leave gaps at NaNs
Line (stacked)	Fill 0s
Bar	Fill 0s
Scatter	Drop NaNs
Histogram	Drop NaNs (column-wise)
Box	Drop NaNs (column-wise)
Area	Fill 0s
KDE	Drop NaNs (column-wise)
Hexbin	Drop NaNs
Pie	Fill 0s

If any of these defaults are not what you want, or if you want to be explicit about how missing values are handled, consider using *fillna()* or *dropna()* before plotting.

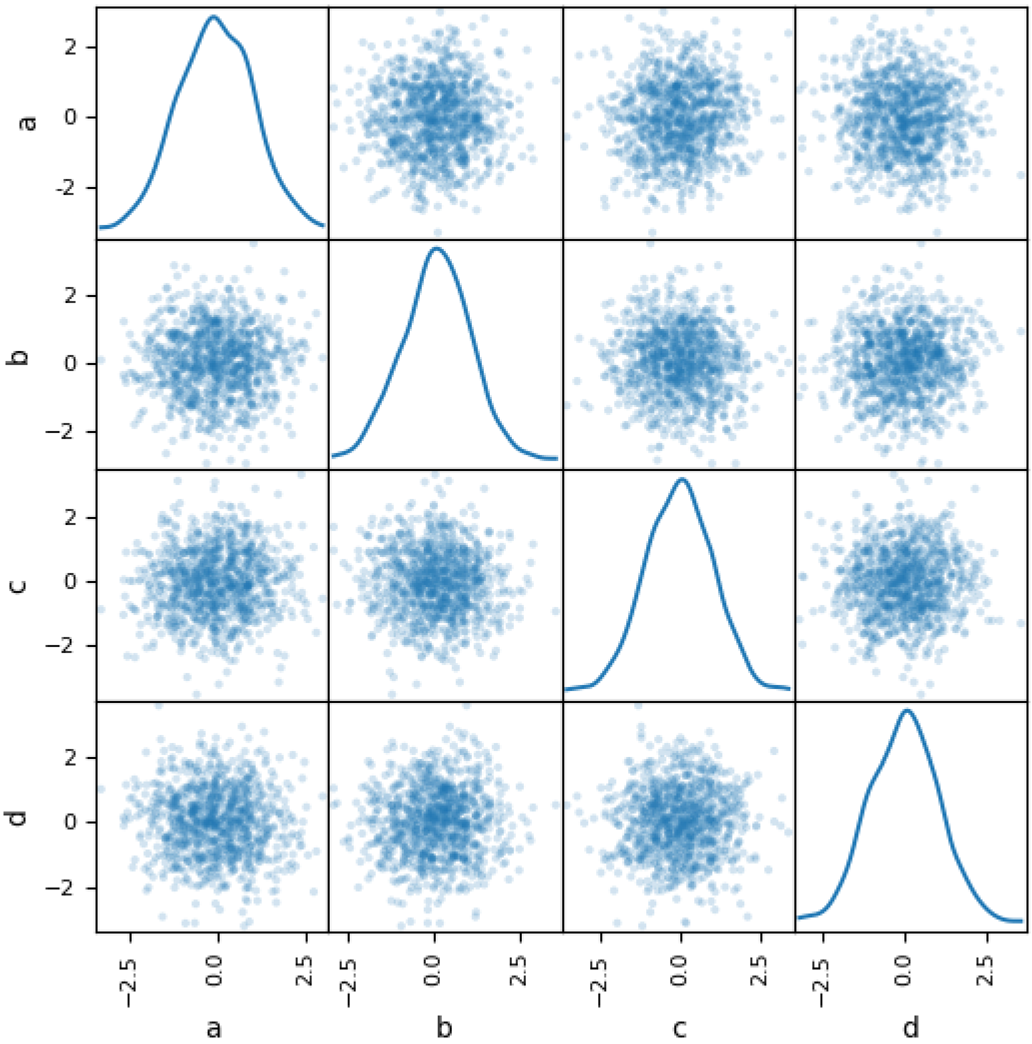
4.10.4 Plotting Tools

These functions can be imported from `pandas.plotting` and take a `Series` or `DataFrame` as an argument.

Scatter matrix plot

You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.plotting`:

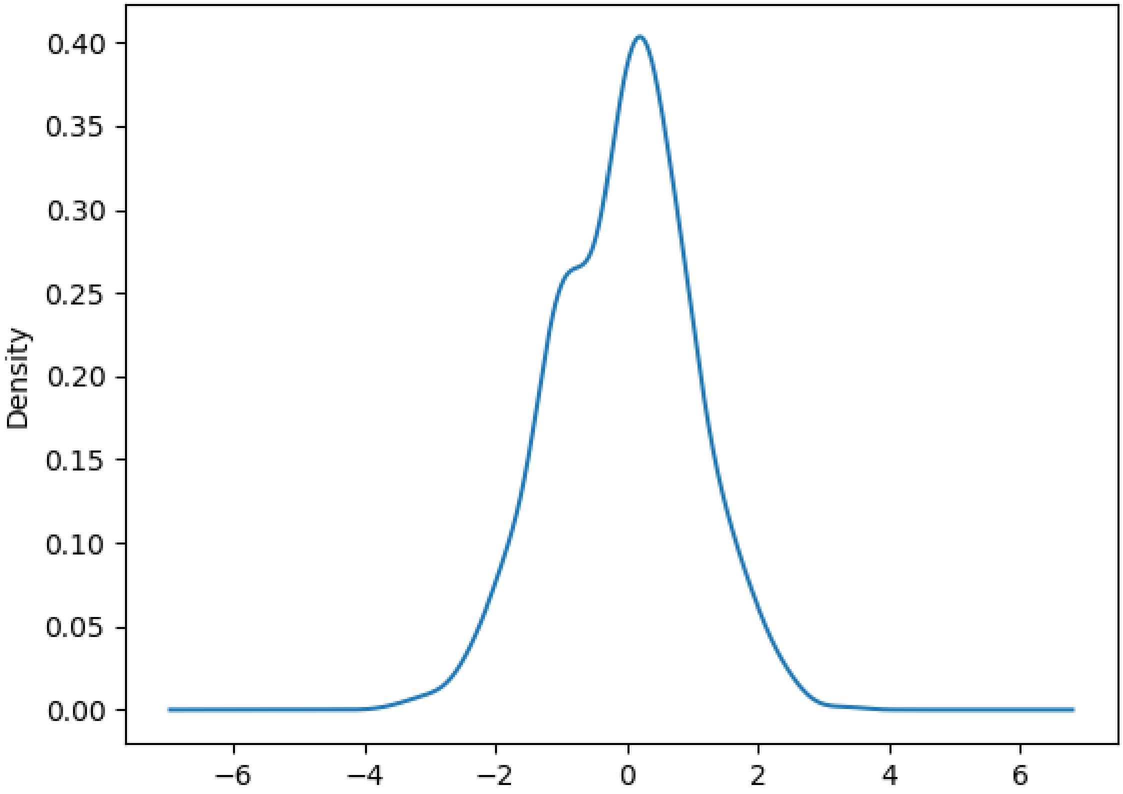
```
In [83]: from pandas.plotting import scatter_matrix

In [84]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [85]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
Out[85]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1258eb5d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1258fbc50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38bf6fd0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3e94ecd0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c3e97f510>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38ea6d10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3e7d1550>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3ee62d50>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c3ee898d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3dea1290>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c39ecf5d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38e3bdd0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c3a497610>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38683e10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38d1b650>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3e14ee50>]],
      dtype=object)
```



Density plot

You can create density plots using the `Series.plot.kde()` and `DataFrame.plot.kde()` methods.

```
In [86]: ser = pd.Series(np.random.randn(1000))

In [87]: ser.plot.kde()
Out[87]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3beb8450>
```

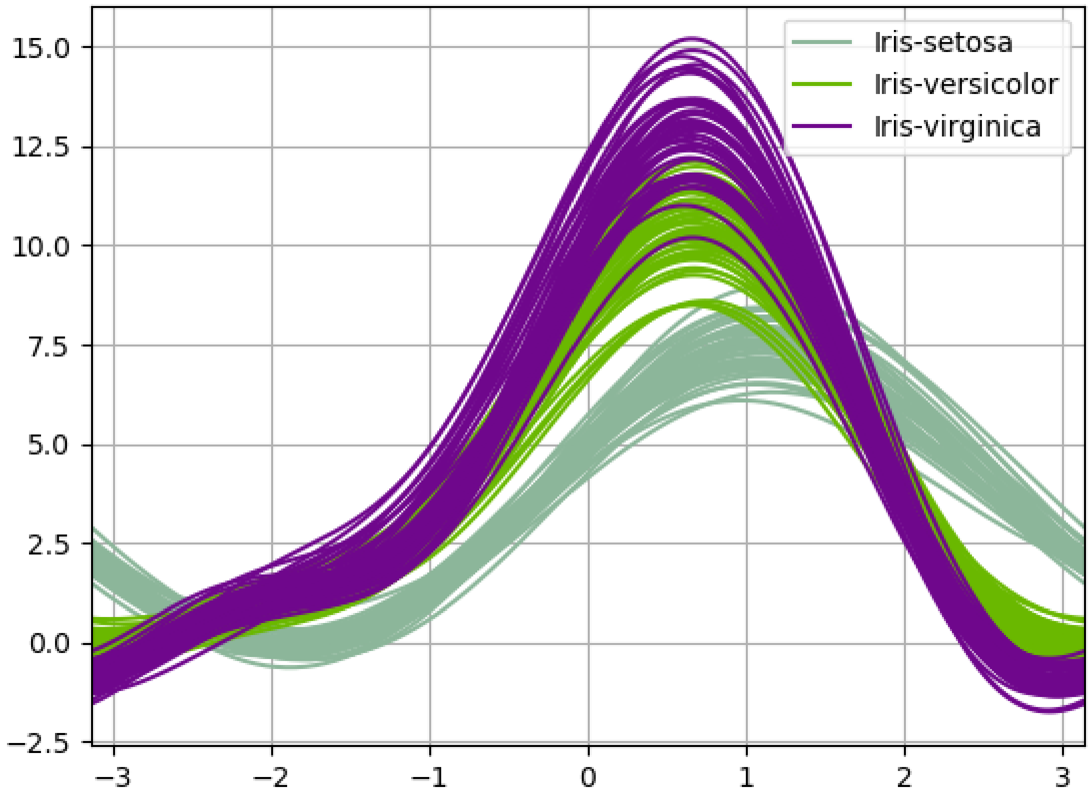


Andrews curves

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, see the Wikipedia entry for more information. 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.

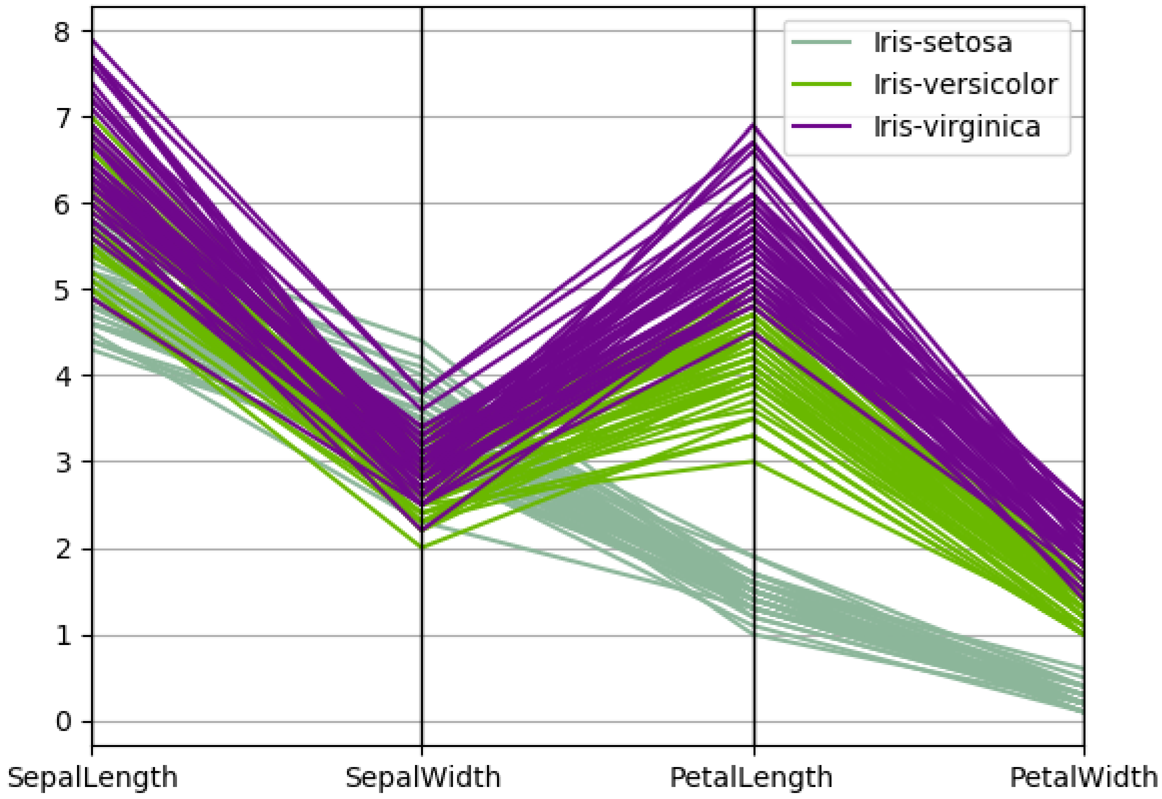
```
In [88]: from pandas.plotting import andrews_curves

In [89]: data = pd.read_csv('data/iris.data')

In [90]: plt.figure()
Out[90]: <Figure size 640x480 with 0 Axes>

In [91]: andrews_curves(data, 'Name')
Out[91]: <matplotlib.axes._subplots.AxesSubplot at 0x1c39b21c10>
```



Parallel coordinates

Parallel coordinates is a plotting technique for plotting multivariate data, see the Wikipedia entry for an introduction. Parallel coordinates 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.

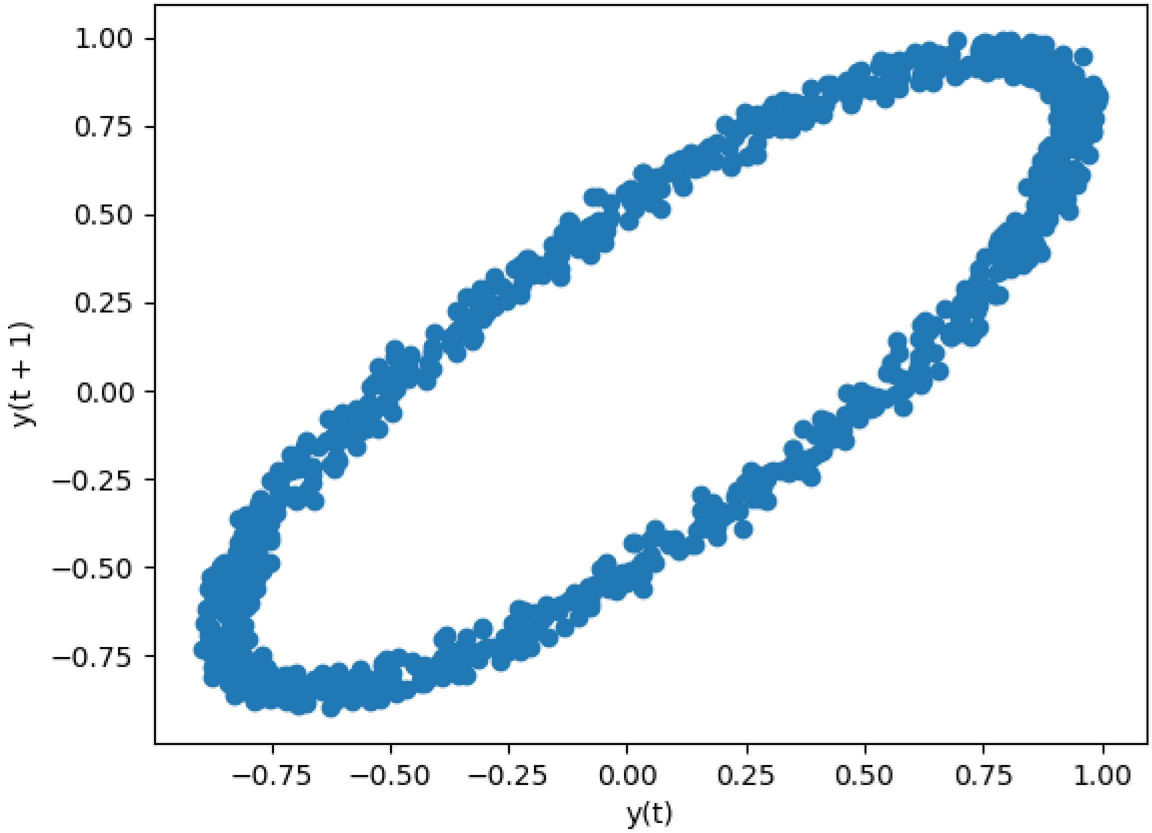
```
In [92]: from pandas.plotting import parallel_coordinates

In [93]: data = pd.read_csv('data/iris.data')

In [94]: plt.figure()
Out[94]: <Figure size 640x480 with 0 Axes>

In [95]: parallel_coordinates(data, 'Name')
Out[95]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3bd54bd0>
```



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. The `lag` argument may be passed, and when `lag=1` the plot is essentially `data[:-1]` vs. `data[1:]`.

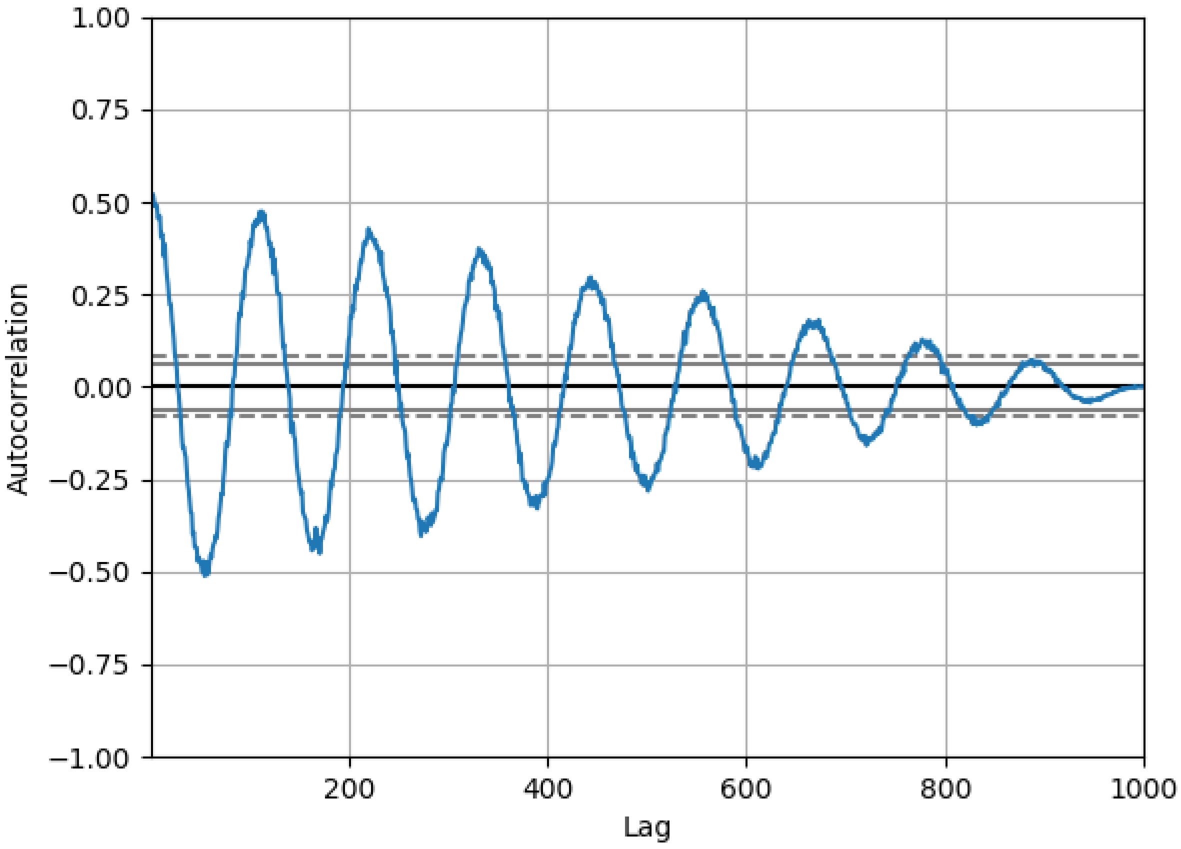
```
In [96]: from pandas.plotting import lag_plot

In [97]: plt.figure()
Out[97]: <Figure size 640x480 with 0 Axes>

In [98]: spacing = np.linspace(-99 * np.pi, 99 * np.pi, num=1000)

In [99]: data = pd.Series(0.1 * np.random.rand(1000) + 0.9 * np.sin(spacing))

In [100]: lag_plot(data)
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3efd8350>
```



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. See the Wikipedia entry for more about autocorrelation plots.

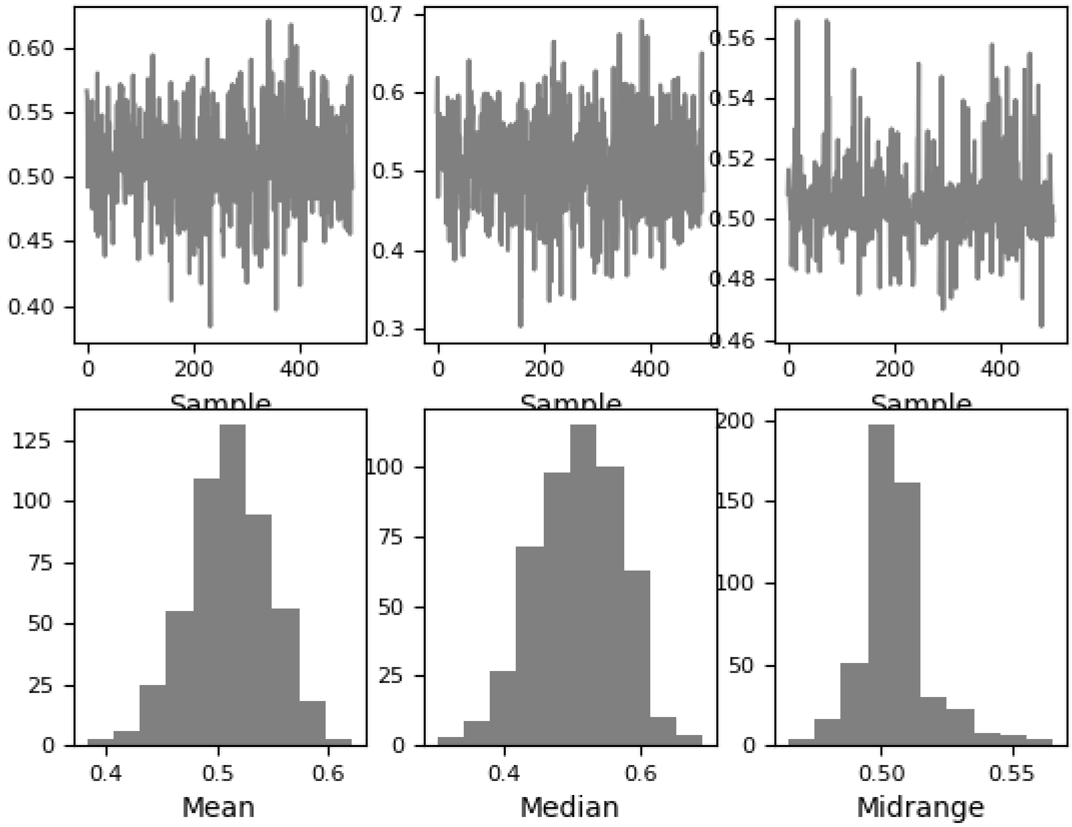
```
In [101]: from pandas.plotting import autocorrelation_plot

In [102]: plt.figure()
Out[102]: <Figure size 640x480 with 0 Axes>

In [103]: spacing = np.linspace(-9 * np.pi, 9 * np.pi, num=1000)

In [104]: data = pd.Series(0.7 * np.random.rand(1000) + 0.3 * np.sin(spacing))

In [105]: autocorrelation_plot(data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x1c39af5c10>
```



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 [106]: from pandas.plotting import bootstrap_plot

In [107]: data = pd.Series(np.random.rand(1000))

In [108]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[108]: <Figure size 640x480 with 6 Axes>
```

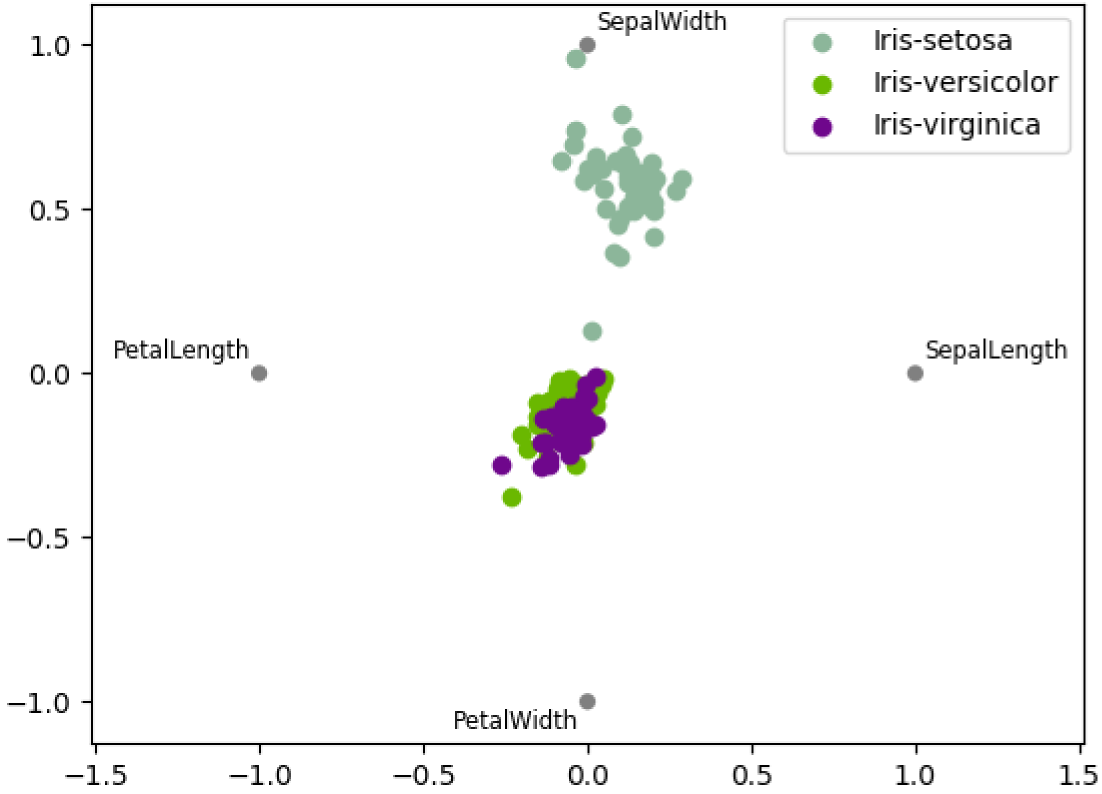


RadViz

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. See the R package Radviz for more information.

Note: The Iris dataset is available here.

```
In [109]: from pandas.plotting import radviz

In [110]: data = pd.read_csv('data/iris.data')

In [111]: plt.figure()
Out[111]: <Figure size 640x480 with 0 Axes>

In [112]: radviz(data, 'Name')
Out[112]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3e9f0110>
```



4.10.5 Plot Formatting

Setting the plot style

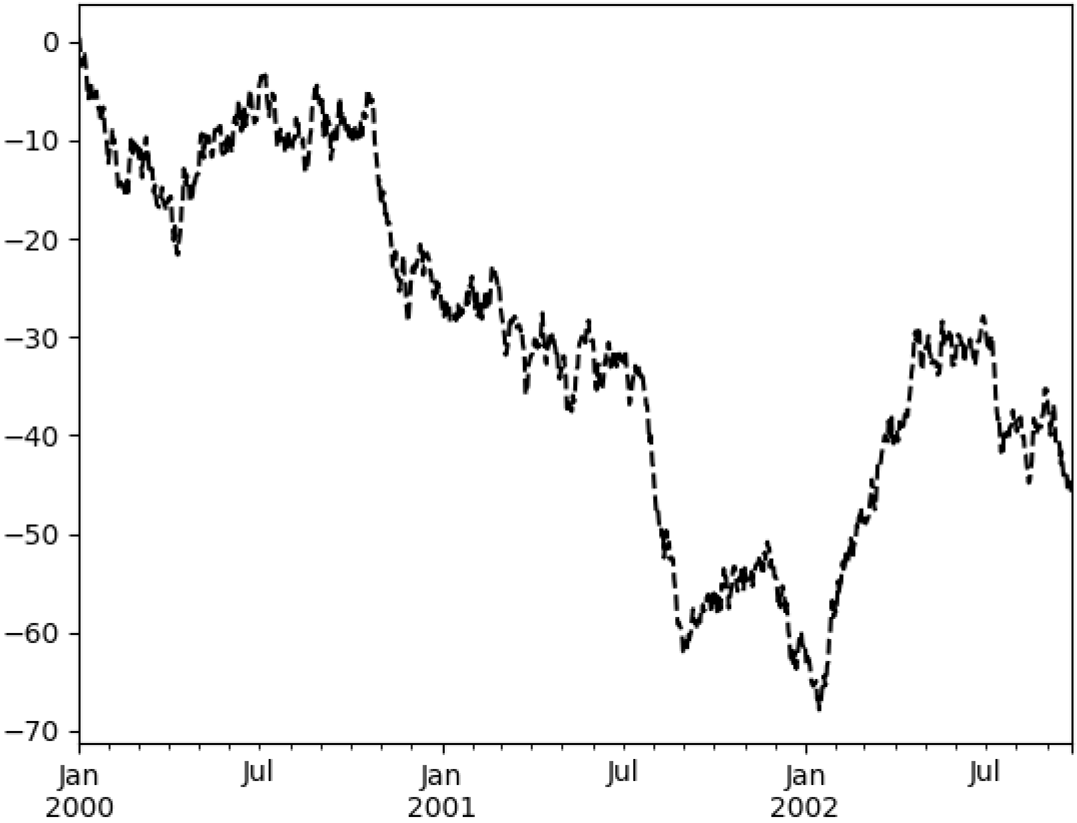
From version 1.5 and up, matplotlib offers a range of pre-configured plotting styles. Setting the style can be used to easily give plots the general look that you want. Setting the style is as easy as calling `matplotlib.style.use(my_plot_style)` before creating your plot. For example you could write `matplotlib.style.use('ggplot')` for ggplot-style plots.

You can see the various available style names at `matplotlib.style.available` and its very easy to try them out.

General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [113]: plt.figure();

In [114]: ts.plot(style='k--', label='Series');
```

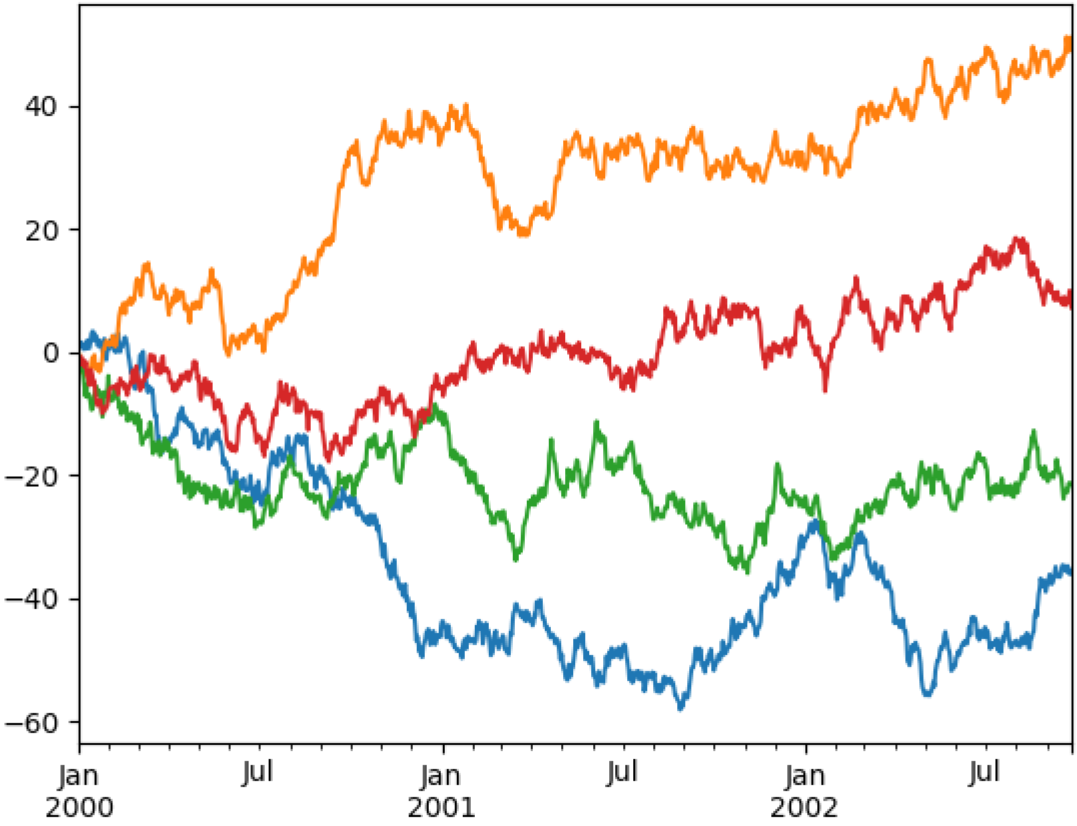


For each kind of plot (e.g. *line*, *bar*, *scatter*) any additional arguments keywords are passed along to the corresponding matplotlib function (`ax.plot()`, `ax.bar()`, `ax.scatter()`). These can be used to control additional styling, beyond what pandas provides.

Controlling the legend

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

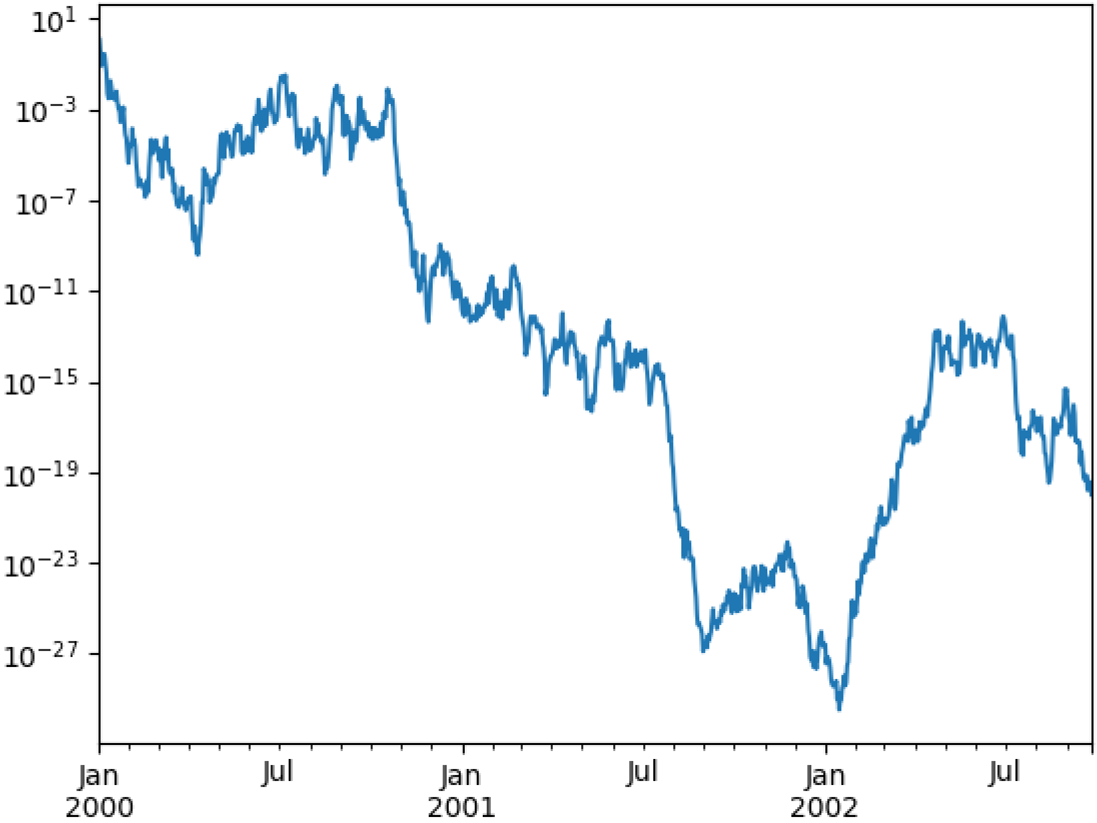
```
In [115]: df = pd.DataFrame(np.random.randn(1000, 4),
   .....:                   index=ts.index, columns=list('ABCD'))
   .....:

In [116]: df = df.cumsum()

In [117]: df.plot(legend=False)
Out[117]: <matplotlib.axes._subplots.AxesSubplot at 0x1c386d8ed0>
```



Scales

You may pass `logy` to get a log-scale Y axis.

```
In [118]: ts = pd.Series(np.random.randn(1000),
   .....:                index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [119]: ts = np.exp(ts.cumsum())

In [120]: ts.plot(logy=True)
Out[120]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3e544ad0>
```

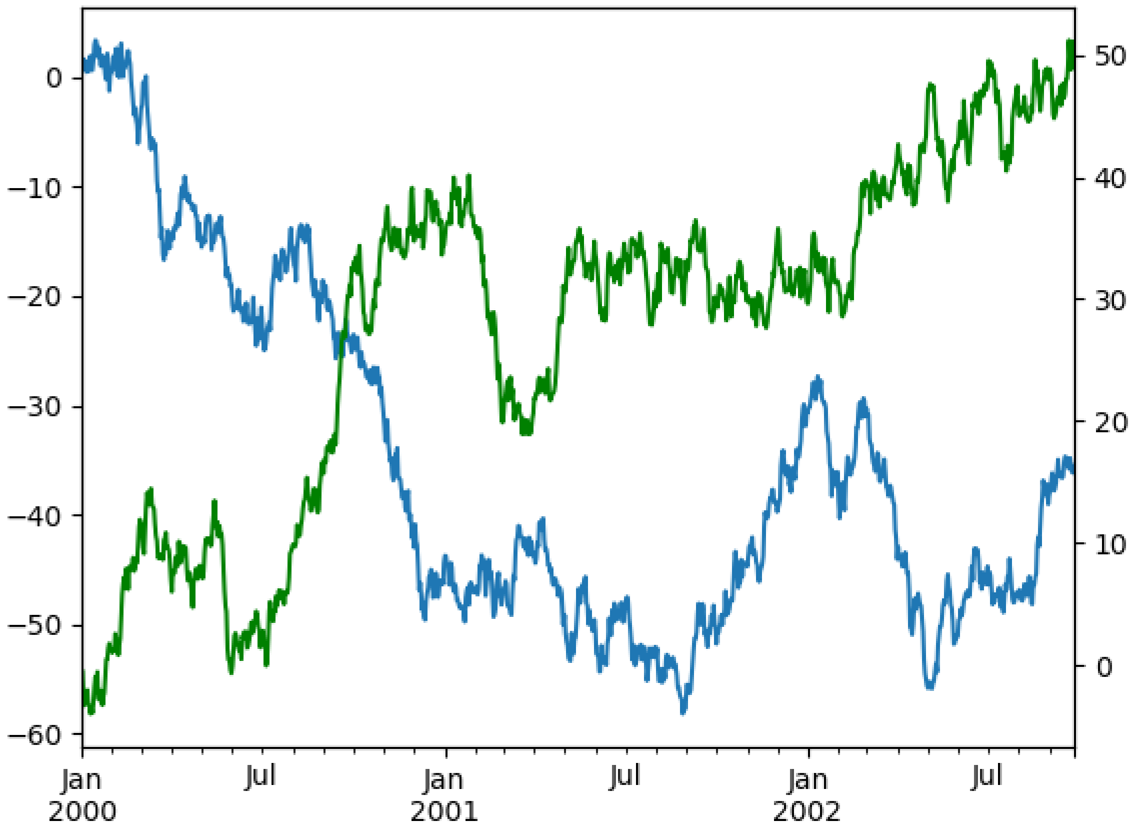


See also the `logx` and `loglog` keyword arguments.

Plotting on a secondary y-axis

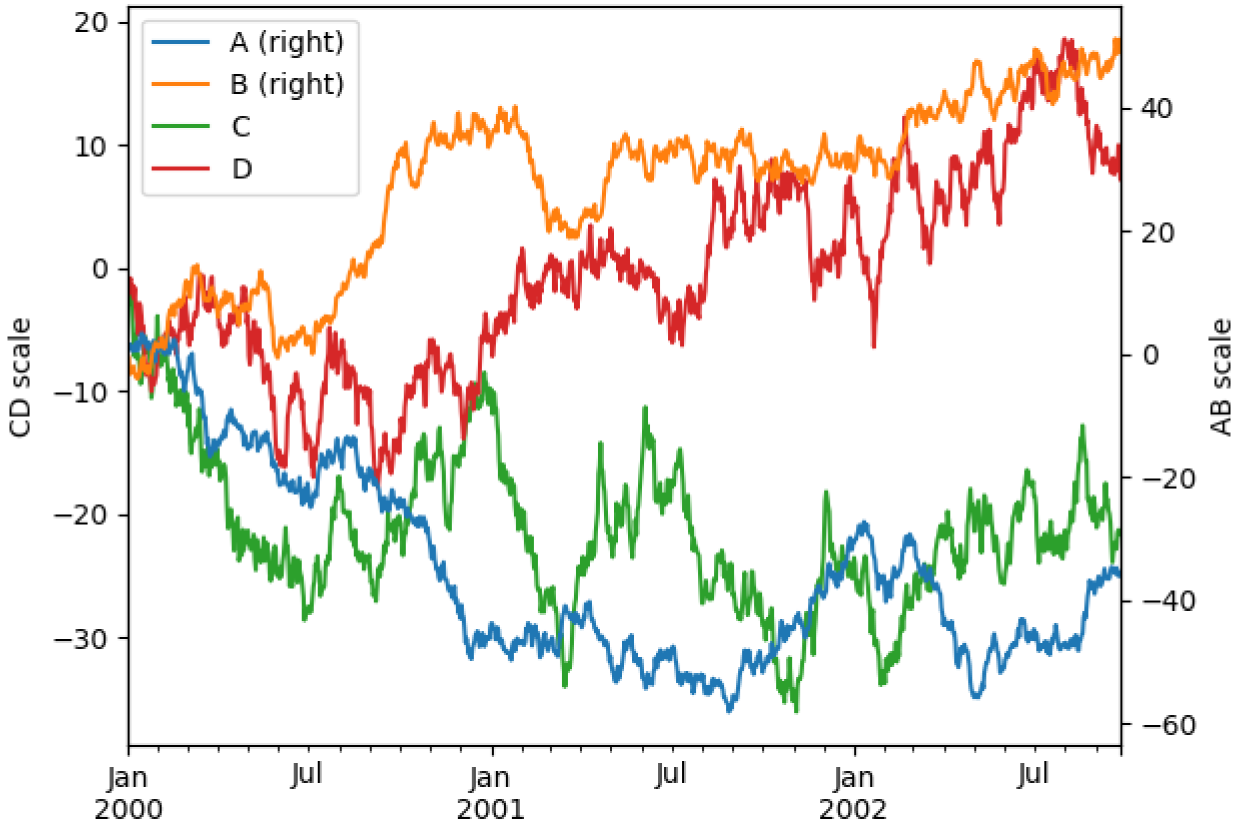
To plot data on a secondary y-axis, use the `secondary_y` keyword:

```
In [121]: df.A.plot()
Out[121]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3ec4f890>

In [122]: df.B.plot(secondary_y=True, style='g')
Out[122]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3ec66150>
```



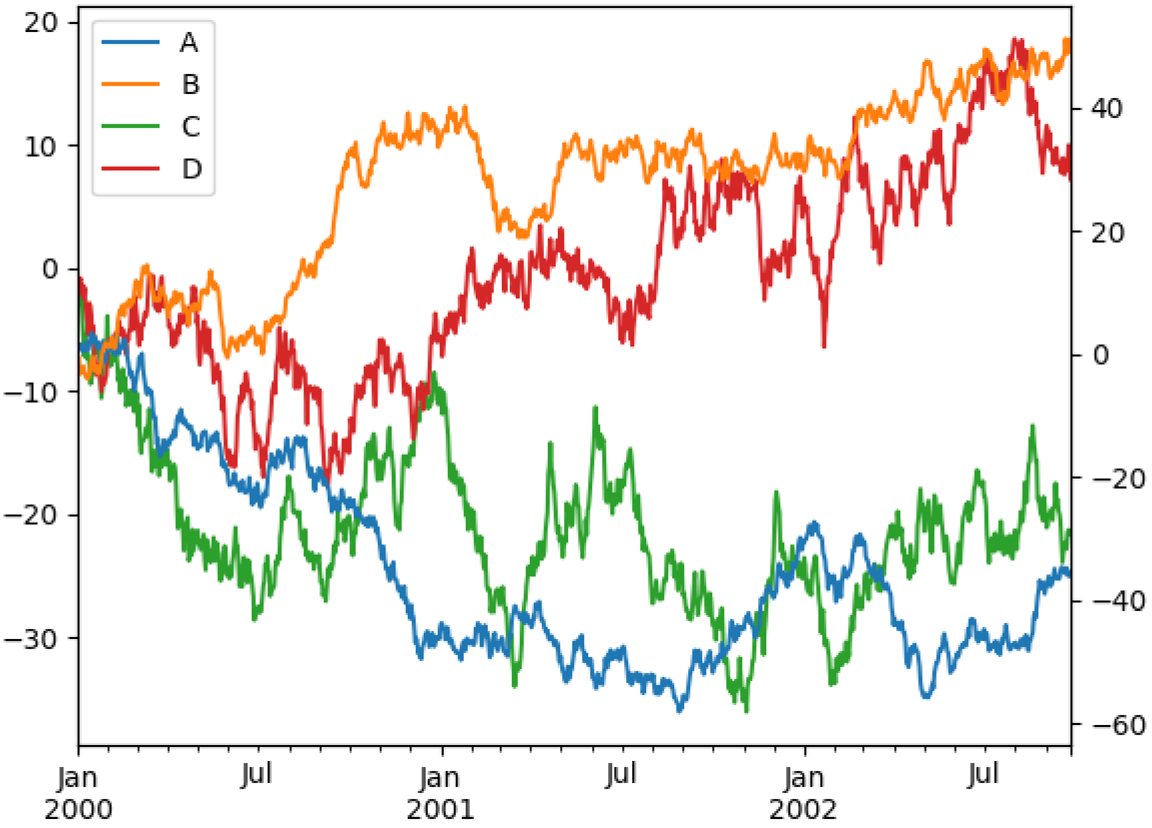
To plot some columns in a `DataFrame`, give the column names to the `secondary_y` keyword:

```
In [123]: plt.figure()
Out[123]: <Figure size 640x480 with 0 Axes>

In [124]: ax = df.plot(secondary_y=['A', 'B'])

In [125]: ax.set_ylabel('CD scale')
Out[125]: Text(0, 0.5, 'CD scale')

In [126]: ax.right_ax.set_ylabel('AB scale')
Out[126]: Text(0, 0.5, 'AB scale')
```



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 [127]: plt.figure()
Out[127]: <Figure size 640x480 with 0 Axes>

In [128]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[128]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3e18f850>
```

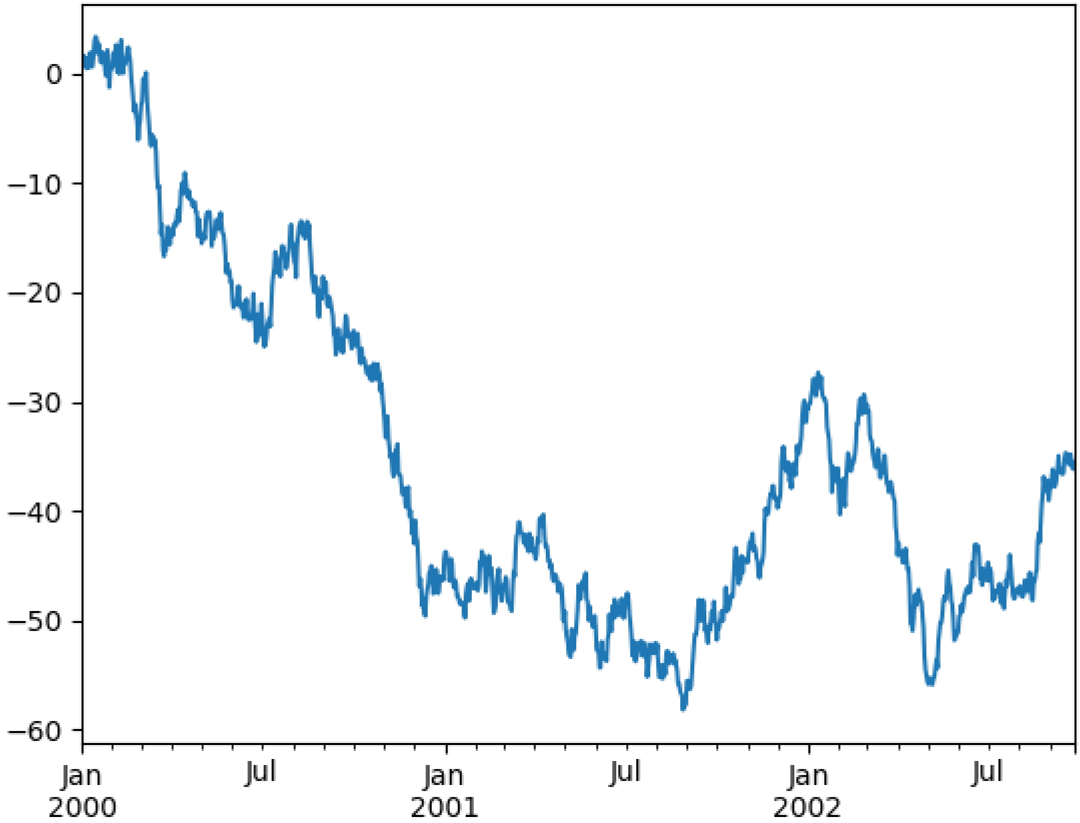


Suppressing tick resolution adjustment

pandas includes automatic tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created `twinx`), you can choose to suppress this behavior for alignment purposes.

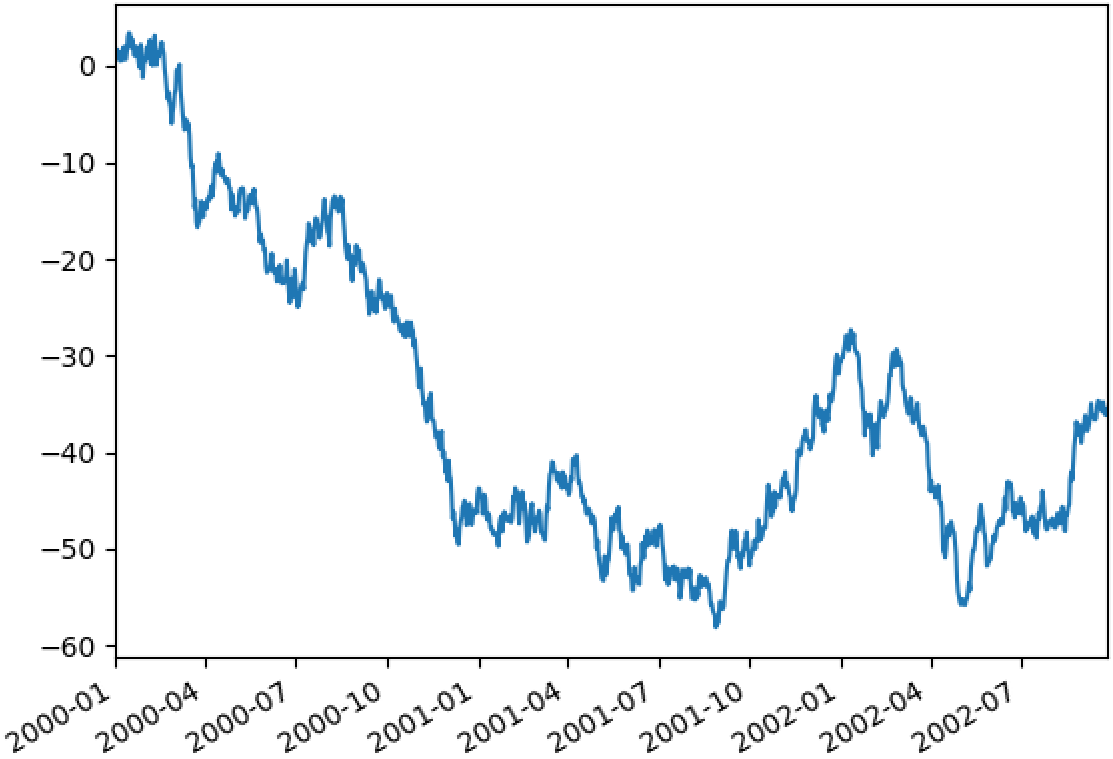
Here is the default behavior, notice how the x-axis tick labeling is performed:

```
In [129]: plt.figure()
Out[129]: <Figure size 640x480 with 0 Axes>

In [130]: df.A.plot()
Out[130]: <matplotlib.axes._subplots.AxesSubplot at 0x1c39659450>
```



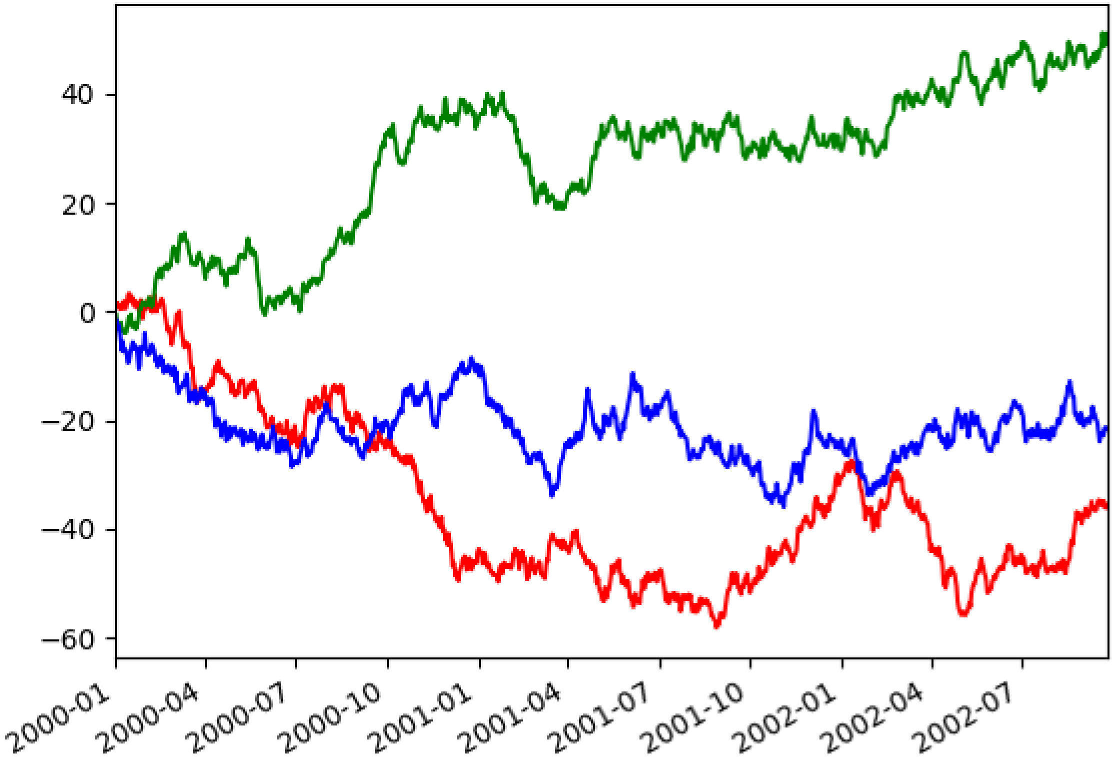
Using the x_compat parameter, you can suppress this behavior:

```
In [131]: plt.figure()
Out[131]: <Figure size 640x480 with 0 Axes>

In [132]: df.A.plot(x_compat=True)
Out[132]: <matplotlib.axes._subplots.AxesSubplot at 0x1a30f11d90>
```



If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plotting.plot_params` can be used in a *with statement*:

```
In [133]: plt.figure()
Out[133]: <Figure size 640x480 with 0 Axes>

In [134]: with pd.plotting.plot_params.use('x_compat', True):
   .....:     df.A.plot(color='r')
   .....:     df.B.plot(color='g')
   .....:     df.C.plot(color='b')
   .....:
```



Automatic date tick adjustment

New in version 0.20.0.

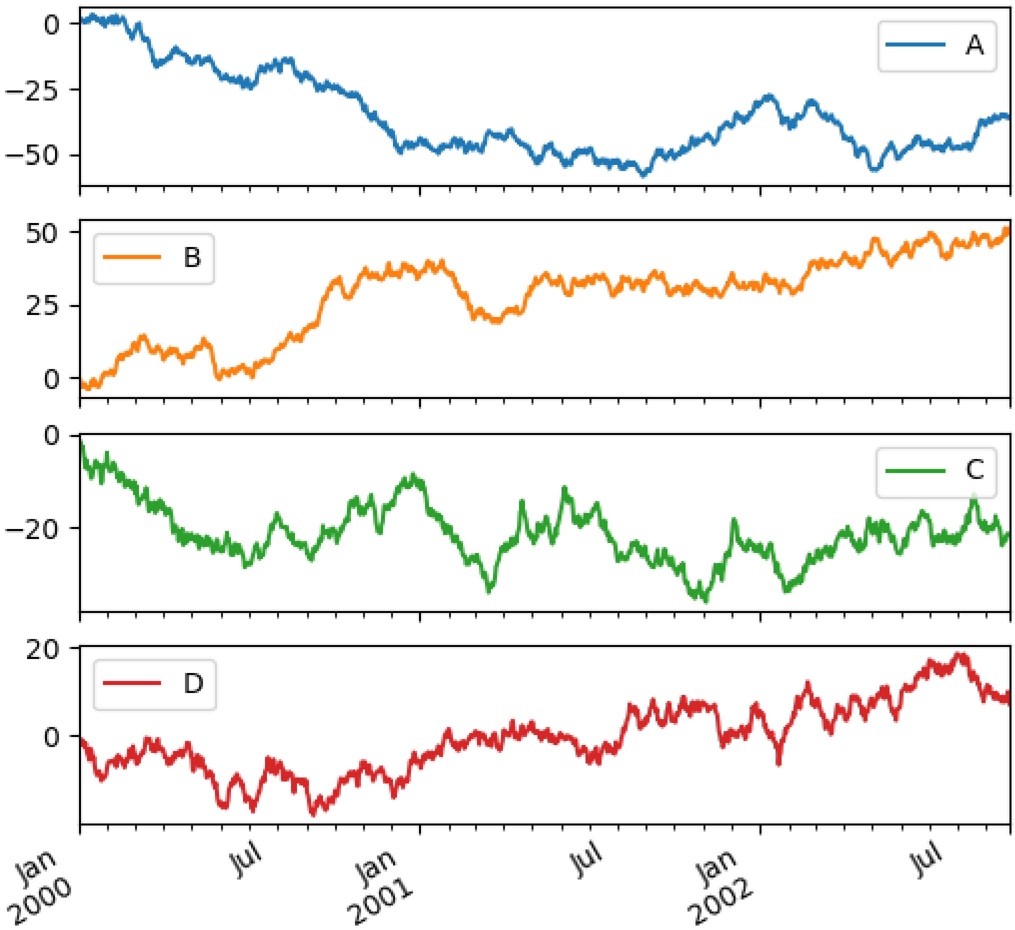
`TimedeltaIndex` now uses the native matplotlib tick locator methods, it is useful to call the automatic date tick adjustment from matplotlib for figures whose ticklabels overlap.

See the `autofmt_xdate` method and the matplotlib documentation for more.

Subplots

Each `Series` in a `DataFrame` can be plotted on a different axis with the `subplots` keyword:

```
In [135]: df.plot(subplots=True, figsize=(6, 6));
```

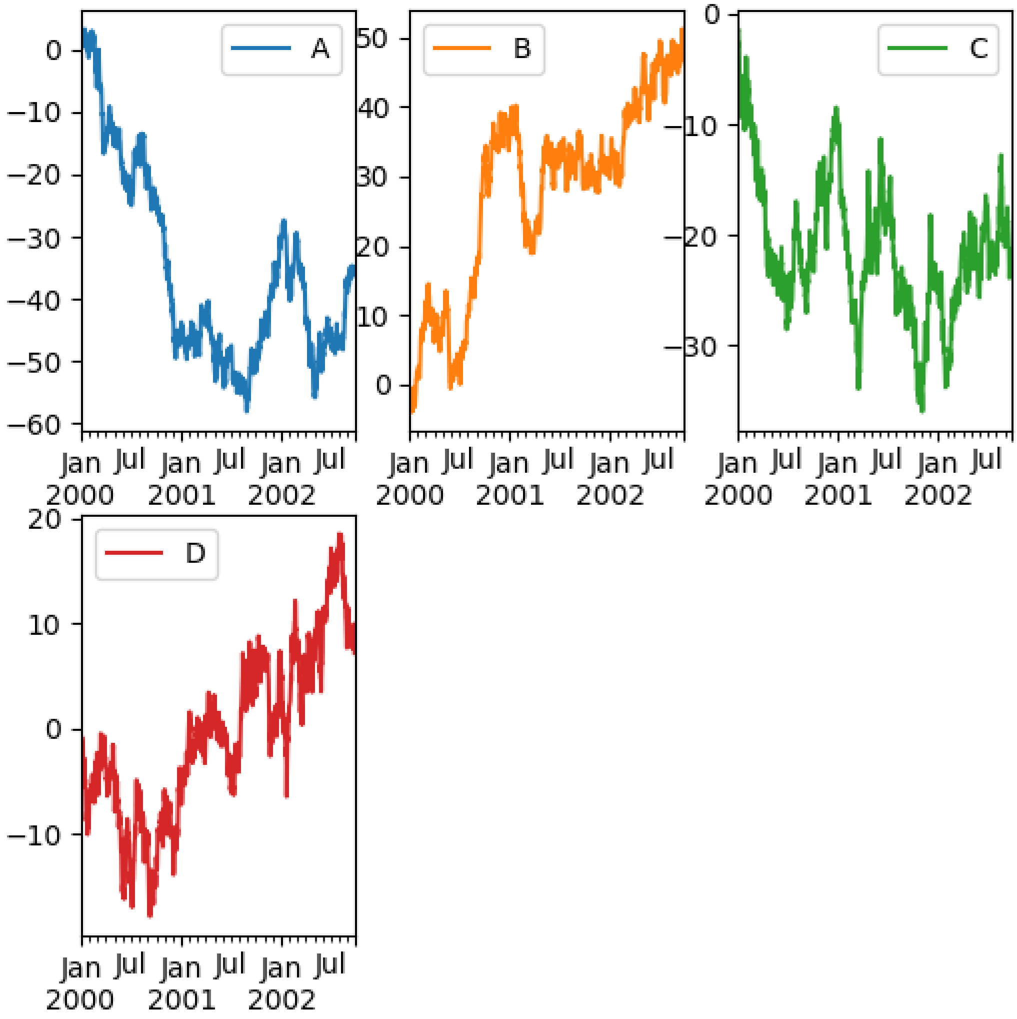


Using layout and targeting multiple axes

The layout of subplots can be specified by the `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If the input is invalid, a `ValueError` will be raised.

The number of axes which can be contained by rows x columns specified by `layout` must be larger than the number of required subplots. If layout can contain more axes than required, blank axes are not drawn. Similar to a NumPy arrays `reshape` method, you can use `-1` for one dimension to automatically calculate the number of rows or columns needed, given the other.

```
In [136]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```



The above example is identical to using:

```
In [137]: df.plot(subplots=True, layout=(2, -1), figsize=(6, 6), sharex=False);
```

The required number of columns (3) is inferred from the number of series to plot and the given number of rows (2).

You can pass multiple axes created beforehand as list-like via `ax` keyword. This allows more complicated layouts. The passed axes must be the same number as the subplots being drawn.

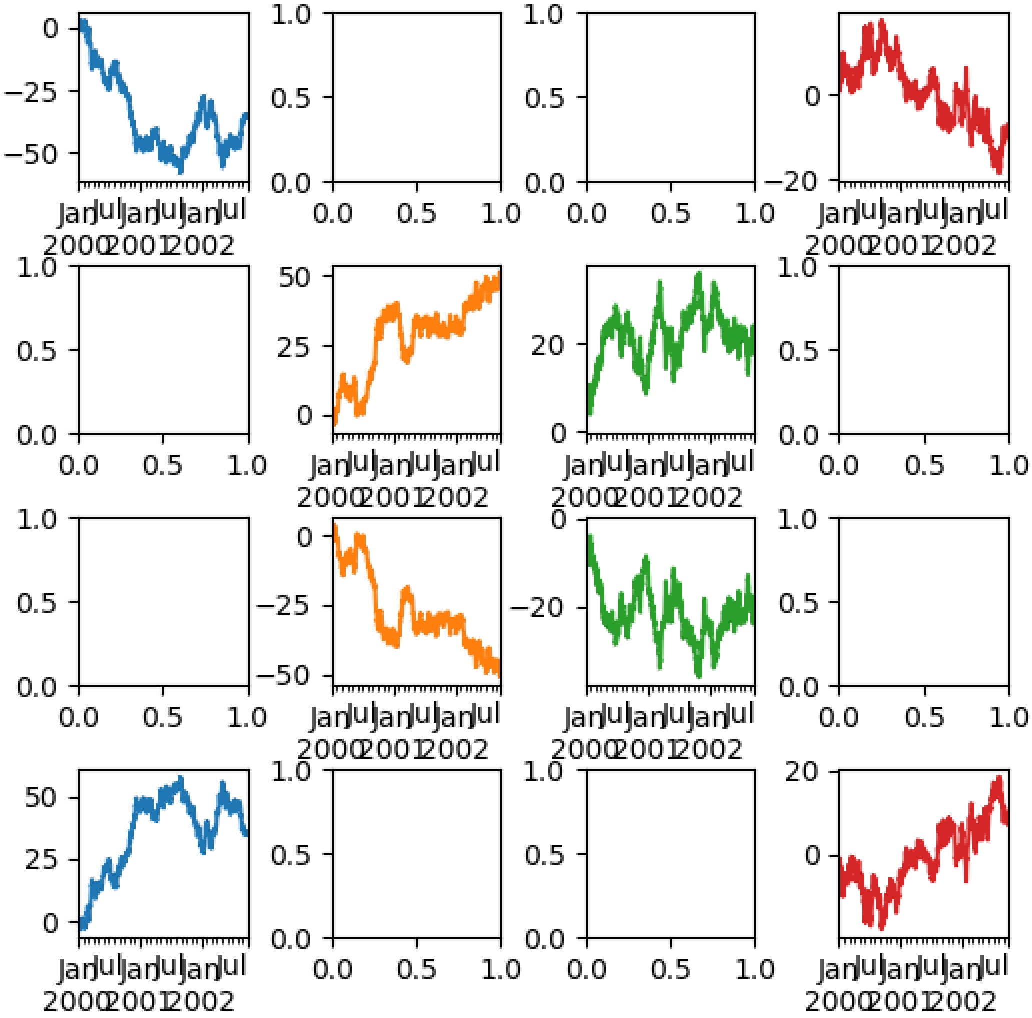
When multiple axes are passed via the `ax` keyword, `layout`, `sharex` and `sharey` keywords dont affect to the output. You should explicitly pass `sharex=False` and `sharey=False`, otherwise you will see a warning.

```
In [138]: fig, axes = plt.subplots(4, 4, figsize=(6, 6))

In [139]: plt.subplots_adjust(wspace=0.5, hspace=0.5)

In [140]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
```

(continues on next page)

(continued from previous page)

```
In [141]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]

In [142]: df.plot(subplots=True, ax=target1, legend=False, sharex=False,␣
↪sharey=False);

In [143]: (-df).plot(subplots=True, ax=target2, legend=False,
   .....:             sharex=False, sharey=False);
   .....:
```



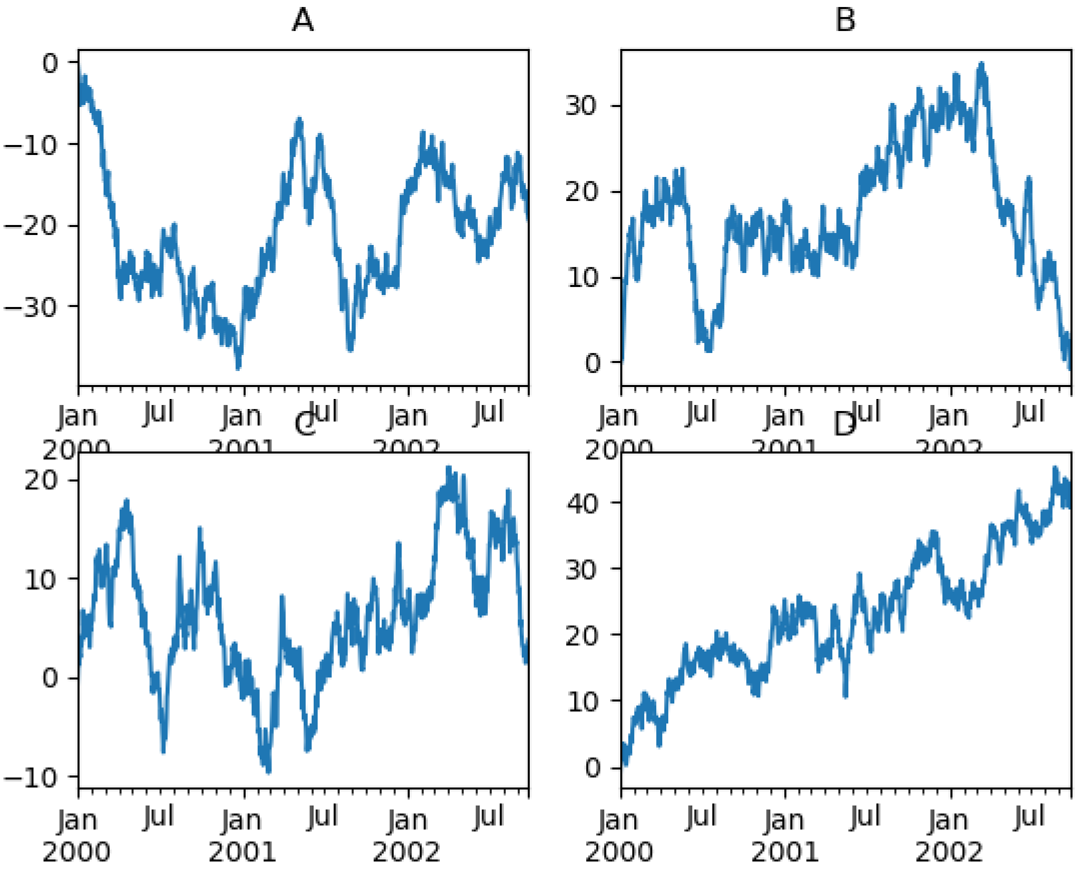
Another option is passing an `ax` argument to `Series.plot()` to plot on a particular axis:

```
In [144]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [145]: df['A'].plot(ax=axes[0, 0]);
```

(continues on next page)

(continued from previous page)

```
In [146]: axes[0, 0].set_title('A');

In [147]: df['B'].plot(ax=axes[0, 1]);

In [148]: axes[0, 1].set_title('B');

In [149]: df['C'].plot(ax=axes[1, 0]);

In [150]: axes[1, 0].set_title('C');

In [151]: df['D'].plot(ax=axes[1, 1]);

In [152]: axes[1, 1].set_title('D');
```



Plotting with error bars

Plotting with error bars is supported in `DataFrame.plot()` and `Series.plot()`.

Horizontal and vertical error bars can be supplied to the `xerr` and `yerr` keyword arguments to `plot()`. The error values can be specified using a variety of formats:

- As a `DataFrame` or `dict` of errors with column names matching the `columns` attribute of the plotting

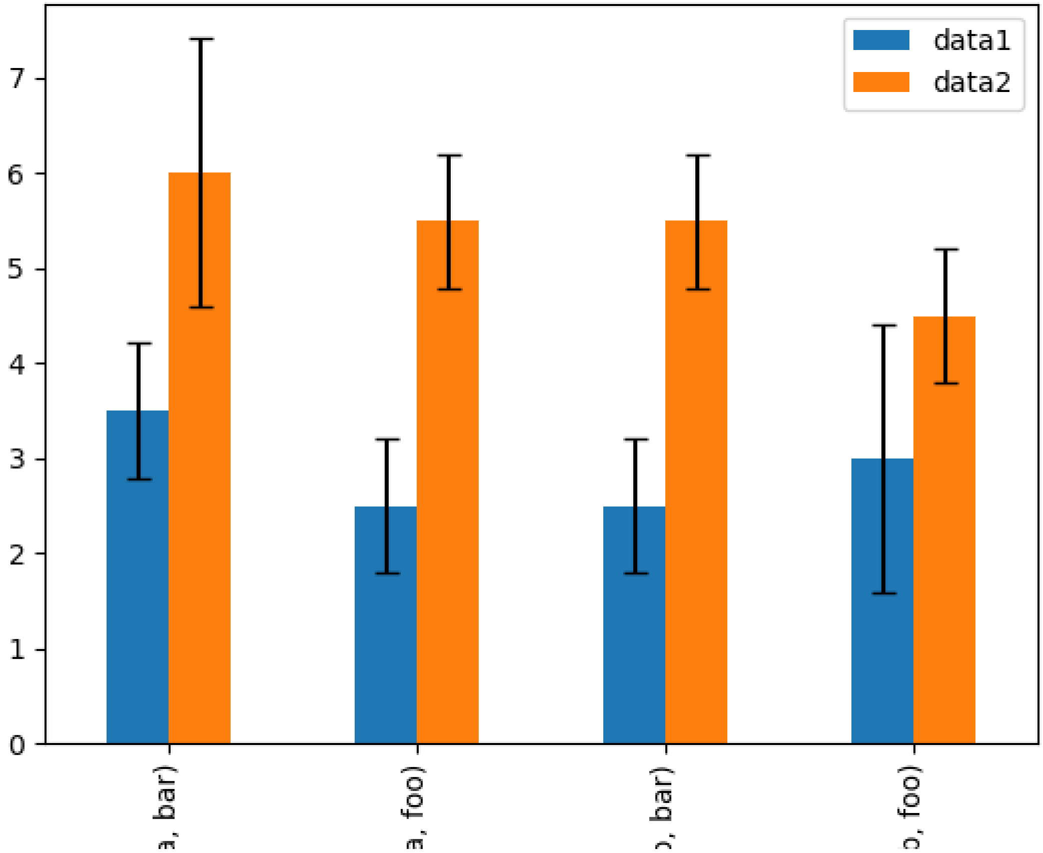
DataFrame or matching the name attribute of the Series.

- As a str indicating which of the columns of plotting DataFrame contain the error values.
- As raw values (list, tuple, or np.ndarray). Must be the same length as the plotting DataFrame/Series.

Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a M length Series, a Mx2 array should be provided indicating lower and upper (or left and right) errors. For a MxN DataFrame, asymmetrical errors should be in a Mx2xN array.

Here is an example of one way to easily plot group means with standard deviations from the raw data.

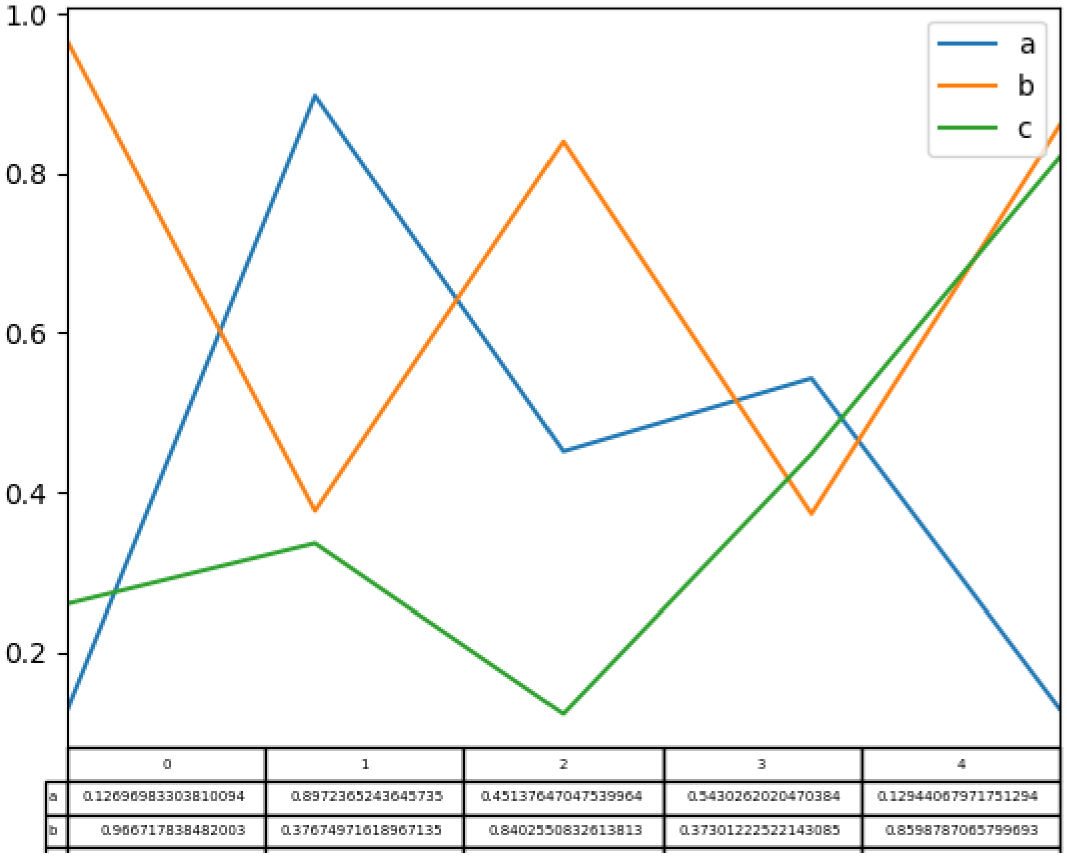
```
# Generate the data
In [153]: ix3 = pd.MultiIndex.from_arrays([
   .....:     ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
   .....:     ['foo', 'foo', 'bar', 'bar', 'foo', 'foo', 'bar', 'bar']],
   .....:     names=['letter', 'word'])
   .....:

In [154]: df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2],
   .....:                     'data2': [6, 5, 7, 5, 4, 5, 6, 5]}, index=ix3)
   .....:

# Group by index labels and take the means and standard deviations
# for each group
In [155]: gp3 = df3.groupby(level=('letter', 'word'))

In [156]: means = gp3.mean()

In [157]: errors = gp3.std()

In [158]: means
Out[158]:
             data1  data2
letter word
a      bar     3.5    6.0
       foo     2.5    5.5
b      bar     2.5    5.5
       foo     3.0    4.5

In [159]: errors
Out[159]:
                data1     data2
letter word
a      bar   0.707107  1.414214
       foo   0.707107  0.707107
b      bar   0.707107  0.707107
       foo   1.414214  0.707107

# Plot
In [160]: fig, ax = plt.subplots()

In [161]: means.plot.bar(yerr=errors, ax=ax, capsize=4)
Out[161]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4327bc50>
```



Plotting tables

Plotting with matplotlib table is now supported in `DataFrame.plot()` and `Series.plot()` with a `table` keyword. The `table` keyword can accept `bool`, `DataFrame` or `Series`. The simple way to draw a table is to specify `table=True`. Data will be transposed to meet matplotlibs default layout.

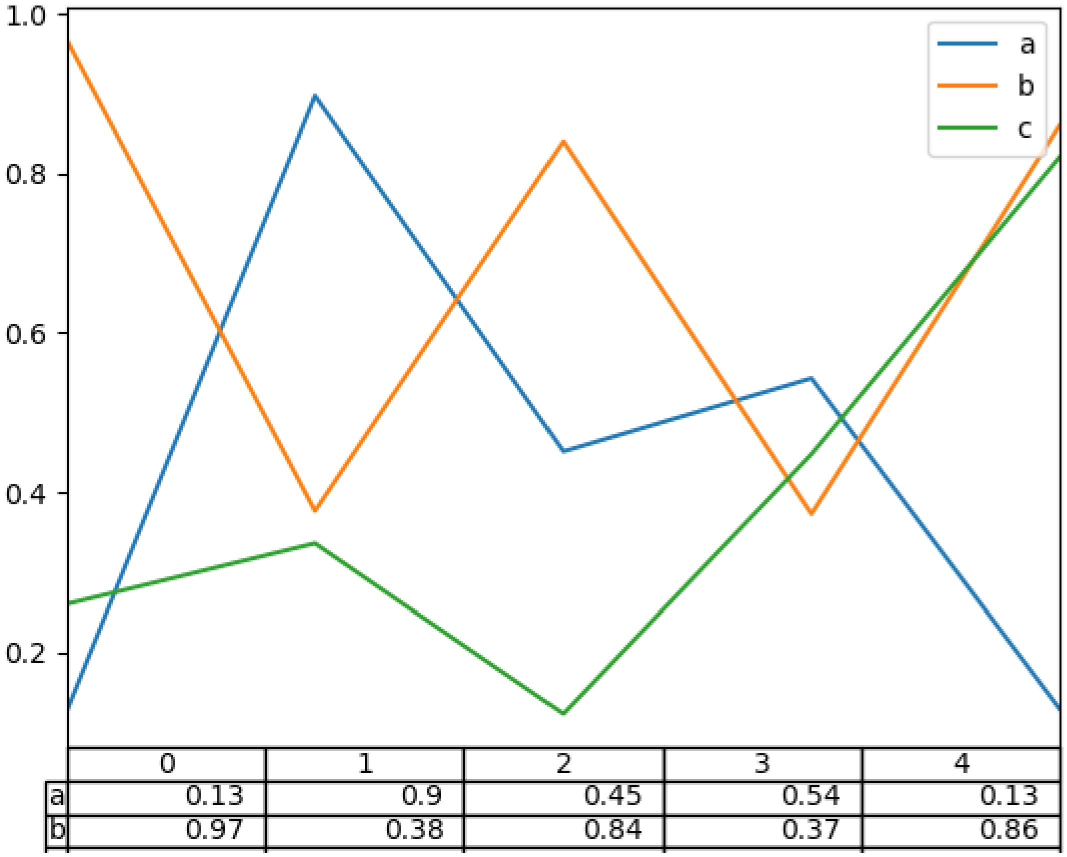
```
In [162]: fig, ax = plt.subplots(1, 1)

In [163]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

In [164]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [165]: df.plot(table=True, ax=ax)
Out[165]: <matplotlib.axes._subplots.AxesSubplot at 0x1c43469350>
```



	0	1	2	3	4
a	0.12696983303810094	0.8972365243645735	0.45137647047539964	0.5430262020470384	0.12944067971751294
b	0.966717838482003	0.37674971618967135	0.8402550832613813	0.37301222522143085	0.8598787065799693

Also, you can pass a different DataFrame or Series to the table keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as seen in the example below.

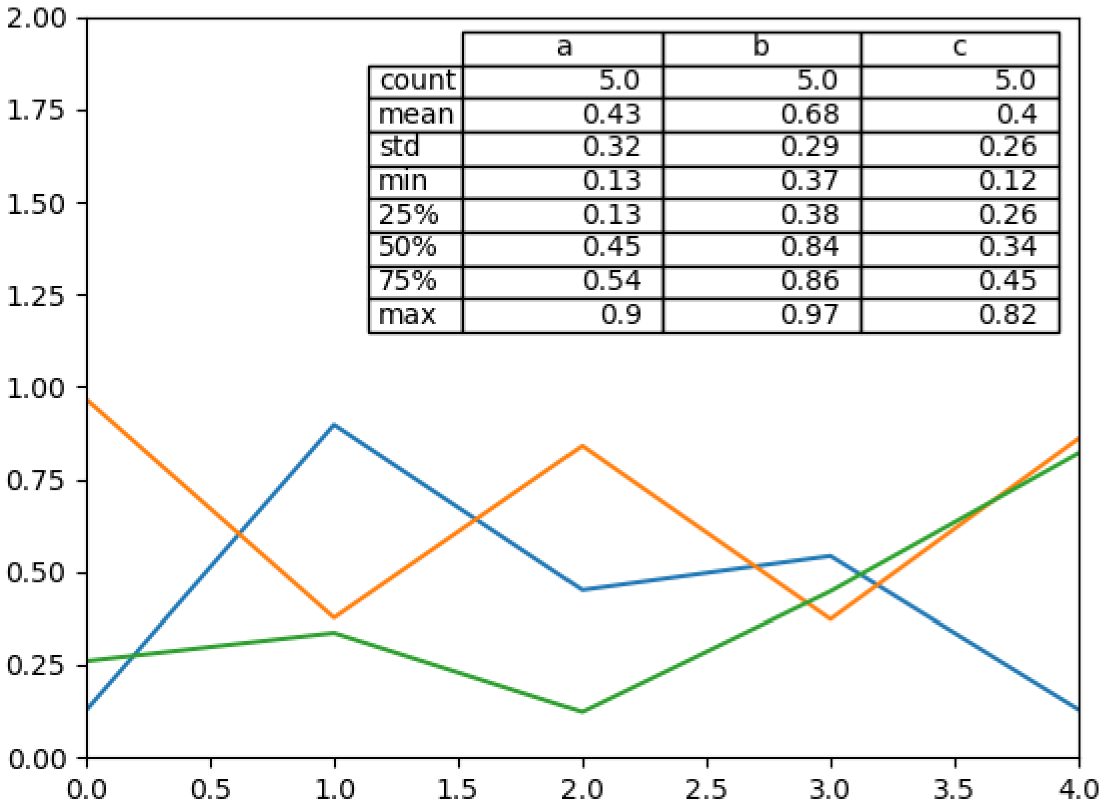
```
In [166]: fig, ax = plt.subplots(1, 1)

In [167]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [168]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[168]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4363e6d0>
```



	0	1	2	3	4
a	0.13	0.9	0.45	0.54	0.13
b	0.97	0.38	0.84	0.37	0.86

There also exists a helper function `pandas.plotting.table`, which creates a table from `DataFrame` or `Series`, and adds it to an `matplotlib.Axes` instance. This function can accept keywords which the matplotlib table has.

```
In [169]: from pandas.plotting import table

In [170]: fig, ax = plt.subplots(1, 1)

In [171]: table(ax, np.round(df.describe(), 2),
   .....:       loc='upper right', colWidths=[0.2, 0.2, 0.2])
   .....:
Out[171]: <matplotlib.table.Table at 0x1c43858fd0>

In [172]: df.plot(ax=ax, ylim=(0, 2), legend=None)
Out[172]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4381c090>
```



	a	b	c
count	5.0	5.0	5.0
mean	0.43	0.68	0.4
std	0.32	0.29	0.26
min	0.13	0.37	0.12
25%	0.13	0.38	0.26
50%	0.45	0.84	0.34
75%	0.54	0.86	0.45
max	0.9	0.97	0.82

Note: You can get table instances on the axes using `axes.tables` property for further decorations. See the matplotlib table documentation for more.

Colormaps

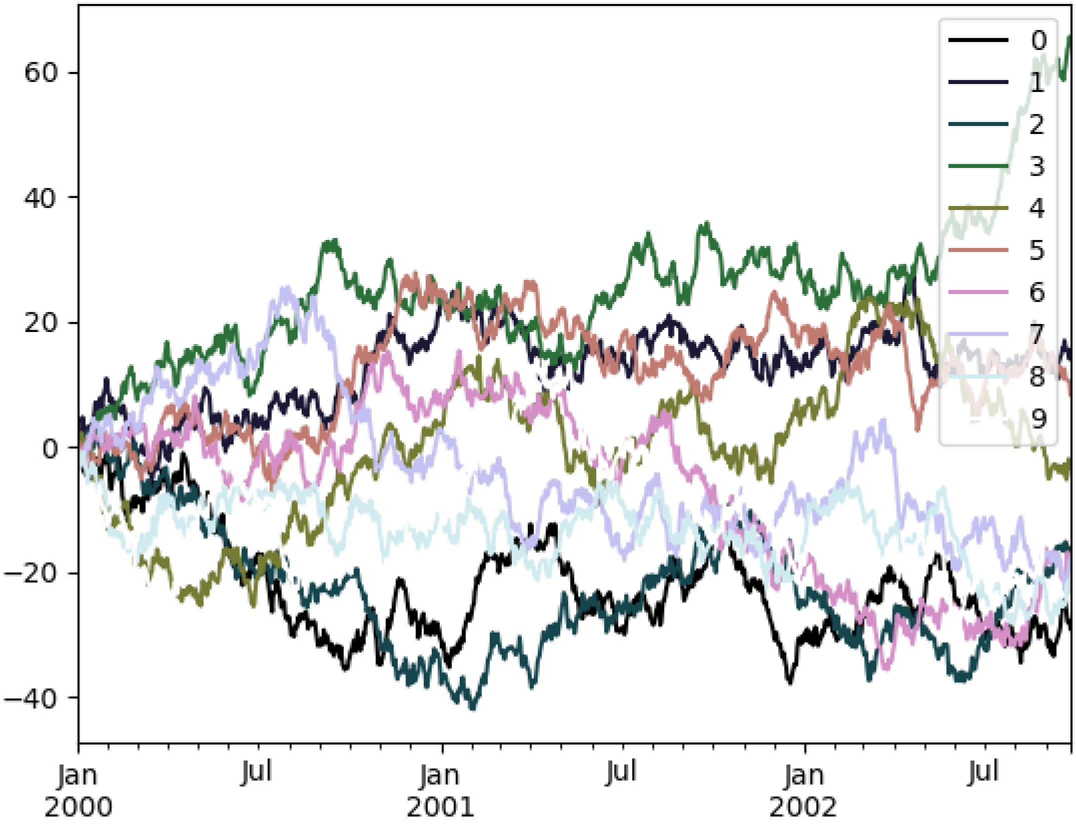
A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, `DataFrame` plotting supports the use of the `colormap` argument, which accepts either a Matplotlib colormap or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available here.

As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the `DataFrame`. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

To use the cubehelix colormap, we can pass `colormap='cubehelix'`.

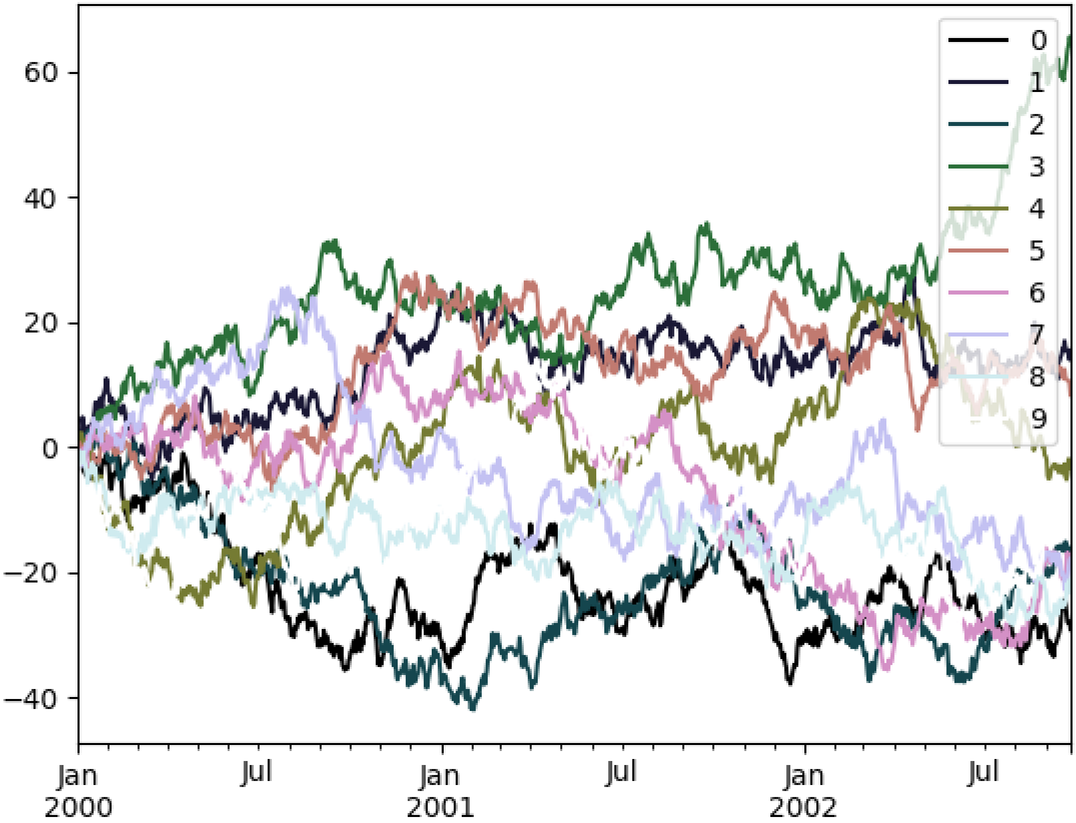
```
In [173]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)

In [174]: df = df.cumsum()

In [175]: plt.figure()
Out[175]: <Figure size 640x480 with 0 Axes>

In [176]: df.plot(colormap='cubehelix')
```

```
Out[176]: <matplotlib.axes._subplots.AxesSubplot at 0x1c43a29e50>
```



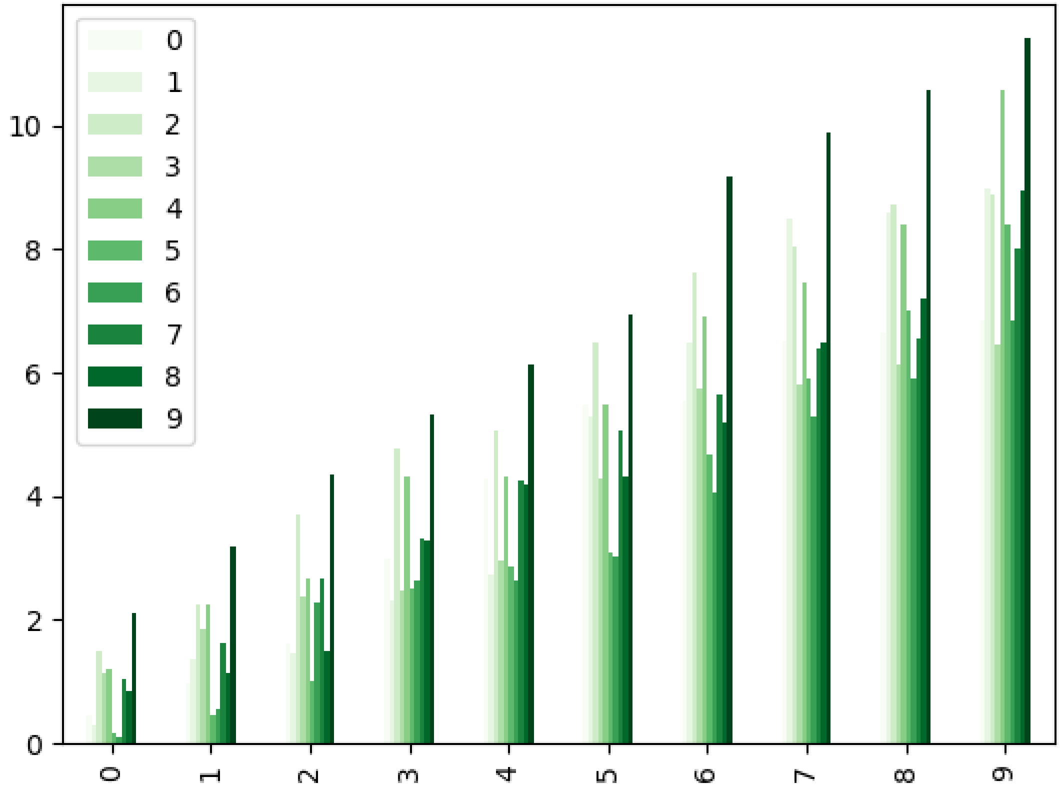
Alternatively, we can pass the colormap itself:

```
In [177]: from matplotlib import cm

In [178]: plt.figure()
Out[178]: <Figure size 640x480 with 0 Axes>

In [179]: df.plot(colormap=cm.cubehelix)
Out[179]: <matplotlib.axes._subplots.AxesSubplot at 0x1c43d3d210>
```



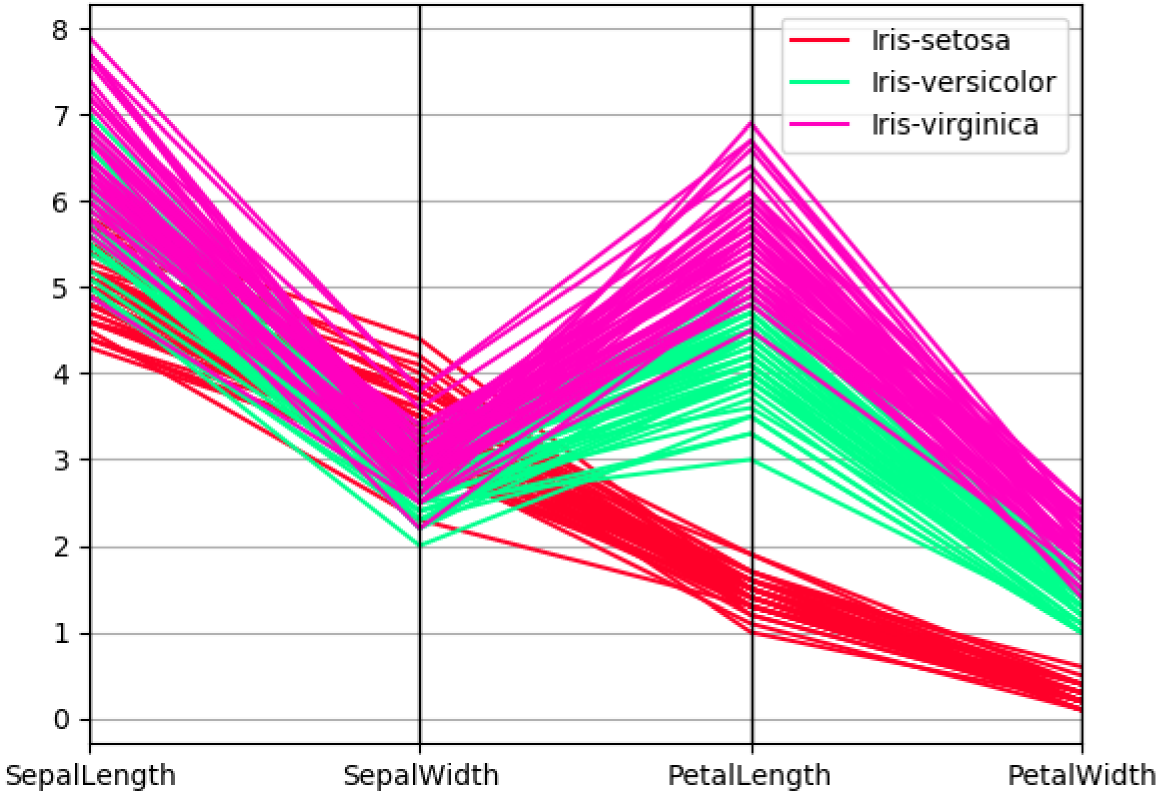
Colormaps can also be used other plot types, like bar charts:

```
In [180]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)

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

In [182]: plt.figure()
Out[182]: <Figure size 640x480 with 0 Axes>

In [183]: dd.plot.bar(colormap='Greens')
Out[183]: <matplotlib.axes._subplots.AxesSubplot at 0x1c44056f10>
```



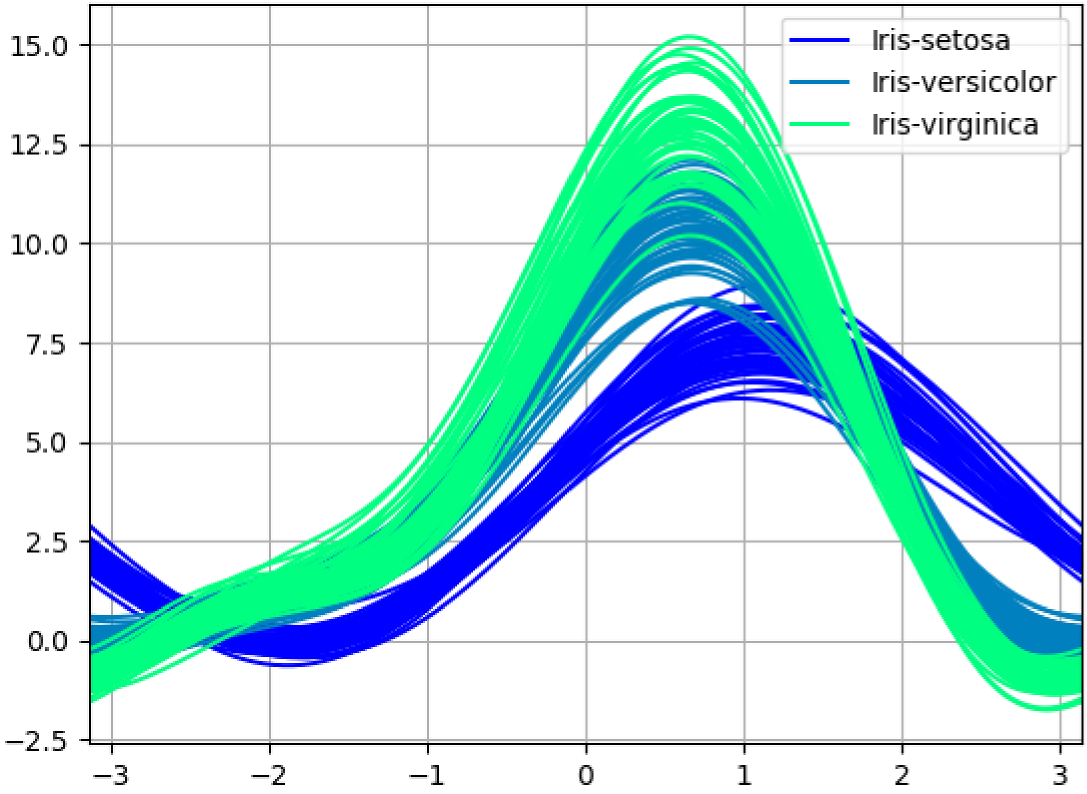
Parallel coordinates charts:

```
In [184]: plt.figure()
Out[184]: <Figure size 640x480 with 0 Axes>

In [185]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
Out[185]: <matplotlib.axes._subplots.AxesSubplot at 0x1c443518d0>
```



Andrews curves charts:

```
In [186]: plt.figure()
Out[186]: <Figure size 640x480 with 0 Axes>

In [187]: andrews_curves(data, 'Name', colormap='winter')
Out[187]: <matplotlib.axes._subplots.AxesSubplot at 0x1c441ff650>
```



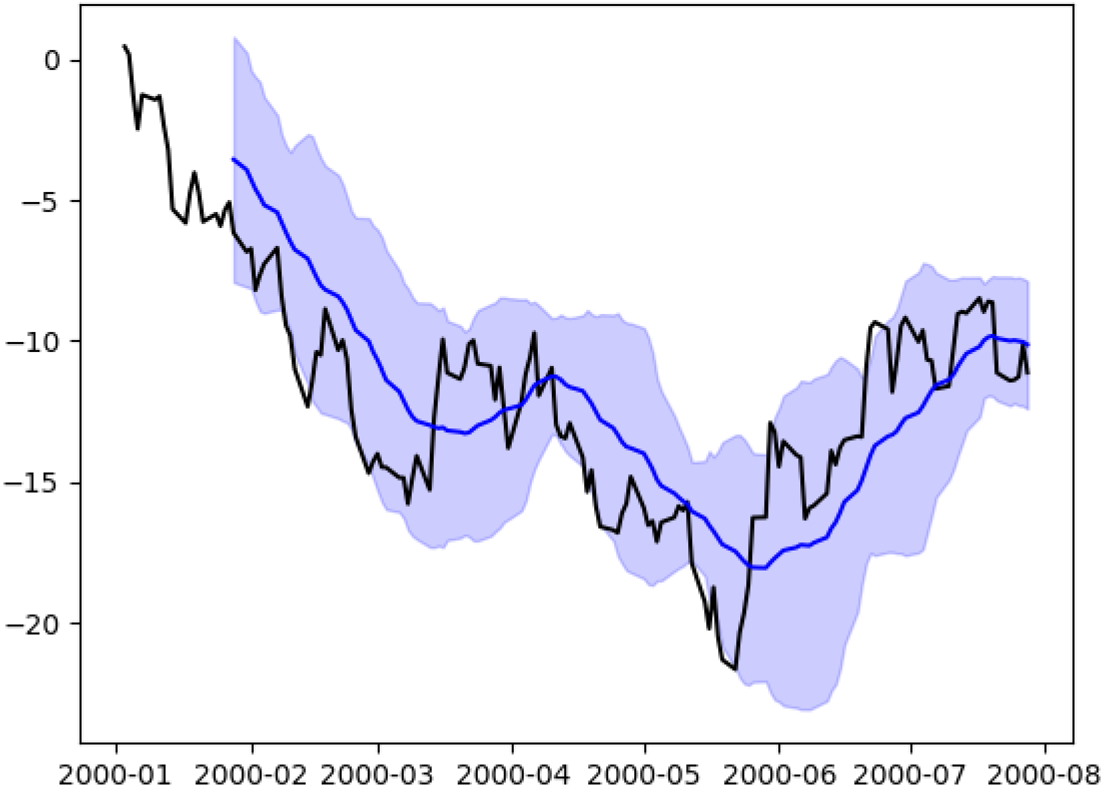
4.10.6 Plotting directly with matplotlib

In some situations it may still be preferable or necessary to prepare plots directly with matplotlib, for instance when a certain type of plot or customization is not (yet) supported by pandas. `Series` and `DataFrame` objects behave like arrays and can therefore be passed directly to matplotlib functions without explicit casts.

pandas also automatically registers formatters and locators that recognize date indices, thereby extending date and time support to practically all plot types available in matplotlib. Although this formatting does not provide the same level of refinement you would get when plotting via pandas, it can be faster when plotting a large number of points.

```
In [188]: price = pd.Series(np.random.randn(150).cumsum(),
   .....:                   index=pd.date_range('2000-1-1', periods=150,␣
↪freq='B'))
   .....:

In [189]: ma = price.rolling(20).mean()

In [190]: mstd = price.rolling(20).std()

In [191]: plt.figure()
Out[191]: <Figure size 640x480 with 0 Axes>

In [192]: plt.plot(price.index, price, 'k')
```

```
Out[192]: [<matplotlib.lines.Line2D at 0x1c44aaa350>]

In [193]: plt.plot(ma.index, ma, 'b')
Out[193]: [<matplotlib.lines.Line2D at 0x1c44a74ed0>]

In [194]: plt.fill_between(mstd.index, ma - 2 * mstd, ma + 2 * mstd,
   .....:                  color='b', alpha=0.2)
   .....:
Out[194]: <matplotlib.collections.PolyCollection at 0x1c44adea10>
```



4.10.7 Trellis plotting interface

> **Warning:** The `rplot` trellis plotting interface has been **removed**. Please use external packages like seaborn for similar but more refined functionality and refer to our 0.18.1 documentation here for how to convert to using it.

{{ header }}

4.11 Computational tools

4.11.1 Statistical functions

Percent change

Series and DataFrame have a method pct_change() to compute the percent change over a given number of periods (using fill_method to fill NA/null values *before* computing the percent change).

```
In [1]: ser = pd.Series(np.random.randn(8))

In [2]: ser.pct_change()
Out[2]:
0          NaN
1    13.547371
2    -1.548836
3    -2.936641
4    -2.449972
5    -1.103130
6    16.241433
7    -1.573605
dtype: float64
```

```
In [3]: df = pd.DataFrame(np.random.randn(10, 4))

In [4]: df.pct_change(periods=3)
Out[4]:
          0         1         2          3
0       NaN       NaN       NaN        NaN
1       NaN       NaN       NaN        NaN
2       NaN       NaN       NaN        NaN
3 -6.381415 -7.189288 -0.796567   5.324072
4 -1.594458 -0.193537 -1.514930  -0.943232
5 -2.689858 -2.369493  0.155967  -1.569737
6 -1.372184  0.635387 -2.303971  -0.509280
7 -6.194810 -2.482964 -0.764169 -25.332016
8 -0.923376 -2.560822 -2.479379  -2.656377
9 -2.978688 -0.504129 -5.959571  -5.033217
```

Covariance

Series.cov() can be used to compute covariance between series (excluding missing values).

```
In [5]: s1 = pd.Series(np.random.randn(1000))

In [6]: s2 = pd.Series(np.random.randn(1000))

In [7]: s1.cov(s2)
Out[7]: -0.035818144141082434
```

Analogously, DataFrame.cov() to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

Note: Assuming the missing data are missing at random this results in an estimate for the covariance matrix which is unbiased. However, for many applications this estimate may not be acceptable because the estimated covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimated correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
In [8]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ...:                      columns=['a', 'b', 'c', 'd', 'e'])
   ...:

In [9]: frame.cov()
Out[9]:
          a         b         c         d         e
a  0.969905 -0.031318 -0.021847 -0.002147 -0.045109
b -0.031318  0.948570  0.003914 -0.022931 -0.013969
c -0.021847  0.003914  1.039872  0.008068 -0.017560
d -0.002147 -0.022931  0.008068  1.059395 -0.045589
e -0.045109 -0.013969 -0.017560 -0.045589  1.105350
```

`DataFrame.cov` also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```
In [10]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [11]: frame.loc[frame.index[:5], 'a'] = np.nan

In [12]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [13]: frame.cov()
Out[13]:
          a         b         c
a  1.000881  0.151591 -0.143082
b  0.151591  0.869675 -0.035753
c -0.143082 -0.035753  0.737483

In [14]: frame.cov(min_periods=12)
Out[14]:
          a         b         c
a  1.000881       NaN -0.143082
b       NaN  0.869675 -0.035753
c -0.143082 -0.035753  0.737483
```

Correlation

Correlation may be computed using the `corr()` method. Using the `method` parameter, several methods for computing correlations 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. Wikipedia has articles covering the above correlation coefficients:

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearmans rank correlation coefficient

Note: Please see the *caveats* associated with this method of calculating correlation matrices in the *covariance section*.

```
In [15]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ....:                      columns=['a', 'b', 'c', 'd', 'e'])
   ....:

In [16]: frame.iloc[::2] = np.nan

# Series with Series
In [17]: frame['a'].corr(frame['b'])
Out[17]: 0.016886089025056897

In [18]: frame['a'].corr(frame['b'], method='spearman')
Out[18]: -0.02348697394789579

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:
          a         b         c         d         e
a  1.000000  0.016886 -0.028542  0.034697 -0.008209
b  0.016886  1.000000  0.000628 -0.004688 -0.028515
c -0.028542  0.000628  1.000000 -0.025158  0.104966
d  0.034697 -0.004688 -0.025158  1.000000 -0.014990
e -0.008209 -0.028515  0.104966 -0.014990  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 = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [21]: frame.loc[frame.index[:5], 'a'] = np.nan

In [22]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [23]: frame.corr()
Out[23]:
          a         b         c
a  1.000000  0.227635 -0.080537
b  0.227635  1.000000  0.034068
c -0.080537  0.034068  1.000000

In [24]: frame.corr(min_periods=12)
Out[24]:
          a         b         c
a  1.000000       NaN -0.080537
b       NaN  1.000000  0.034068
c -0.080537  0.034068  1.000000
```

New in version 0.24.0.

The `method` argument can also be a callable for a generic correlation calculation. In this case, it should be a single function that produces a single value from two ndarray inputs. Suppose we wanted to compute the correlation based on histogram intersection:

```
# histogram intersection
In [25]: def histogram_intersection(a, b):
   ....:     return np.minimum(np.true_divide(a, a.sum()),
   ....:                       np.true_divide(b, b.sum())).sum()
   ....:

In [26]: frame.corr(method=histogram_intersection)
Out[26]:
           a          b          c
a   1.000000 -7.283861 -31.104637
b  -7.283861  1.000000  -7.810865
c -31.104637 -7.810865   1.000000
```

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

```
In [27]: index = ['a', 'b', 'c', 'd', 'e']

In [28]: columns = ['one', 'two', 'three', 'four']

In [29]: df1 = pd.DataFrame(np.random.randn(5, 4), index=index,␣
↪columns=columns)

In [30]: df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4],␣
↪columns=columns)

In [31]: df1.corrwith(df2)
Out[31]:
one      0.526632
two      0.611188
three    0.434034
four     0.000843
dtype: float64

In [32]: df2.corrwith(df1, axis=1)
Out[32]:
a    0.791016
b    0.285772
c    0.798940
d   -0.946595
e         NaN
dtype: float64
```

Data ranking

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

```
In [33]: s = pd.Series(np.random.randn(5), index=list('abcde'))

In [34]: s['d'] = s['b']  # so there's a tie
```

(continues on next page)

(continued from previous page)

```
In [35]: s.rank()
Out[35]:
a    4.0
b    1.5
c    5.0
d    1.5
e    3.0
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 [36]: df = pd.DataFrame(np.random.randn(10, 6))

In [37]: df[4] = df[2][:5]  # some ties

In [38]: df
Out[38]:
          0         1         2         3         4         5
0  0.944050 -0.657738 -0.125993  0.286736 -0.125993 -0.864790
1  0.055086 -1.056551  0.458775  1.329940  0.458775 -0.697985
2 -0.567600  0.693650  0.962299 -1.216697  0.962299  1.092375
3  0.343301  0.407707 -0.698956 -1.110074 -0.698956  0.126931
4  1.084096  0.971689 -1.368406  2.444578 -1.368406  0.143508
5  0.349580 -1.681880 -0.403970 -1.023733       NaN  0.166872
6 -0.120038  0.797128  0.389633 -1.661451       NaN -0.673658
7 -1.121792 -1.485899 -0.804363  1.317388       NaN -1.382258
8  0.470715 -0.179893 -0.299046  0.153052       NaN -0.265199
9 -0.981253 -0.055122  1.602212  1.304666       NaN  0.443094

In [39]: df.rank(1)
Out[39]:
     0    1    2    3    4    5
0  6.0  2.0  3.5  5.0  3.5  1.0
1  3.0  1.0  4.5  6.0  4.5  2.0
2  2.0  3.0  4.5  1.0  4.5  6.0
3  5.0  6.0  2.5  1.0  2.5  4.0
4  5.0  4.0  1.5  6.0  1.5  3.0
5  5.0  1.0  3.0  2.0  NaN  4.0
6  3.0  5.0  4.0  1.0  NaN  2.0
7  3.0  1.0  4.0  5.0  NaN  2.0
8  5.0  3.0  1.0  4.0  NaN  2.0
9  1.0  2.0  5.0  4.0  NaN  3.0
```

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

4.11.2 Window Functions

For working with data, a number of window functions are provided for computing common *window* or *rolling* statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis.

The `rolling()` and `expanding()` functions can be used directly from DataFrameGroupBy objects, see the *groupby docs*.

Note: The API for window statistics is quite similar to the way one works with `GroupBy` objects, see the documentation *here*.

We work with `rolling`, `expanding` and `exponentially weighted` data through the corresponding objects, `Rolling`, `Expanding` and `EWM`.

```
In [40]: s = pd.Series(np.random.randn(1000),
   ....:               index=pd.date_range('1/1/2000', periods=1000))
   ....:

In [41]: s = s.cumsum()

In [42]: s
Out[42]:
2000-01-01    -0.176917
2000-01-02     0.316541
2000-01-03    -0.787075
2000-01-04    -2.107309
2000-01-05    -3.235250
                 ...
2002-09-22   -22.662237
2002-09-23   -24.731927
2002-09-24   -25.898290
2002-09-25   -25.522878
2002-09-26   -25.091748
Freq: D, Length: 1000, dtype: float64
```

These are created from methods on `Series` and `DataFrame`.

```
In [43]: r = s.rolling(window=60)

In [44]: r
Out[44]: Rolling [window=60,center=False,axis=0]
```

These object provide tab-completion of the available methods and properties.

```
In [14]: r.<TAB>                                          # noqa: E225, E999
r.agg          r.apply        r.count        r.exclusions   r.max          r.median       r.
↪name          r.skew         r.sum
r.aggregate    r.corr         r.cov          r.kurt         r.mean         r.min          r.
↪quantile      r.std          r.var
```

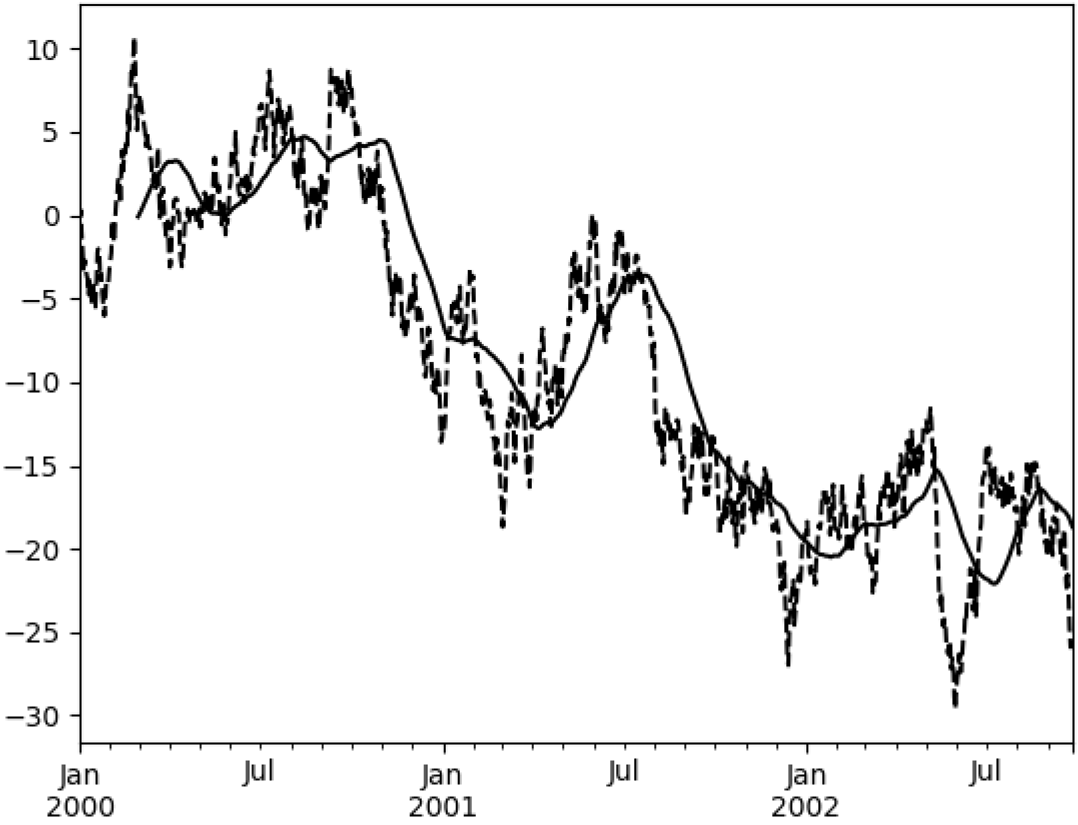
Generally these methods all have the same interface. They all accept the following arguments:

- `window`: size of moving window
- `min_periods`: threshold of non-null data points to require (otherwise result is NA)
- `center`: boolean, whether to set the labels at the center (default is False)

We can then call methods on these rolling objects. These return like-indexed objects:

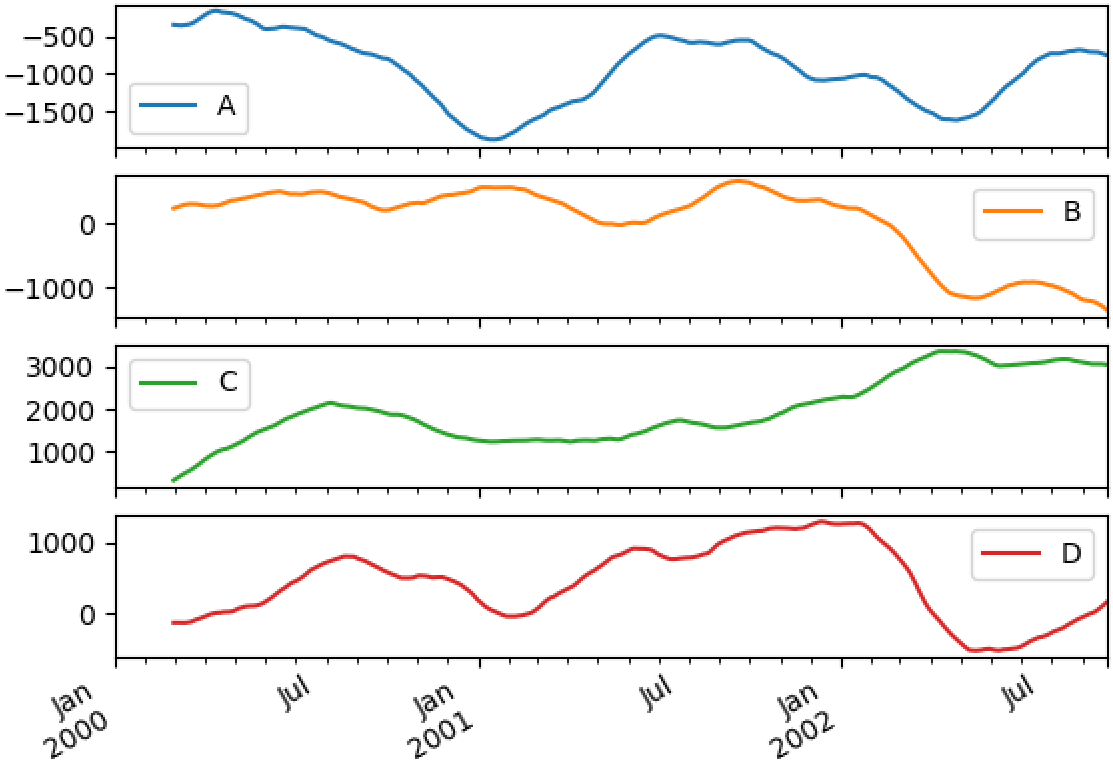
```
In [45]: r.mean()
Out[45]:
2000-01-01          NaN
2000-01-02          NaN
2000-01-03          NaN
2000-01-04          NaN
2000-01-05          NaN
                ...
2002-09-22   -18.067286
2002-09-23   -18.203986
2002-09-24   -18.378035
2002-09-25   -18.539518
2002-09-26   -18.676091
Freq: D, Length: 1000, dtype: float64
```

```
In [46]: s.plot(style='k--')
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x129c78b50>

In [47]: r.mean().plot(style='k')
Out[47]: <matplotlib.axes._subplots.AxesSubplot at 0x129c78b50>
```



They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrames columns:

```
In [48]: df = pd.DataFrame(np.random.randn(1000, 4),
   ....:                   index=pd.date_range('1/1/2000', periods=1000),
   ....:                   columns=['A', 'B', 'C', 'D'])
   ....:

In [49]: df = df.cumsum()

In [50]: df.rolling(window=60).sum().plot(subplots=True)
Out[50]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x125586b10>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x1a30878f90>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x125552f10>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x1a30415a90>],
      dtype=object)
```



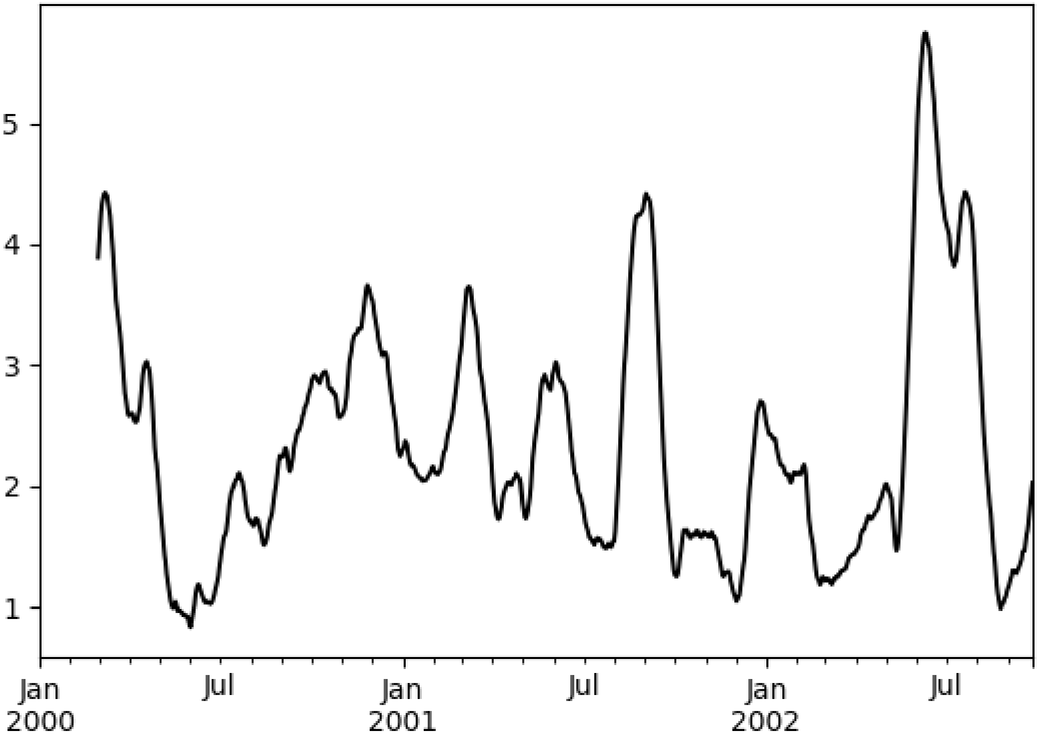
Method summary

We provide a number of common statistical functions:

Method	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
std()	Bessel-corrected sample standard deviation
var()	Unbiased variance
skew()	Sample skewness (3rd moment)
kurt()	Sample kurtosis (4th moment)
quantile()	Sample quantile (value at %)
apply()	Generic apply
cov()	Unbiased covariance (binary)
corr()	Correlation (binary)

The *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 [51]: def mad(x):
   ....:     return np.fabs(x - x.mean()).mean()
   ....:

In [52]: s.rolling(window=60).apply(mad, raw=True).plot(style='k')
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x1253717d0>
```



Rolling windows

Passing `win_type` to `.rolling` generates a generic rolling window computation, that is weighted according the `win_type`. The following methods are available:

Method	Description
sum()	Sum of values
mean()	Mean of values

The weights used in the window are specified by the `win_type` keyword. The list of recognized types are the scipy.signal window functions:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`

- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width)
- exponential (needs tau).

```
In [53]: ser = pd.Series(np.random.randn(10),
   ....:                 index=pd.date_range('1/1/2000', periods=10))
   ....:

In [54]: ser.rolling(window=5, win_type='triang').mean()
Out[54]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05    0.380626
2000-01-06    0.573337
2000-01-07    0.378764
2000-01-08    0.004447
2000-01-09   -0.331045
2000-01-10   -0.178263
Freq: D, dtype: float64
```

Note that the boxcar window is equivalent to *mean()*.

```
In [55]: ser.rolling(window=5, win_type='boxcar').mean()
Out[55]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05    0.540420
2000-01-06    0.306313
2000-01-07    0.081345
2000-01-08    0.254765
2000-01-09   -0.095579
2000-01-10   -0.237109
Freq: D, dtype: float64

In [56]: ser.rolling(window=5).mean()
Out[56]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05    0.540420
2000-01-06    0.306313
2000-01-07    0.081345
2000-01-08    0.254765
```

```
2000-01-09   -0.095579
2000-01-10   -0.237109
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [57]: ser.rolling(window=5, win_type='gaussian').mean(std=0.1)
Out[57]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.254437
2000-01-06    1.434351
2000-01-07    1.014205
2000-01-08   -0.460612
2000-01-09   -1.326783
2000-01-10    0.612663
Freq: D, dtype: float64
```

Note: For `.sum()` with a `win_type`, there is no normalization done to the weights for the window. Passing custom weights of `[1, 1, 1]` will yield a different result than passing weights of `[2, 2, 2]`, for example. When passing a `win_type` instead of explicitly specifying the weights, the weights are already normalized so that the largest weight is 1.

In contrast, the nature of the `.mean()` calculation is such that the weights are normalized with respect to each other. Weights of `[1, 1, 1]` and `[2, 2, 2]` yield the same result.

Time-aware rolling

New in version 0.19.0.

New in version 0.19.0 are the ability to pass an offset (or convertible) to a `.rolling()` method and have it produce variable sized windows based on the passed time window. For each time point, this includes all preceding values occurring within the indicated time delta.

This can be particularly useful for a non-regular time frequency index.

```
In [58]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.date_range('20130101 09:00:00',
   ....:                                        periods=5,
   ....:                                        freq='s'))
   ....:

In [59]: dft
Out[59]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [60]: dft.rolling(2).sum()
```

```
Out[60]:
                       B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN

In [61]: dft.rolling(2, min_periods=1).sum()
Out[61]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [62]: dft.rolling('2s').sum()
Out[62]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```
In [63]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.Index([pd.Timestamp('20130101 09:00:00'),
   ....:                                    pd.Timestamp('20130101 09:00:02'),
   ....:                                    pd.Timestamp('20130101 09:00:03'),
   ....:                                    pd.Timestamp('20130101 09:00:05'),
   ....:                                    pd.Timestamp('20130101␣
↪09:00:06')],
   ....:                                   name='foo'))
   ....:

In [64]: dft
Out[64]:
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

In [65]: dft.rolling(2).sum()
Out[65]:
                       B
foo
2013-01-01 09:00:00  NaN
2013-01-01 09:00:02  1.0
```

```
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  NaN
```

Using the time-specification generates variable windows for this sparse data.

```
In [66]: dft.rolling('2s').sum()
Out[66]:
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a DataFrame.

```
In [67]: dft = dft.reset_index()

In [68]: dft
Out[68]:
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  2.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0

In [69]: dft.rolling('2s', on='foo').sum()
Out[69]:
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  3.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0
```

Rolling window endpoints

New in version 0.20.0.

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

`closed`	Description	Default for
`right`	close right endpoint	time-based windows
`left`	close left endpoint	
`both`	close both endpoints	fixed windows
`neither`	open endpoints	

For example, having the right endpoint open is useful in many problems that require that there is no contamination from present information back to past information. This allows the rolling window to compute statistics up to that point in time, but not including that point in time.

```
In [70]: df = pd.DataFrame({'x': 1},
   ....:                   index=[pd.Timestamp('20130101 09:00:01'),
   ....:                          pd.Timestamp('20130101 09:00:02'),
   ....:                          pd.Timestamp('20130101 09:00:03'),
   ....:                          pd.Timestamp('20130101 09:00:04'),
   ....:                          pd.Timestamp('20130101 09:00:06')])
   ....:

In [71]: df["right"] = df.rolling('2s', closed='right').x.sum()  # default

In [72]: df["both"] = df.rolling('2s', closed='both').x.sum()

In [73]: df["left"] = df.rolling('2s', closed='left').x.sum()

In [74]: df["neither"] = df.rolling('2s', closed='neither').x.sum()

In [75]: df
Out[75]:
                     x  right  both  left  neither
2013-01-01 09:00:01  1    1.0   1.0   NaN      NaN
2013-01-01 09:00:02  1    2.0   2.0   1.0      1.0
2013-01-01 09:00:03  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:04  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:06  1    1.0   2.0   1.0      NaN
```

Currently, this feature is only implemented for time-based windows. For fixed windows, the closed parameter cannot be set and the rolling window will always have both endpoints closed.

Time-aware rolling vs. resampling

Using `.rolling()` with a time-based index is quite similar to *resampling*. They both operate and perform reductive operations on time-indexed pandas objects.

When using `.rolling()` with an offset. The offset is a time-delta. Take a backwards-in-time looking window, and aggregate all of the values in that window (including the end-point, but not the start-point). This is the new value at that point in the result. These are variable sized windows in time-space for each point of the input. You will get a same sized result as the input.

When using `.resample()` with an offset. Construct a new index that is the frequency of the offset. For each frequency bin, aggregate points from the input within a backwards-in-time looking window that fall in that bin. The result of this aggregation is the output for that frequency point. The windows are fixed size in the frequency space. Your result will have the shape of a regular frequency between the min and the max of the original input object.

To summarize, `.rolling()` is a time-based window operation, while `.resample()` is a frequency-based window operation.

Centering windows

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.

```
In [76]: ser.rolling(window=5).mean()
Out[76]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
```

```
2000-01-04          NaN
2000-01-05     0.540420
2000-01-06     0.306313
2000-01-07     0.081345
2000-01-08     0.254765
2000-01-09    -0.095579
2000-01-10    -0.237109
Freq: D, dtype: float64

In [77]: ser.rolling(window=5, center=True).mean()
Out[77]:
2000-01-01          NaN
2000-01-02          NaN
2000-01-03     0.540420
2000-01-04     0.306313
2000-01-05     0.081345
2000-01-06     0.254765
2000-01-07    -0.095579
2000-01-08    -0.237109
2000-01-09          NaN
2000-01-10          NaN
Freq: D, dtype: float64
```

Binary window functions

cov() and *corr()* can compute moving window statistics about two `Series` or any combination of `DataFrame/Series` or `DataFrame/DataFrame`. Here is the behavior in each case:

- two `Series`: compute the statistic for the pairing.
- `DataFrame/Series`: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame.
- `DataFrame/DataFrame`: by default compute the statistic for matching column names, returning a DataFrame. If the keyword argument `pairwise=True` is passed then computes the statistic for each pair of columns, returning a `MultiIndexed DataFrame` whose `index` are the dates in question (see *the next section*).

For example:

```
In [78]: df = pd.DataFrame(np.random.randn(1000, 4),
   ....:                   index=pd.date_range('1/1/2000', periods=1000),
   ....:                   columns=['A', 'B', 'C', 'D'])
   ....:

In [79]: df = df.cumsum()

In [80]: df2 = df[:20]

In [81]: df2.rolling(window=5).corr(df2['B'])
Out[81]:
                   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
```

(continues on next page)

(continued from previous page)

```
2000-01-05 -0.410454  1.0  0.523221 -0.860934
2000-01-06  0.691714  1.0 -0.006852 -0.883151
2000-01-07  0.127010  1.0 -0.819233 -0.920657
2000-01-08  0.083226  1.0 -0.707348 -0.612911
2000-01-09  0.366106  1.0 -0.868906  0.814301
2000-01-10 -0.027745  1.0 -0.948739 -0.236256
2000-01-11 -0.503110  1.0 -0.512448 -0.387532
2000-01-12  0.245194  1.0 -0.269958 -0.284629
2000-01-13  0.045798  1.0 -0.512087 -0.209225
2000-01-14 -0.226867  1.0 -0.580830  0.203087
2000-01-15 -0.152839  1.0 -0.566645  0.034572
2000-01-16  0.385888  1.0 -0.283130  0.108951
2000-01-17  0.057373  1.0  0.493143 -0.044273
2000-01-18 -0.162383  1.0  0.449045 -0.189665
2000-01-19 -0.174013  1.0  0.266842 -0.661392
2000-01-20  0.486804  1.0  0.120970  0.139172
```

Computing rolling pairwise covariances and correlations

In financial data analysis and other fields its common to compute covariance and correlation matrices for a collection of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by passing the `pairwise` keyword argument, which in the case of `DataFrame` inputs will yield a MultiIndexed `DataFrame` whose `index` are the dates in question. In the case of a single DataFrame argument the `pairwise` argument can even be omitted:

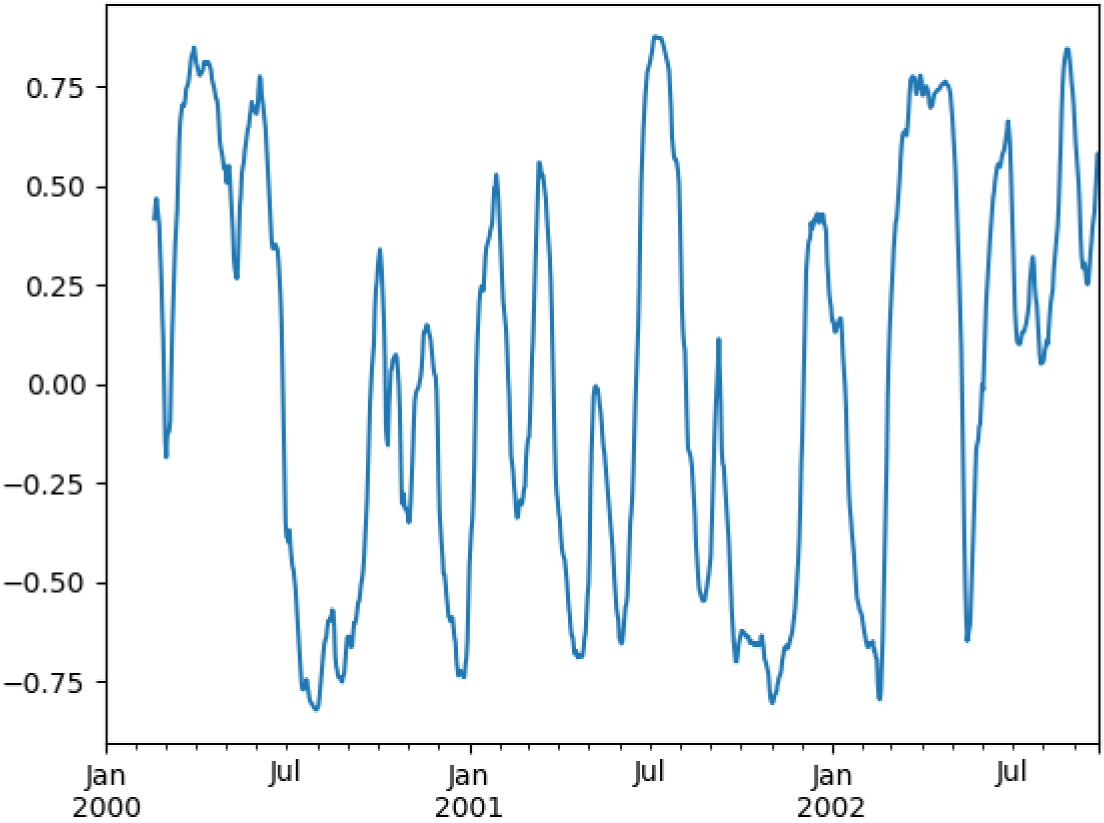
Note: Missing values are ignored and each entry is computed using the pairwise complete observations. Please see the *covariance section* for *caveats* associated with this method of calculating covariance and correlation matrices.

```
In [82]: covs = (df[['B', 'C', 'D']].rolling(window=50)
   ....:                               .cov(df[['A', 'B', 'C']], pairwise=True))
   ....:

In [83]: covs.loc['2002-09-22':]
Out[83]:
                      B          C          D
2002-09-22 A -11.788482  3.788620  -9.532202
           B  27.214936  0.218853  13.566773
           C   0.218853  5.933531  -4.591019
2002-09-23 A -12.096941  4.076409 -10.208554
           B  27.436671 -0.830191  15.370018
           C  -0.830191  5.582126  -4.811035
2002-09-24 A -12.239394  4.330590 -10.742513
           B  27.226658 -1.658535  16.437777
           C  -1.658535  5.143402  -4.960592
2002-09-25 A -12.752127  4.548132 -11.577657
           B  27.467830 -2.486227  17.845438
           C  -2.486227  4.540150  -4.973443
2002-09-26 A -13.236082  4.531937 -12.379683
           B  27.459811 -2.780055  19.188470
           C  -2.780055  4.387162  -4.798969
```

```
In [84]: correls = df.rolling(window=50).corr()

In [85]: correls.loc['2002-09-22':]
Out[85]:
                     A         B         C         D
2002-09-22 A  1.000000 -0.632380  0.435259 -0.758112
           B -0.632380  1.000000  0.017222  0.739077
           C  0.435259  0.017222  1.000000 -0.535635
           D -0.758112  0.739077 -0.535635  1.000000
2002-09-23 A  1.000000 -0.642018  0.479640 -0.757166
           B -0.642018  1.000000 -0.067083  0.782885
           C  0.479640 -0.067083  1.000000 -0.543285
           D -0.757166  0.782885 -0.543285  1.000000
2002-09-24 A  1.000000 -0.648609  0.528010 -0.758338
           B -0.648609  1.000000 -0.140153  0.804234
           C  0.528010 -0.140153  1.000000 -0.558399
           D -0.758338  0.804234 -0.558399  1.000000
2002-09-25 A  1.000000 -0.662761  0.581413 -0.768971
           B -0.662761  1.000000 -0.222635  0.830267
           C  0.581413 -0.222635  1.000000 -0.569147
           D -0.768971  0.830267 -0.569147  1.000000
2002-09-26 A  1.000000 -0.678836  0.581495 -0.774662
           B -0.678836  1.000000 -0.253287  0.852590
           C  0.581495 -0.253287  1.000000 -0.533463
           D -0.774662  0.852590 -0.533463  1.000000
```

You can efficiently retrieve the time series of correlations between two columns by reshaping and indexing:

```
In [86]: correls.unstack(1)[('A', 'C')].plot()
Out[86]: <matplotlib.axes._subplots.AxesSubplot at 0x1a30dc52d0>
```



4.11.3 Aggregation

Once the `Rolling`, `Expanding` or `EWM` objects have been created, several methods are available to perform multiple computations on the data. These operations are similar to the *aggregating API*, *groupby API*, and *resample API*.

```
In [87]: dfa = pd.DataFrame(np.random.randn(1000, 3),
   ....:                    index=pd.date_range('1/1/2000', periods=1000),
   ....:                    columns=['A', 'B', 'C'])
   ....:

In [88]: r = dfa.rolling(window=60, min_periods=1)

In [89]: r
Out[89]: Rolling [window=60,min_periods=1,center=False,axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a Series (or multiple Series) via standard `__getitem__`.

```
In [90]: r.aggregate(np.sum)
Out[90]:
                   A         B         C
2000-01-01  1.467445  0.520922  0.333619
2000-01-02  0.913815  0.659062  0.678084
```

```
2000-01-03  1.697014  2.104923  1.540576
2000-01-04  1.205306  2.617423  0.435920
2000-01-05  0.474929  1.072642  0.866405
...              ...       ...       ...
2002-09-22 -2.120855  1.278532  0.615368
2002-09-23 -1.517675  2.423909  4.625449
2002-09-24 -0.633229  1.370941  5.138608
2002-09-25 -1.552531  0.546496  6.269487
2002-09-26  0.711361  2.818447  4.596824

[1000 rows x 3 columns]

In [91]: r['A'].aggregate(np.sum)
Out[91]:
2000-01-01    1.467445
2000-01-02    0.913815
2000-01-03    1.697014
2000-01-04    1.205306
2000-01-05    0.474929
                ...
2002-09-22   -2.120855
2002-09-23   -1.517675
2002-09-24   -0.633229
2002-09-25   -1.552531
2002-09-26    0.711361
Freq: D, Name: A, Length: 1000, dtype: float64

In [92]: r[['A', 'B']].aggregate(np.sum)
Out[92]:
                   A         B
2000-01-01  1.467445  0.520922
2000-01-02  0.913815  0.659062
2000-01-03  1.697014  2.104923
2000-01-04  1.205306  2.617423
2000-01-05  0.474929  1.072642
...              ...       ...
2002-09-22 -2.120855  1.278532
2002-09-23 -1.517675  2.423909
2002-09-24 -0.633229  1.370941
2002-09-25 -1.552531  0.546496
2002-09-26  0.711361  2.818447

[1000 rows x 2 columns]
```

As you can see, the result of the aggregation will have the selected columns, or all columns if none are selected.

Applying multiple functions

With windowed `Series` you can also pass a list of functions to do aggregation with, outputting a DataFrame:

```
In [93]: r['A'].agg([np.sum, np.mean, np.std])
Out[93]:
               sum      mean       std
```

(continues on next page)

(continued from previous page)

```
2000-01-01  1.467445  1.467445       NaN
2000-01-02  0.913815  0.456908  1.429116
2000-01-03  1.697014  0.565671  1.027947
2000-01-04  1.205306  0.301326  0.991949
2000-01-05  0.474929  0.094986  0.975118
...              ...       ...       ...
2002-09-22 -2.120855 -0.035348  1.000696
2002-09-23 -1.517675 -0.025295  0.990434
2002-09-24 -0.633229 -0.010554  0.995740
2002-09-25 -1.552531 -0.025876  1.003939
2002-09-26  0.711361  0.011856  1.018469

[1000 rows x 3 columns]
```

On a windowed DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [94]: r.agg([np.sum, np.mean])
Out[94]:
                   A                   B                   C          
                 sum      mean       sum      mean       sum      mean
2000-01-01  1.467445  1.467445  0.520922  0.520922  0.333619  0.333619
2000-01-02  0.913815  0.456908  0.659062  0.329531  0.678084  0.339042
2000-01-03  1.697014  0.565671  2.104923  0.701641  1.540576  0.513525
2000-01-04  1.205306  0.301326  2.617423  0.654356  0.435920  0.108980
2000-01-05  0.474929  0.094986  1.072642  0.214528  0.866405  0.173281
...              ...       ...       ...       ...       ...       ...
2002-09-22 -2.120855 -0.035348  1.278532  0.021309  0.615368  0.010256
2002-09-23 -1.517675 -0.025295  2.423909  0.040398  4.625449  0.077091
2002-09-24 -0.633229 -0.010554  1.370941  0.022849  5.138608  0.085643
2002-09-25 -1.552531 -0.025876  0.546496  0.009108  6.269487  0.104491
2002-09-26  0.711361  0.011856  2.818447  0.046974  4.596824  0.076614

[1000 rows x 6 columns]
```

Passing a dict of functions has different behavior by default, see the next section.

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 [95]: r.agg({'A': np.sum, 'B': lambda x: np.std(x, ddof=1)})
Out[95]:
                   A         B
2000-01-01  1.467445       NaN
2000-01-02  0.913815  0.270668
2000-01-03  1.697014  0.672331
2000-01-04  1.205306  0.557042
2000-01-05  0.474929  1.095428
...              ...       ...
2002-09-22 -2.120855  1.100184
2002-09-23 -1.517675  1.106164
2002-09-24 -0.633229  1.099861
2002-09-25 -1.552531  1.099031
2002-09-26  0.711361  1.159605
```

(continues on next page)

(continued from previous page)

```

[1000 rows x 2 columns]
```

The function names can also be strings. In order for a string to be valid it must be implemented on the windowed object

```
In [96]: r.agg({'A': 'sum', 'B': 'std'})
Out[96]:
                   A         B
2000-01-01  1.467445       NaN
2000-01-02  0.913815  0.270668
2000-01-03  1.697014  0.672331
2000-01-04  1.205306  0.557042
2000-01-05  0.474929  1.095428
...              ...       ...
2002-09-22 -2.120855  1.100184
2002-09-23 -1.517675  1.106164
2002-09-24 -0.633229  1.099861
2002-09-25 -1.552531  1.099031
2002-09-26  0.711361  1.159605

[1000 rows x 2 columns]
```

Furthermore you can pass a nested dict to indicate different aggregations on different columns.

```
In [97]: r.agg({'A': ['sum', 'std'], 'B': ['mean', 'std']})
Out[97]:
                   A                   B
                 sum       std      mean       std
2000-01-01  1.467445       NaN  0.520922       NaN
2000-01-02  0.913815  1.429116  0.329531  0.270668
2000-01-03  1.697014  1.027947  0.701641  0.672331
2000-01-04  1.205306  0.991949  0.654356  0.557042
2000-01-05  0.474929  0.975118  0.214528  1.095428
...              ...       ...       ...       ...
2002-09-22 -2.120855  1.000696  0.021309  1.100184
2002-09-23 -1.517675  0.990434  0.040398  1.106164
2002-09-24 -0.633229  0.995740  0.022849  1.099861
2002-09-25 -1.552531  1.003939  0.009108  1.099031
2002-09-26  0.711361  1.018469  0.046974  1.159605

[1000 rows x 4 columns]
```

4.11.4 Expanding windows

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.

These follow a similar interface to `.rolling`, with the `.expanding` method returning an `Expanding` object.

As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```
In [98]: df.rolling(window=len(df), min_periods=1).mean()[:5]
Out[98]:
                   A         B         C         D
```

```
2000-01-01  0.382444  0.209948 -0.510030 -0.394100
2000-01-02  0.030851  0.445864 -0.374918 -1.316470
2000-01-03 -0.100175  1.104252 -0.160086 -1.775538
2000-01-04 -0.192204  1.391755  0.077095 -1.869501
2000-01-05 -0.181876  1.961661  0.071067 -2.189869

In [99]: df.expanding(min_periods=1).mean()[:5]
Out[99]:
                   A         B         C         D
2000-01-01  0.382444  0.209948 -0.510030 -0.394100
2000-01-02  0.030851  0.445864 -0.374918 -1.316470
2000-01-03 -0.100175  1.104252 -0.160086 -1.775538
2000-01-04 -0.192204  1.391755  0.077095 -1.869501
2000-01-05 -0.181876  1.961661  0.071067 -2.189869
```

These have a similar set of methods to `.rolling` methods.

Method summary

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
std()	Unbiased standard deviation
var()	Unbiased variance
skew()	Unbiased skewness (3rd moment)
kurt()	Unbiased kurtosis (4th moment)
quantile()	Sample quantile (value at %)
apply()	Generic apply
cov()	Unbiased covariance (binary)
corr()	Correlation (binary)

Aside from not having a `window` parameter, these functions have the same interfaces as their `.rolling` counterparts. 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 `NaN`s will be output once `min_periods` non-null data points have been seen.
- `center`: boolean, whether to set the labels at the center (default is False).

Note: The output of the `.rolling` and `.expanding` methods do not return a `NaN` if there are at least `min_periods` non-null values in the current window. For example:

```
In [100]: sn = pd.Series([1, 2, np.nan, 3, np.nan, 4])

In [101]: sn
Out[101]:
0    1.0
1    2.0
2    NaN
```

```
3    3.0
4    NaN
5    4.0
dtype: float64

In [102]: sn.rolling(2).max()
Out[102]:
0    NaN
1    2.0
2    NaN
3    NaN
4    NaN
5    NaN
dtype: float64

In [103]: sn.rolling(2, min_periods=1).max()
Out[103]:
0    1.0
1    2.0
2    2.0
3    3.0
4    3.0
5    4.0
dtype: float64
```

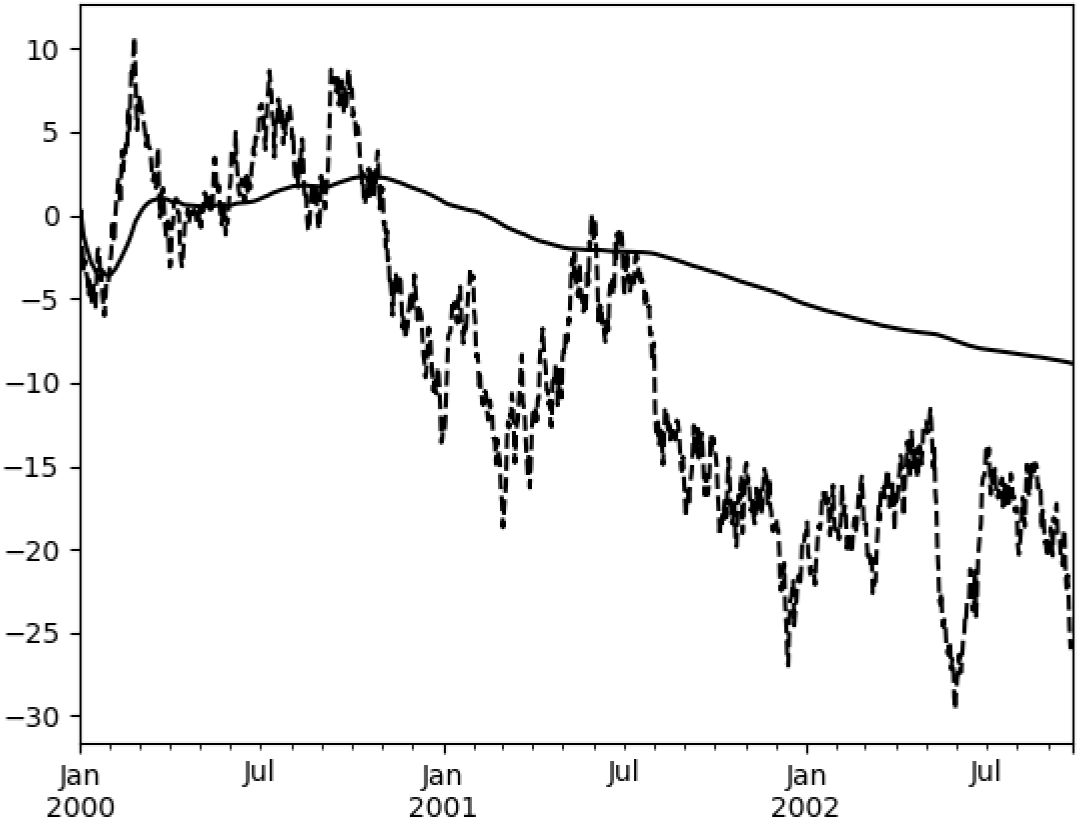
In case of expanding functions, this differs from *cumsum()*, *cumprod()*, *cummax()*, and *cummin()*, which return NaN in the output wherever a NaN is encountered in the input. In order to match the output of cumsum with expanding, use *fillna()*:

```
In [104]: sn.expanding().sum()
Out[104]:
0     1.0
1     3.0
2     3.0
3     6.0
4     6.0
5    10.0
dtype: float64

In [105]: sn.cumsum()
Out[105]:
0     1.0
1     3.0
2     NaN
3     6.0
4     NaN
5    10.0
dtype: float64

In [106]: sn.cumsum().fillna(method='ffill')
Out[106]:
0     1.0
1     3.0
2     3.0
3     6.0
```

```
4     6.0
5    10.0
dtype: float64
```

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 *mean()* output for the previous time series dataset:

```
In [107]: s.plot(style='k--')
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x1a313078d0>

In [108]: s.expanding().mean().plot(style='k')
Out[108]: <matplotlib.axes._subplots.AxesSubplot at 0x1a313078d0>
```



4.11.5 Exponentially weighted windows

A related set of functions are exponentially weighted versions of several of the above statistics. A similar interface to `.rolling` and `.expanding` is accessed through the `.ewm` method to receive an `EWM` object. A number of expanding EW (exponentially weighted) methods are provided:

Function	Description
mean()	EW moving average
var()	EW moving variance
std()	EW moving standard deviation
corr()	EW moving correlation
cov()	EW moving covariance

In general, a weighted moving average is calculated as

$$y_t = \frac{\sum_{i=0}^t w_i x_{t-i}}{\sum_{i=0}^t w_i},$$

where x_t is the input, y_t is the result and the w_i are the weights.

The EW functions support two variants of exponential weights. The default, `adjust=True`, uses the weights $w_i = (1-\alpha)^i$ which gives

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ... + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^t}$$

When `adjust=False` is specified, moving averages are calculated as

$$\begin{split}y_0 &= x_0\\ y_t &= (1-\alpha)y_{t-1} + \alpha x_t,\end{split}$$

which is equivalent to using weights

$$w_i = \begin{cases} \alpha(1-\alpha)^i & \text{if } i < t\\ (1-\alpha)^i & \text{if } i = t.\end{cases}$$

Note: These equations are sometimes written in terms of $\alpha' = 1 - \alpha$, e.g.

$$y_t = \alpha' y_{t-1} + (1-\alpha')x_t.$$

The difference between the above two variants arises because we are dealing with series which have finite history. Consider a series of infinite history, with `adjust=True`:

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{1 + (1-\alpha) + (1-\alpha)^2 + ...}$$

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of $1-\alpha$ we have

$$\begin{split}y_t &= \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{\frac{1}{1-(1-\alpha)}}\\ &= [x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha\\ &= \alpha x_t + [(1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha\\ &= \alpha x_t + (1-\alpha)[x_{t-1} + (1-\alpha)x_{t-2} + ...]\alpha\\ &= \alpha x_t + (1-\alpha)y_{t-1}\end{split}$$

which is the same expression as `adjust=False` above and therefore shows the equivalence of the two variants for infinite series. When `adjust=False`, we have $y_0 = x_0$ and $y_t = \alpha x_t + (1-\alpha)y_{t-1}$. Therefore, there is an assumption that x_0 is not an ordinary value but rather an exponentially weighted moment of the infinite series up to that point.

One must have $0 < \alpha \leq 1$, and while since version 0.18.0 it has been possible to pass α directly, its often easier to think about either the **span**, **center of mass (com)** or **half-life** of an EW moment:

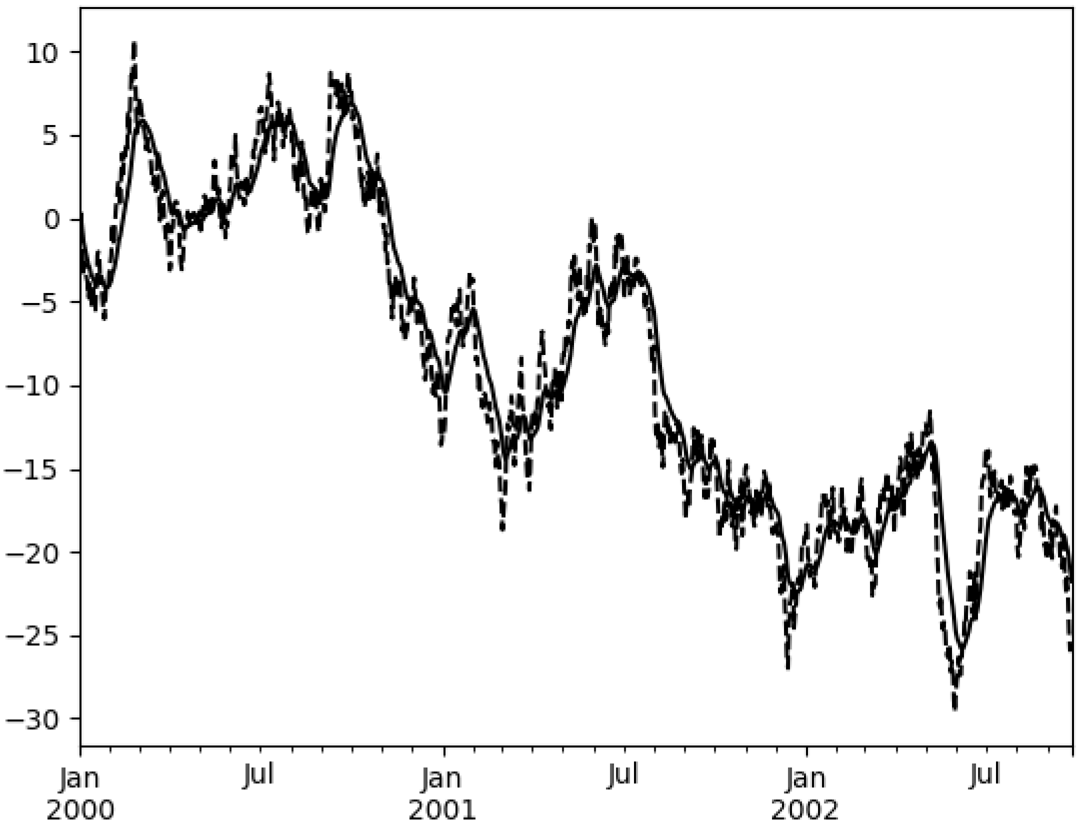
$$\alpha = \begin{cases} \frac{2}{s+1}, & \text{for span } s \geq 1\\ \frac{1}{1+c}, & \text{for center of mass } c \geq 0\\ 1 - \exp^{\frac{\log 0.5}{h}}, & \text{for half-life } h > 0 \end{cases}$$

One must specify precisely one of **span**, **center of mass**, **half-life** and **alpha** to the EW functions:

- **Span** corresponds to what is commonly called an N-day EW moving average.
- **Center of mass** has a more physical interpretation and can be thought of in terms of span: $c = (s - 1)/2$.
- **Half-life** is the period of time for the exponential weight to reduce to one half.
- **Alpha** specifies the smoothing factor directly.

Here is an example for a univariate time series:

```
In [109]: s.plot(style='k--')
Out[109]: <matplotlib.axes._subplots.AxesSubplot at 0x1a312dc6d0>

In [110]: s.ewm(span=20).mean().plot(style='k')
Out[110]: <matplotlib.axes._subplots.AxesSubplot at 0x1a312dc6d0>
```



EWM has a `min_periods` argument, which has the same meaning it does for all the `.expanding` and `.rolling` methods: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window.

EWM also has an `ignore_na` argument, which determines how intermediate null values affect the calculation of the weights. When `ignore_na=False` (the default), weights are calculated based on absolute positions, so that intermediate null values affect the result. When `ignore_na=True`, weights are calculated by ignoring intermediate null values. For example, assuming `adjust=True`, if `ignore_na=False`, the weighted average of `3, NaN, 5` would be calculated as

$$\frac{(1-\alpha)^2 \cdot 3 + 1 \cdot 5}{(1-\alpha)^2 + 1}.$$

Whereas if `ignore_na=True`, the weighted average would be calculated as

$$\frac{(1-\alpha) \cdot 3 + 1 \cdot 5}{(1-\alpha) + 1}.$$

The `var()`, `std()`, and `cov()` functions have a `bias` argument, specifying whether the result should contain biased or unbiased statistics. For example, if `bias=True`, `ewmvar(x)` is calculated as `ewmvar(x) = ewma(x**2) - ewma(x)**2`; whereas if `bias=False` (the default), the biased variance statistics are scaled by debiasing factors

$$\frac{\left(\sum_{i=0}^t w_i\right)^2}{\left(\sum_{i=0}^t w_i\right)^2 - \sum_{i=0}^t w_i^2}.$$

(For $w_i = 1$, this reduces to the usual $N/(N-1)$ factor, with $N = t + 1$.) See Weighted Sample Variance on Wikipedia for further details. {{ header }}

4.12 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.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we might wish to do one of the following:

- **Aggregation**: compute a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means.
  - Compute group sizes / counts.
- **Transformation**: perform some group-specific computations and return a like-indexed object. Some examples:
  - Standardize data (zscore) within a 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:
  - Discard data that belongs to groups with only a few members.
  - Filter 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 doesnt fit into either of the above two categories.

Since the set of object instance methods 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. Well address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the *cookbook* for some advanced strategies.

4.12.1 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 may do the following:

```
In [1]: df = pd.DataFrame([('bird', 'Falconiformes', 389.0),
   ...:                    ('bird', 'Psittaciformes', 24.0),
   ...:                    ('mammal', 'Carnivora', 80.2),
   ...:                    ('mammal', 'Primates', np.nan),
   ...:                    ('mammal', 'Carnivora', 58)],
   ...:                   index=['falcon', 'parrot', 'lion', 'monkey', 'leopard'],
   ...:                   columns=('class', 'order', 'max_speed'))
   ...:

In [2]: df
Out[2]:
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0

# default is axis=0
In [3]: grouped = df.groupby('class')

In [4]: grouped = df.groupby('order', axis='columns')

In [5]: grouped = df.groupby(['class', 'order'])
```

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.
- For `DataFrame` objects, a string indicating an index level to be used to group.
- A list of any of the above things.

Collectively we refer to the grouping objects as the **keys**. For example, consider the following DataFrame:

Note: A string passed to groupby may refer to either a column or an index level. If a string matches both a column name and an index level name, a ValueError will be raised.

```
In [6]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ...:                          'foo', 'bar', 'foo', 'foo'],
   ...:                    'B': ['one', 'one', 'two', 'three',
   ...:                          'two', 'two', 'one', 'three'],
   ...:                    'C': np.random.randn(8),
   ...:                    'D': np.random.randn(8)})
   ...:

In [7]: df
Out[7]:
     A      B         C         D
0  foo    one -0.024896  0.238006
1  bar    one  0.286068 -0.761142
2  foo    two  0.177576  2.097385
3  bar  three -0.143033  0.354051
4  foo    two  0.960747 -0.687808
5  bar    two -1.675479  1.725302
6  foo    one  0.227102 -0.755041
7  foo  three -0.557170  0.230925
```

On a DataFrame, we obtain a GroupBy object by calling groupby(). We could naturally group by either the A or B columns, or both:

```
In [8]: grouped = df.groupby('A')

In [9]: grouped = df.groupby(['A', 'B'])
```

New in version 0.24.

If we also have a MultiIndex on columns A and B, we can group by all but the specified columns

```
In [10]: df2 = df.set_index(['A', 'B'])

In [11]: grouped = df2.groupby(level=df2.index.names.difference(['B']))

In [12]: grouped.sum()
Out[12]:
            C         D
A
bar -1.532443  1.318211
foo  0.783358  1.123467
```

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

```
In [13]: def get_letter_type(letter):
   ....:     if letter.lower() in 'aeiou':
   ....:         return 'vowel'
   ....:     else:
   ....:         return 'consonant'
   ....:

In [14]: grouped = df.groupby(get_letter_type, axis=1)
```

pandas `Index` objects support 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 [15]: lst = [1, 2, 3, 1, 2, 3]

In [16]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)

In [17]: grouped = s.groupby(level=0)

In [18]: grouped.first()
Out[18]:
1    1
2    2
3    3
dtype: int64

In [19]: grouped.last()
Out[19]:
1    10
2    20
3    30
dtype: int64

In [20]: grouped.sum()
Out[20]:
1    11
2    22
3    33
dtype: int64
```

Note that **no splitting occurs** until its needed. Creating the GroupBy object only verifies that youve passed a valid mapping.

Note: Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though cant be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.

GroupBy sorting

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

```
In [21]: df2 = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [22]: df2.groupby(['X']).sum()
Out[22]:
   Y
X
A  7
B  3

In [23]: df2.groupby(['X'], sort=False).sum()
Out[23]:
   Y
X
B  3
A  7
```

Note that `groupby` will preserve the order in which *observations* are sorted *within* each group. For example, the groups created by `groupby()` below are in the order they appeared in the original `DataFrame`:

```
In [24]: df3 = pd.DataFrame({'X': ['A', 'B', 'A', 'B'], 'Y': [1, 4, 3, 2]})

In [25]: df3.groupby(['X']).get_group('A')
Out[25]:
   X  Y
0  A  1
2  A  3

In [26]: df3.groupby(['X']).get_group('B')
Out[26]:
   X  Y
1  B  4
3  B  2
```

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 [27]: df.groupby('A').groups
Out[27]:
{'bar': Int64Index([1, 3, 5], dtype='int64'),
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}

In [28]: df.groupby(get_letter_type, axis=1).groups
Out[28]:
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}
```

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 [29]: grouped = df.groupby(['A', 'B'])

In [30]: grouped.groups
Out[30]:
{('bar', 'one'): Int64Index([1], dtype='int64'),
 ('bar', 'three'): Int64Index([3], dtype='int64'),
 ('bar', 'two'): Int64Index([5], dtype='int64'),
 ('foo', 'one'): Int64Index([0, 6], dtype='int64'),
 ('foo', 'three'): Int64Index([7], dtype='int64'),
 ('foo', 'two'): Int64Index([2, 4], dtype='int64')}

In [31]: len(grouped)
Out[31]: 6
```

`GroupBy` will tab complete column names (and other attributes):

```
In [32]: df
Out[32]:
               height      weight  gender
2000-01-01  66.625231  167.442586  female
2000-01-02  67.783272  170.355866    male
2000-01-03  59.226150  132.525248  female
2000-01-04  45.989647  132.927883  female
2000-01-05  55.485350  142.663201    male
2000-01-06  49.650341  169.739406    male
2000-01-07  70.884977  152.414050  female
2000-01-08  57.512599  178.333864    male
2000-01-09  57.653796  177.259389  female
2000-01-10  59.657849  146.833646  female

In [33]: gb = df.groupby('gender')
```

```
In [34]: gb.<TAB>  # noqa: E225, E999
gb.agg        gb.boxplot    gb.cummin     gb.describe   gb.filter     gb.get_group ␣
↪gb.height      gb.last       gb.median     gb.ngroups    gb.plot       gb.rank      ␣
↪gb.std         gb.transform
gb.aggregate  gb.count      gb.cumprod    gb.dtype      gb.first      gb.groups    ␣
↪gb.hist        gb.max        gb.min        gb.nth        gb.prod       gb.resample  ␣
↪gb.sum         gb.var
gb.apply      gb.cummax     gb.cumsum     gb.fillna     gb.gender     gb.head      ␣
↪gb.indices     gb.mean       gb.name       gb.ohlc       gb.quantile   gb.size      ␣
↪gb.tail        gb.weight
```

GroupBy with MultiIndex

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

Lets create a Series with a two-level MultiIndex.

```
In [35]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [36]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [37]: s = pd.Series(np.random.randn(8), index=index)

In [38]: s
Out[38]:
first  second
bar    one       0.589964
       two      -1.842844
baz    one       1.601648
       two       0.322967
foo    one       1.668732
       two       1.291467
qux    one      -0.492679
       two      -0.557110
dtype: float64
```

We can then group by one of the levels in s.

```
In [39]: grouped = s.groupby(level=0)

In [40]: grouped.sum()
Out[40]:
first
bar   -1.252880
baz    1.924614
foo    2.960199
qux   -1.049789
dtype: float64
```

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

```
In [41]: s.groupby(level='second').sum()
Out[41]:
second
one    3.367665
two   -0.785521
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 [42]: s.sum(level='second')
Out[42]:
second
one    3.367665
two   -0.785521
dtype: float64
```

Grouping with multiple levels is supported.

```
In [43]: s
Out[43]:
first  second  third
bar    doo     one     -2.165187
               two     -0.293011
baz    bee     one      0.416483
               two      1.027560
foo    bop     one     -0.422062
               two     -0.840953
qux    bop     one     -0.390604
               two     -0.242654
dtype: float64

In [44]: s.groupby(level=['first', 'second']).sum()
Out[44]:
first  second
bar    doo      -2.458198
baz    bee       1.444043
foo    bop      -1.263015
qux    bop      -0.633258
dtype: float64
```

New in version 0.20.

Index level names may be supplied as keys.

```
In [45]: s.groupby(['first', 'second']).sum()
Out[45]: 
first  second
bar    doo       -2.458198
baz    bee        1.444043
foo    bop       -1.263015
qux    bop       -0.633258
dtype: float64
```

More on the sum function and aggregation later.

Grouping DataFrame with Index levels and columns

A DataFrame may be grouped by a combination of columns and index levels by specifying the column names as strings and the index levels as pd.Grouper objects.

```
In [46]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....: 

In [47]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [48]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
   ....:                    'B': np.arange(8)},
   ....:                   index=index)
   ....: 

In [49]: df
Out[49]: 
              A  B
first second      
bar   one     1  0
      two     1  1
baz   one     1  2
      two     1  3
foo   one     2  4
      two     2  5
qux   one     3  6
      two     3  7
```

The following example groups df by the second index level and the A column.

```
In [50]: df.groupby([pd.Grouper(level=1), 'A']).sum()
Out[50]: 
          B
second A   
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

Index levels may also be specified by name.

```
In [51]: df.groupby([pd.Grouper(level='second'), 'A']).sum()
Out[51]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

New in version 0.20.

Index level names may be specified as keys directly to `groupby`.

```
In [52]: df.groupby(['second', 'A']).sum()
Out[52]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

DataFrame column selection in GroupBy

Once you have created the GroupBy object from a DataFrame, 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 [53]: grouped = df.groupby(['A'])

In [54]: grouped_C = grouped['C']

In [55]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [56]: df['C'].groupby(df['A'])
Out[56]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x1c398c76d0>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

4.12.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [57]: grouped = df.groupby('A')

In [58]: for name, group in grouped:
   ....:     print(name)
   ....:     print(group)
   ....:
```

(continues on next page)

(continued from previous page)

```
bar
     A      B         C         D
1  bar    one -0.732034  0.531245
3  bar  three  0.503626 -1.293798
5  bar    two  2.485152 -0.028109
foo
     A      B         C         D
0  foo    one  1.018905  1.623114
2  foo    two -0.242912 -1.390677
4  foo    two  0.195596  1.312507
6  foo    one  0.299609  1.312379
7  foo  three  1.073480  1.869813
```

In the case of grouping by multiple keys, the group name will be a tuple:

```
In [59]: for name, group in df.groupby(['A', 'B']):
   ....:     print(name)
   ....:     print(group)
   ....:
('bar', 'one')
     A    B         C         D
1  bar  one -0.732034  0.531245
('bar', 'three')
     A      B         C         D
3  bar  three  0.503626 -1.293798
('bar', 'two')
     A    B         C         D
5  bar  two  2.485152 -0.028109
('foo', 'one')
     A    B         C         D
0  foo  one  1.018905  1.623114
6  foo  one  0.299609  1.312379
('foo', 'three')
     A      B        C         D
7  foo  three  1.07348  1.869813
('foo', 'two')
     A    B         C         D
2  foo  two -0.242912 -1.390677
4  foo  two  0.195596  1.312507
```

See *Iterating through groups*.

4.12.3 Selecting a group

A single group can be selected using `get_group()`:

```
In [60]: grouped.get_group('bar')
Out[60]:
     A      B         C         D
1  bar    one -0.732034  0.531245
3  bar  three  0.503626 -1.293798
5  bar    two  2.485152 -0.028109
```

Or for an object grouped on multiple columns:

```
In [61]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[61]: 
     A    B         C         D
1  bar  one -0.732034  0.531245
```

4.12.4 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. These operations are similar to the *aggregating API*, *window functions API*, and *resample API*.

An obvious one is aggregation via the `aggregate()` or equivalently `agg()` method:

```
In [62]: grouped = df.groupby('A')

In [63]: grouped.aggregate(np.sum)
Out[63]: 
            C         D
A                      
bar  2.256744 -0.790662
foo  2.344678  4.727137

In [64]: grouped = df.groupby(['A', 'B'])

In [65]: grouped.aggregate(np.sum)
Out[65]: 
                  C         D
A   B                        
bar one   -0.732034  0.531245
    three  0.503626 -1.293798
    two    2.485152 -0.028109
foo one    1.318514  2.935493
    three  1.073480  1.869813
    two   -0.047316 -0.078170
```

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 [66]: grouped = df.groupby(['A', 'B'], as_index=False)

In [67]: grouped.aggregate(np.sum)
Out[67]: 
     A      B         C         D
0  bar    one -0.732034  0.531245
1  bar  three  0.503626 -1.293798
2  bar    two  2.485152 -0.028109
3  foo    one  1.318514  2.935493
4  foo  three  1.073480  1.869813
5  foo    two -0.047316 -0.078170

In [68]: df.groupby('A', as_index=False).sum()
Out[68]: 
     A         C         D
0  bar  2.256744 -0.790662
1  foo  2.344678  4.727137
```

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 [69]: df.groupby(['A', 'B']).sum().reset_index()
Out[69]:
     A      B         C         D
0  bar    one -0.732034  0.531245
1  bar  three  0.503626 -1.293798
2  bar    two  2.485152 -0.028109
3  foo    one  1.318514  2.935493
4  foo  three  1.073480  1.869813
5  foo    two -0.047316 -0.078170
```

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 [70]: grouped.size()
Out[70]:
A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

```
In [71]: grouped.describe()
Out[71]:
      C                                            ...         D                          ␣
↪
  count      mean       std       min       25%  ...       min       25%       50%      ␣
↪     75%       max
0   1.0 -0.732034       NaN -0.732034 -0.732034  ...  0.531245  0.531245  0.531245  0.
↪531245  0.531245
1   1.0  0.503626       NaN  0.503626  0.503626  ... -1.293798 -1.293798 -1.293798 -1.
↪293798 -1.293798
2   1.0  2.485152       NaN  2.485152  2.485152  ... -0.028109 -0.028109 -0.028109 -0.
↪028109 -0.028109
3   2.0  0.659257  0.508619  0.299609  0.479433  ...  1.312379  1.390063  1.467747  1.
↪545430  1.623114
4   1.0  1.073480       NaN  1.073480  1.073480  ...  1.869813  1.869813  1.869813  1.
↪869813  1.869813
5   2.0 -0.023658  0.310072 -0.242912 -0.133285  ... -1.390677 -0.714881 -0.039085  0.
↪636711  1.312507

[6 rows x 16 columns]
```

Note: Aggregation functions **will not** return the groups that you are aggregating over if they are named *columns*, when as_index=True, the default. The grouped columns will be the **indices** of the returned object.

Passing as_index=False **will** return the groups that you are aggregating over, if they are named *columns*.

Aggregating functions are the ones that reduce the dimension of the returned objects. Some common aggregating functions are tabulated below:

Function	Description
mean()	Compute mean of groups
sum()	Compute sum of group values
size()	Compute group sizes
count()	Compute count of group
std()	Standard deviation of groups
var()	Compute variance of groups
sem()	Standard error of the mean of groups
describe()	Generates descriptive statistics
first()	Compute first of group values
last()	Compute last of group values
nth()	Take nth value, or a subset if n is a list
min()	Compute min of group values
max()	Compute max of group values

The aggregating functions above will exclude NA values. Any function which reduces a `Series` to a scalar value is an aggregation function and will work, a trivial example is `df.groupby('A').agg(lambda ser: 1)`. Note that `nth()` can act as a reducer *or* a filter, see *here*.

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 [72]: grouped = df.groupby('A')

In [73]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[73]: 
          sum      mean       std
A                                
bar  2.256744  0.752248  1.622939
foo  2.344678  0.468936  0.565255
```

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 [74]: grouped.agg([np.sum, np.mean, np.std])
Out[74]: 
            C                                D                    
          sum      mean       std       sum      mean       std
A                                                              
bar  2.256744  0.752248  1.622939 -0.790662 -0.263554  0.935025
foo  2.344678  0.468936  0.565255  4.727137  0.945427  1.326700
```

The resulting aggregations are named for the functions themselves. If you need to rename, then you can add in a chained operation for a `Series` like this:

```
In [75]: (grouped['C'].agg([np.sum, np.mean, np.std])
   ....:              .rename(columns={'sum': 'foo',
   ....:                               'mean': 'bar',
   ....:                               'std': 'baz'}))
   ....: 
Out[75]: 
          foo       bar       baz
A                                
```

(continues on next page)

(continued from previous page)

```
bar  2.256744  0.752248  1.622939
foo  2.344678  0.468936  0.565255
```

For a grouped DataFrame, you can rename in a similar manner:

```
In [76]: (grouped.agg([np.sum, np.mean, np.std])
   ....:         .rename(columns={'sum': 'foo',
   ....:                          'mean': 'bar',
   ....:                          'std': 'baz'}))
   ....:
Out[76]:
            C                             D
          foo       bar       baz       foo       bar       baz
A
bar  2.256744  0.752248  1.622939 -0.790662 -0.263554  0.935025
foo  2.344678  0.468936  0.565255  4.727137  0.945427  1.326700
```

Note: In general, the output column names should be unique. You cant apply the same function (or two functions with the same name) to the same column.

```
In [77]: grouped['C'].agg(['sum', 'sum'])
---------------------------------------------------------------------------
SpecificationError                        Traceback (most recent call last)
<ipython-input-77-7be02859f395> in <module>
----> 1 grouped['C'].agg(['sum', 'sum'])

~/sandbox/pandas-release/pandas/pandas/core/groupby/generic.py in aggregate(self,␣
↪func_or_funcs, *args, **kwargs)
    849             # but not the class list / tuple itself.
    850             func_or_funcs = _maybe_mangle_lambdas(func_or_funcs)
--> 851             ret = self._aggregate_multiple_funcs(func_or_funcs, (_level or 0)␣
↪+ 1)
    852             if relabeling:
    853                 ret.columns = columns

~/sandbox/pandas-release/pandas/pandas/core/groupby/generic.py in _aggregate_multiple_
↪funcs(self, arg, _level)
    919                 raise SpecificationError(
    920                     "Function names must be unique, found multiple named "
--> 921                     "{}".format(name)
    922                 )
    923

SpecificationError: Function names must be unique, found multiple named sum
```

Pandas *does* allow you to provide multiple lambdas. In this case, pandas will mangle the name of the (nameless) lambda functions, appending _<i> to each subsequent lambda.

```
In [78]: grouped['C'].agg([lambda x: x.max() - x.min(),
   ....:                   lambda x: x.median() - x.mean()])
   ....:
Out[78]:
     <lambda_0>  <lambda_1>
A
bar    3.217186   -0.248622
foo    1.316392   -0.169326
```

Named aggregation

New in version 0.25.0.

To support column-specific aggregation *with control over the output column names*, pandas accepts the special syntax in `GroupBy.agg()`, known as named aggregation, where

- The keywords are the *output* column names
- The values are tuples whose first element is the column to select and the second element is the aggregation to apply to that column. Pandas provides the `pandas.NamedAgg` namedtuple with the fields `['column', 'aggfunc']` to make it clearer what the arguments are. As usual, the aggregation can be a callable or a string alias.

```
In [79]: animals = pd.DataFrame({'kind': ['cat', 'dog', 'cat', 'dog'],
   ....:                         'height': [9.1, 6.0, 9.5, 34.0],
   ....:                         'weight': [7.9, 7.5, 9.9, 198.0]})
   ....:

In [80]: animals
Out[80]:
  kind  height  weight
0  cat     9.1     7.9
1  dog     6.0     7.5
2  cat     9.5     9.9
3  dog    34.0   198.0

In [81]: animals.groupby("kind").agg(
   ....:     min_height=pd.NamedAgg(column='height', aggfunc='min'),
   ....:     max_height=pd.NamedAgg(column='height', aggfunc='max'),
   ....:     average_weight=pd.NamedAgg(column='weight', aggfunc=np.mean),
   ....: )
   ....:
Out[81]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

`pandas.NamedAgg` is just a `namedtuple`. Plain tuples are allowed as well.

```
In [82]: animals.groupby("kind").agg(
   ....:     min_height=('height', 'min'),
   ....:     max_height=('height', 'max'),
   ....:     average_weight=('weight', np.mean),
   ....: )
   ....:
Out[82]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

If your desired output column names are not valid python keywords, construct a dictionary and unpack the keyword arguments

```
In [83]: animals.groupby("kind").agg(**{
   ....:     'total weight': pd.NamedAgg(column='weight', aggfunc=sum),
   ....: })
   ....:
Out[83]:
      total weight
kind
cat           17.8
dog          205.5
```

Additional keyword arguments are not passed through to the aggregation functions. Only pairs of `(column, aggfunc)` should be passed as `**kwargs`. If your aggregation functions requires additional arguments, partially apply them with `functools.partial()`.

Note: For Python 3.5 and earlier, the order of `**kwargs` in a functions was not preserved. This means that the output column ordering would not be consistent. To ensure consistent ordering, the keys (and so output columns) will always be sorted for Python 3.5.

Named aggregation is also valid for Series groupby aggregations. In this case theres no column selection, so the values are just the functions.

```
In [84]: animals.groupby("kind").height.agg(
   ....:     min_height='min',
   ....:     max_height='max',
   ....: )
   ....:
Out[84]:
      min_height  max_height
kind
cat          9.1         9.5
dog          6.0        34.0
```

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 [85]: grouped.agg({'C': np.sum,
   ....:              'D': lambda x: np.std(x, ddof=1)})
   ....:
Out[85]:
            C         D
A
bar  2.256744  0.935025
foo  2.344678  1.326700
```

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 [86]: grouped.agg({'C': 'sum', 'D': 'std'})
Out[86]:
            C         D
A
bar  2.256744  0.935025
foo  2.344678  1.326700
```

Cython-optimized aggregation functions

Some common aggregations, currently only sum, mean, std, and sem, have optimized Cython implementations:

```
In [87]: df.groupby('A').sum()
Out[87]:
            C         D
A
bar  2.256744 -0.790662
foo  2.344678  4.727137

In [88]: df.groupby(['A', 'B']).mean()
Out[88]:
                  C         D
A   B
bar one   -0.732034  0.531245
    three  0.503626 -1.293798
    two    2.485152 -0.028109
foo one    0.659257  1.467747
    three  1.073480  1.869813
    two   -0.023658 -0.039085
```

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).

4.12.5 Transformation

The transform method returns an object that is indexed the same (same size) as the one being grouped. The transform function must:

- Return a result that is either the same size as the group chunk or broadcastable to the size of the group chunk (e.g., a scalar, grouped.transform(lambda x: x.iloc[-1])).
- Operate column-by-column on the group chunk. The transform is applied to the first group chunk using chunk.apply.
- Not perform in-place operations on the group chunk. Group chunks should be treated as immutable, and changes to a group chunk may produce unexpected results. For example, when using fillna, inplace must be False (grouped.transform(lambda x: x.fillna(inplace=False))).
- (Optionally) operates on the entire group chunk. If this is supported, a fast path is used starting from the *second* chunk.

For example, suppose we wished to standardize the data within each group:

```
In [89]: index = pd.date_range('10/1/1999', periods=1100)

In [90]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)

In [91]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()

In [92]: ts.head()
Out[92]:
2000-01-08    0.482962
2000-01-09    0.486928
2000-01-10    0.479684
2000-01-11    0.443526
```

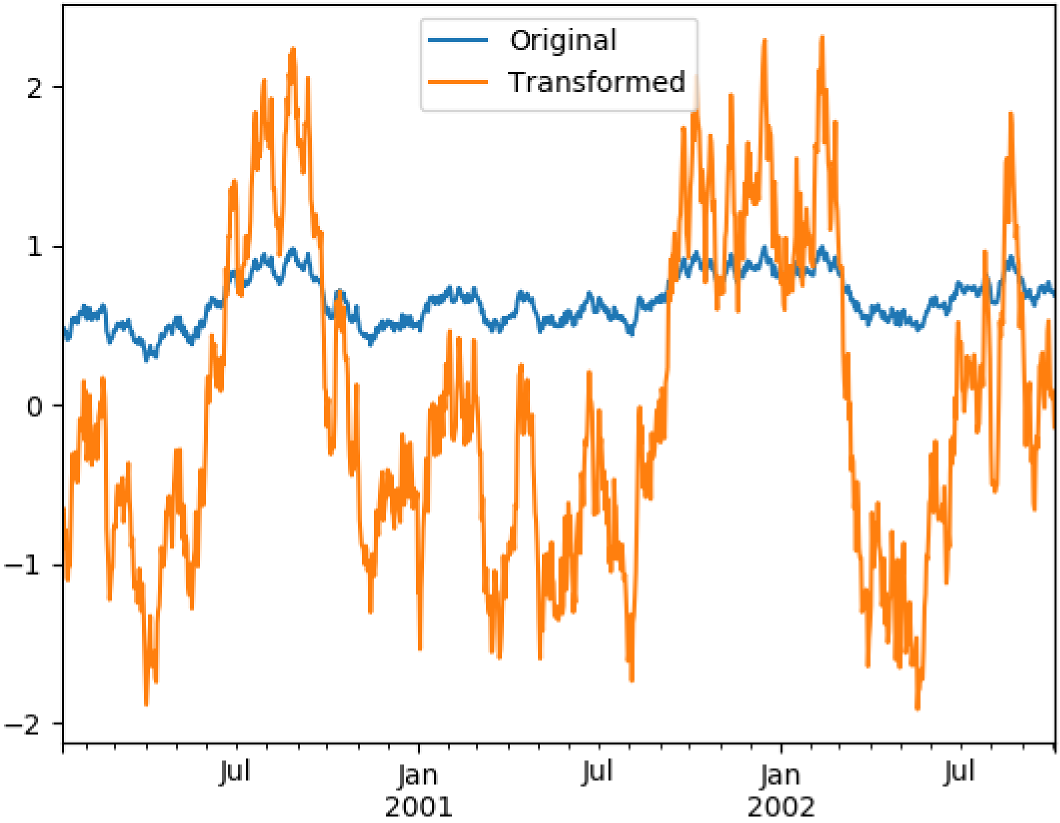
```
2000-01-12    0.462199
Freq: D, dtype: float64

In [93]: ts.tail()
Out[93]: 
2002-09-30    0.712722
2002-10-01    0.717748
2002-10-02    0.710348
2002-10-03    0.712006
2002-10-04    0.688134
Freq: D, dtype: float64

In [94]: transformed = (ts.groupby(lambda x: x.year)
   ....:                  .transform(lambda x: (x - x.mean()) / x.std()))
   ....: 
```

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 [95]: grouped = ts.groupby(lambda x: x.year)

In [96]: grouped.mean()
Out[96]: 
2000    0.597322
2001    0.677152
2002    0.705474
dtype: float64

In [97]: grouped.std()
Out[97]: 
2000    0.171206
2001    0.137719
2002    0.125031
dtype: float64

# Transformed Data
In [98]: grouped_trans = transformed.groupby(lambda x: x.year)

In [99]: grouped_trans.mean()
Out[99]: 
2000    4.221322e-15
2001    5.161168e-15
2002    1.169542e-15
dtype: float64

In [100]: grouped_trans.std()
Out[100]: 
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [101]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})

In [102]: compare.plot()
Out[102]: <matplotlib.axes._subplots.AxesSubplot at 0x1a312e2650>
```



Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```
In [103]: ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
Out[103]:
2000-01-08    0.706553
2000-01-09    0.706553
2000-01-10    0.706553
2000-01-11    0.706553
2000-01-12    0.706553
                ...
2002-09-30    0.528844
2002-10-01    0.528844
2002-10-02    0.528844
2002-10-03    0.528844
2002-10-04    0.528844
Freq: D, Length: 1001, dtype: float64
```

Alternatively, the built-in methods could be used to produce the same outputs.

```
In [104]: max = ts.groupby(lambda x: x.year).transform('max')

In [105]: min = ts.groupby(lambda x: x.year).transform('min')

In [106]: max - min
Out[106]: 
2000-01-08    0.706553
2000-01-09    0.706553
2000-01-10    0.706553
2000-01-11    0.706553
2000-01-12    0.706553
                ...   
2002-09-30    0.528844
2002-10-01    0.528844
2002-10-02    0.528844
2002-10-03    0.528844
2002-10-04    0.528844
Freq: D, Length: 1001, dtype: float64
```

Another common data transform is to replace missing data with the group mean.

```
In [107]: data_df
Out[107]: 
            A         B         C
0    0.869121  1.473802  0.787744
1    0.464706  1.323886 -1.113432
2   -0.777595  0.143230 -0.718546
3   -0.501424 -2.031157       NaN
4   -0.489464  0.187423 -1.036766
..        ...       ...       ...
995  1.742327 -0.883907 -0.848783
996  0.168746 -1.180773       NaN
997       NaN -0.678201  0.080136
998 -1.389014 -0.372892  0.572905
999 -0.858635 -1.784332       NaN

[1000 rows x 3 columns]

In [108]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [109]: key = countries[np.random.randint(0, 4, 1000)]

In [110]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [111]: grouped.count()
Out[111]: 
      A    B    C
GR  244  256  212
JP  213  223  201
UK  224  242  211
US  221  231  199

In [112]: transformed = grouped.transform(lambda x: x.fillna(x.mean()))
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.

```
In [113]: grouped_trans = transformed.groupby(key)

In [114]: grouped.mean()  # original group means
Out[114]: 
           A         B         C
GR -0.058842  0.078292 -0.081559
JP  0.108123 -0.038631  0.087081
UK  0.130078  0.067902 -0.076353
US  0.002237  0.048024  0.092084

In [115]: grouped_trans.mean()  # transformation did not change group means
Out[115]: 
           A         B         C
GR -0.058842  0.078292 -0.081559
JP  0.108123 -0.038631  0.087081
UK  0.130078  0.067902 -0.076353
US  0.002237  0.048024  0.092084

In [116]: grouped.count()  # original has some missing data points
Out[116]: 
      A    B    C
GR  244  256  212
JP  213  223  201
UK  224  242  211
US  221  231  199

In [117]: grouped_trans.count()  # counts after transformation
Out[117]: 
      A    B    C
GR  266  266  266
JP  235  235  235
UK  252  252  252
US  247  247  247

In [118]: grouped_trans.size()  # Verify non-NA count equals group size
Out[118]: 
GR    266
JP    235
UK    252
US    247
dtype: int64
```

Note: Some functions will automatically transform the input when applied to a GroupBy object, but returning an object of the same shape as the original. Passing `as_index=False` will not affect these transformation methods.

For example: `fillna, ffill, bfill, shift.`.

```
In [119]: grouped.ffill()
Out[119]: 
            A         B         C
0    0.869121  1.473802  0.787744
1    0.464706  1.323886 -1.113432
2   -0.777595  0.143230 -0.718546
3   -0.501424 -2.031157 -0.718546
```

(continues on next page)

(continued from previous page)

```
4   -0.489464  0.187423 -1.036766
..        ...       ...       ...
995  1.742327 -0.883907 -0.848783
996  0.168746 -1.180773 -0.838714
997 -1.159006 -0.678201  0.080136
998 -1.389014 -0.372892  0.572905
999 -0.858635 -1.784332 -0.848783

[1000 rows x 3 columns]
```

New syntax to window and resample operations

New in version 0.18.1.

Working with the resample, expanding or rolling operations on the groupby level used to require the application of helper functions. However, now it is possible to use `resample()`, `expanding()` and `rolling()` as methods on groupbys.

The example below will apply the `rolling()` method on the samples of the column B based on the groups of column A.

```
In [120]: df_re = pd.DataFrame({'A': [1] * 10 + [5] * 10,
   .....:                       'B': np.arange(20)})
   .....:

In [121]: df_re
Out[121]:
    A   B
0   1   0
1   1   1
2   1   2
3   1   3
4   1   4
5   1   5
6   1   6
7   1   7
8   1   8
9   1   9
10  5  10
11  5  11
12  5  12
13  5  13
14  5  14
15  5  15
16  5  16
17  5  17
18  5  18
19  5  19

In [122]: df_re.groupby('A').rolling(4).B.mean()
Out[122]:
A
1  0      NaN
```

```
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
   7      5.5
   8      6.5
   9      7.5
5  10     NaN
   11     NaN
   12     NaN
   13    11.5
   14    12.5
   15    13.5
   16    14.5
   17    15.5
   18    16.5
   19    17.5
Name: B, dtype: float64
```

The `expanding()` method will accumulate a given operation (`sum()` in the example) for all the members of each particular group.

```
In [123]: df_re.groupby('A').expanding().sum()
Out[123]:
         A      B
A
1 0    1.0    0.0
  1    2.0    1.0
  2    3.0    3.0
  3    4.0    6.0
  4    5.0   10.0
  5    6.0   15.0
  6    7.0   21.0
  7    8.0   28.0
  8    9.0   36.0
  9   10.0   45.0
5 10   5.0   10.0
  11  10.0   21.0
  12  15.0   33.0
  13  20.0   46.0
  14  25.0   60.0
  15  30.0   75.0
  16  35.0   91.0
  17  40.0  108.0
  18  45.0  126.0
  19  50.0  145.0
```

Suppose you want to use the `resample()` method to get a daily frequency in each group of your dataframe and wish to complete the missing values with the `ffill()` method.

```
In [124]: df_re = pd.DataFrame({'date': pd.date_range(start='2016-01-01',␣
 ↪periods=4,
   .....:                                                 freq='W'),
   .....:                             'group': [1, 1, 2, 2],
   .....:                             'val': [5, 6, 7, 8]}).set_index('date')
   .....:

In [125]: df_re
Out[125]:
            group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

In [126]: df_re.groupby('group').resample('1D').ffill()
Out[126]:
                  group  val
group date
1     2016-01-03      1    5
      2016-01-04      1    5
      2016-01-05      1    5
      2016-01-06      1    5
      2016-01-07      1    5
      2016-01-08      1    5
      2016-01-09      1    5
      2016-01-10      1    6
2     2016-01-17      2    7
      2016-01-18      2    7
      2016-01-19      2    7
      2016-01-20      2    7
      2016-01-21      2    7
      2016-01-22      2    7
      2016-01-23      2    7
      2016-01-24      2    8
```

4.12.6 Filtration

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 [127]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [128]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[128]:
3    3
4    3
5    3
dtype: int64
```

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

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

```
In [129]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [130]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[130]:
```

(continues on next page)

(continued from previous page)

```
   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 [131]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[131]:
     A    B
0  NaN  NaN
1  NaN  NaN
2  2.0    b
3  3.0    b
4  4.0    b
5  5.0    b
6  NaN  NaN
7  NaN  NaN
```

For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [132]: dff['C'] = np.arange(8)

In [133]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[133]:
   A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
```

Note: Some functions when applied to a groupby object will act as a **filter** on the input, returning a reduced shape of the original (and potentially eliminating groups), but with the index unchanged. Passing `as_index=False` will not affect these transformation methods.

For example: `head, tail`.

```
In [134]: dff.groupby('B').head(2)
Out[134]:
   A  B  C
0  0  a  0
1  1  a  1
2  2  b  2
3  3  b  3
6  6  c  6
7  7  c  7
```

4.12.7 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```
In [135]: grouped = df.groupby('A')

In [136]: grouped.agg(lambda x: x.std())
Out[136]:
            C         D
A
bar  1.622939  0.935025
foo  0.565255  1.326700
```

But, its 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 [137]: grouped.std()
Out[137]:
            C         D
A
bar  1.622939  0.935025
foo  0.565255  1.326700
```

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 [138]: tsdf = pd.DataFrame(np.random.randn(1000, 3),
   .....:                     index=pd.date_range('1/1/2000', periods=1000),
   .....:                     columns=['A', 'B', 'C'])
   .....:

In [139]: tsdf.iloc[::2] = np.nan

In [140]: grouped = tsdf.groupby(lambda x: x.year)

In [141]: grouped.fillna(method='pad')
Out[141]:
                   A         B         C
2000-01-01       NaN       NaN       NaN
2000-01-02 -0.183449 -0.611494 -0.178486
2000-01-03 -0.183449 -0.611494 -0.178486
2000-01-04 -2.105115 -1.569569  0.553558
2000-01-05 -2.105115 -1.569569  0.553558
...              ...       ...       ...
2002-09-22 -1.138283  1.475569 -0.063934
2002-09-23 -1.138283  1.475569 -0.063934
2002-09-24  0.069982  1.189623 -0.212112
2002-09-25  0.069982  1.189623 -0.212112
2002-09-26 -1.601811  0.215340  1.432049

[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

The `nlargest` and `nsmallest` methods work on `Series` style groupbys:

```
In [142]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])

In [143]: g = pd.Series(list('abababab'))
In [144]: gb = s.groupby(g)

In [145]: gb.nlargest(3)
Out[145]:
a  4    19.0
   0     9.0
   2     7.0
b  1     8.0
   3     5.0
   7     3.3
dtype: float64

In [146]: gb.nsmallest(3)
Out[146]:
a  6    4.2
   2    7.0
   0    9.0
b  5    1.0
   7    3.3
   3    5.0
dtype: float64
```

4.12.8 Flexible apply

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want GroupBy to infer how to combine the results. For these, use the `apply` function, which can be substituted for both `aggregate` and `transform` in many standard use cases. However, `apply` can handle some exceptional use cases, for example:

```
In [147]: df
Out[147]:
     A      B         C         D
0  foo    one  1.018905  1.623114
1  bar    one -0.732034  0.531245
2  foo    two -0.242912 -1.390677
3  bar  three  0.503626 -1.293798
4  foo    two  0.195596  1.312507
5  bar    two  2.485152 -0.028109
6  foo    one  0.299609  1.312379
7  foo  three  1.073480  1.869813

In [148]: grouped = df.groupby('A')

# could also just call .describe()
In [149]: grouped['C'].apply(lambda x: x.describe())
Out[149]:
A
bar  count    3.000000
     mean     0.752248
     std      1.622939
     min     -0.732034
     25%     -0.114204
     50%      0.503626
     75%      1.494389
```

(continues on next page)

(continued from previous page)

```
     max       2.485152
foo  count     5.000000
     mean      0.468936
     std       0.565255
     min      -0.242912
     25%       0.195596
     50%       0.299609
     75%       1.018905
     max       1.073480
Name: C, dtype: float64
```

The dimension of the returned result can also change:

```
In [150]: grouped = df.groupby('A')['C']

In [151]: def f(group):
   .....:     return pd.DataFrame({'original': group,
   .....:                          'demeaned': group - group.mean()})
   .....:

In [152]: grouped.apply(f)
Out[152]:
   original  demeaned
0  1.018905  0.549970
1 -0.732034 -1.484282
2 -0.242912 -0.711848
3  0.503626 -0.248622
4  0.195596 -0.273340
5  2.485152  1.732904
6  0.299609 -0.169326
7  1.073480  0.604544
```

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

In [154]: s = pd.Series(np.random.rand(5))

In [155]: s
Out[155]:
0    0.656427
1    0.691656
2    0.352607
3    0.059733
4    0.451923
dtype: float64

In [156]: s.apply(f)
Out[156]:
          x       x^2
0  0.656427  0.430896
1  0.691656  0.478388
2  0.352607  0.124332
```

```
3  0.059733  0.003568
4  0.451923  0.204234
```

Note: apply can act as a reducer, transformer, *or* filter function, depending on exactly what is passed to it. So depending on the path taken, and exactly what you are grouping. Thus the grouped columns(s) may be included in the output as well as set the indices.

4.12.9 Other useful features

Automatic exclusion of nuisance columns

Again consider the example DataFrame weve been looking at:

```
In [157]: df
Out[157]:
     A      B         C         D
0  foo    one  1.018905  1.623114
1  bar    one -0.732034  0.531245
2  foo    two -0.242912 -1.390677
3  bar  three  0.503626 -1.293798
4  foo    two  0.195596  1.312507
5  bar    two  2.485152 -0.028109
6  foo    one  0.299609  1.312379
7  foo  three  1.073480  1.869813
```

Suppose we wish to compute the standard deviation grouped by the A column. There is a slight problem, namely that we dont care about the data in column B. We refer to this as a nuisance column. If the passed aggregation function cant be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [158]: df.groupby('A').std()
Out[158]:
            C         D
A
bar  1.622939  0.935025
foo  0.565255  1.326700
```

Note that df.groupby('A').colname.std(). is more efficient than df.groupby('A').std().colname, so if the result of an aggregation function is only interesting over one column (here colname), it may be filtered *before* applying the aggregation function.

Note: Any object column, also if it contains numerical values such as Decimal objects, is considered as a nuisance columns. They are excluded from aggregate functions automatically in groupby.

If you do wish to include decimal or object columns in an aggregation with other non-nuisance data types, you must do so explicitly.

```
In [159]: from decimal import Decimal

In [160]: df_dec = pd.DataFrame(
   .....:     {'id': [1, 2, 1, 2],
   .....:      'int_column': [1, 2, 3, 4],
   .....:      'dec_column': [Decimal('0.50'), Decimal('0.15'),
   .....:                          Decimal('0.25'), Decimal('0.40')]
   .....:        }
   .....: )
   .....:

# Decimal columns can be sum'd explicitly by themselves...
In [161]: df_dec.groupby(['id'])[['dec_column']].sum()
Out[161]:
   dec_column
id
1        0.75
2        0.55

# ...but cannot be combined with standard data types or they will be excluded
In [162]: df_dec.groupby(['id'])[['int_column', 'dec_column']].sum()
Out[162]:
    int_column
id
1            4
2            6

# Use .agg function to aggregate over standard and "nuisance" data types
# at the same time
In [163]: df_dec.groupby(['id']).agg({'int_column': 'sum', 'dec_column':␣
↪'sum'})
Out[163]:
    int_column dec_column
id
1            4       0.75
2            6       0.55
```

Handling of (un)observed Categorical values

When using a `Categorical` grouper (as a single grouper, or as part of multiple groupers), the `observed` keyword controls whether to return a cartesian product of all possible groupers values (`observed=False`) or only those that are observed groupers (`observed=True`).

Show all values:

```
In [164]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                              categories=['a', 'b']),
   .....:                               observed=False).count()
   .....:
Out[164]:
a    3
b    0
dtype: int64
```

Show only the observed values:

```
In [165]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                              categories=['a', 'b']),
   .....:                               observed=True).count()
   .....:
```

(continues on next page)

(continued from previous page)

```
Out[165]:
a    3
dtype: int64
```

The returned dtype of the grouped will *always* include *all* of the categories that were grouped.

```
In [166]: s = pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                                 categories=['a', 'b']),
   .....:                                  observed=False).count()
   .....:

In [167]: s.index.dtype
Out[167]: CategoricalDtype(categories=['a', 'b'], ordered=False)
```

NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. In other words, there will never be an NA group or NaT group. This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

Grouping with ordered factors

Categorical variables represented as instance of pandass `Categorical` class can be used as group keys. If so, the order of the levels will be preserved:

```
In [168]: data = pd.Series(np.random.randn(100))

In [169]: factor = pd.qcut(data, [0, .25, .5, .75, 1.])

In [170]: data.groupby(factor).mean()
Out[170]:
(-3.221, -0.48]   -1.238032
(-0.48, 0.119]    -0.216324
(0.119, 0.702]     0.381676
(0.702, 2.538]     1.216886
dtype: float64
```

Grouping with a grouper specification

You may need to specify a bit more data to properly group. You can use the `pd.Grouper` to provide this local control.

```
In [171]: import datetime

In [172]: df = pd.DataFrame({'Branch': 'A A A A A A A B'.split(),
   .....:                     'Buyer': 'Carl Mark Carl Carl Joe Joe Joe Carl'.split(),
   .....:                     'Quantity': [1, 3, 5, 1, 8, 1, 9, 3],
   .....:                     'Date': [
   .....:                         datetime.datetime(2013, 1, 1, 13, 0),
   .....:                         datetime.datetime(2013, 1, 1, 13, 5),
   .....:                         datetime.datetime(2013, 10, 1, 20, 0),
   .....:                         datetime.datetime(2013, 10, 2, 10, 0),
```

(continues on next page)

(continued from previous page)

```
   .....:                                datetime.datetime(2013, 10, 1, 20, 0),
   .....:                                datetime.datetime(2013, 10, 2, 10, 0),
   .....:                                datetime.datetime(2013, 12, 2, 12, 0),
   .....:                                datetime.datetime(2013, 12, 2, 14, 0)]
   .....:                            })
   .....:

In [173]: df
Out[173]:
  Branch Buyer  Quantity                Date
0      A  Carl         1 2013-01-01 13:00:00
1      A  Mark         3 2013-01-01 13:05:00
2      A  Carl         5 2013-10-01 20:00:00
3      A  Carl         1 2013-10-02 10:00:00
4      A   Joe         8 2013-10-01 20:00:00
5      A   Joe         1 2013-10-02 10:00:00
6      A   Joe         9 2013-12-02 12:00:00
7      B  Carl         3 2013-12-02 14:00:00
```

Groupby a specific column with the desired frequency. This is like resampling.

```
In [174]: df.groupby([pd.Grouper(freq='1M', key='Date'), 'Buyer']).sum()
Out[174]:
                  Quantity
Date       Buyer
2013-01-31 Carl          1
           Mark          3
2013-10-31 Carl          6
           Joe           9
2013-12-31 Carl          3
           Joe           9
```

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

```
In [175]: df = df.set_index('Date')

In [176]: df['Date'] = df.index + pd.offsets.MonthEnd(2)

In [177]: df.groupby([pd.Grouper(freq='6M', key='Date'), 'Buyer']).sum()
Out[177]:
                  Quantity
Date       Buyer
2013-02-28 Carl          1
           Mark          3
2014-02-28 Carl          9
           Joe          18

In [178]: df.groupby([pd.Grouper(freq='6M', level='Date'), 'Buyer']).sum()
Out[178]:
                  Quantity
Date       Buyer
2013-01-31 Carl          1
           Mark          3
2014-01-31 Carl          9
           Joe          18
```

Taking the first rows of each group

Just like for a DataFrame or Series you can call head and tail on a groupby:

```
In [179]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [180]: df
Out[180]:
   A  B
0  1  2
1  1  4
2  5  6

In [181]: g = df.groupby('A')

In [182]: g.head(1)
Out[182]:
   A  B
0  1  2
2  5  6

In [183]: g.tail(1)
Out[183]:
   A  B
1  1  4
2  5  6
```

This shows the first or last n rows from each group.

Taking the nth row of each group

To select from a DataFrame or Series the nth item, use `nth()`. This is a reduction method, and will return a single row (or no row) per group if you pass an int for n:

```
In [184]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [185]: g = df.groupby('A')

In [186]: g.nth(0)
Out[186]:
     B
A
1  NaN
5  6.0

In [187]: g.nth(-1)
Out[187]:
     B
A
1  4.0
5  6.0

In [188]: g.nth(1)
Out[188]:
     B
```

```
A
1  4.0
```

If you want to select the nth not-null item, use the `dropna` kwarg. For a DataFrame this should be either `'any'` or `'all'` just like you would pass to dropna:

```
# nth(0) is the same as g.first()
In [189]: g.nth(0, dropna='any')
Out[189]:
     B
A
1  4.0
5  6.0

In [190]: g.first()
Out[190]:
     B
A
1  4.0
5  6.0

# nth(-1) is the same as g.last()
In [191]: g.nth(-1, dropna='any')  # NaNs denote group exhausted when using␣
↪dropna
Out[191]:
     B
A
1  4.0
5  6.0

In [192]: g.last()
Out[192]:
     B
A
1  4.0
5  6.0

In [193]: g.B.nth(0, dropna='all')
Out[193]:
A
1    4.0
5    6.0
Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```
In [194]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [195]: g = df.groupby('A', as_index=False)

In [196]: g.nth(0)
Out[196]:
   A    B
0  1  NaN
2  5  6.0
```

```
In [197]: g.nth(-1)
Out[197]:
   A    B
1  1  4.0
2  5  6.0
```

You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

```
In [198]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [199]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [200]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[200]:
        a  b
2014 4  1  1
     4  1  1
     4  1  1
     5  1  1
     5  1  1
     5  1  1
     6  1  1
     6  1  1
     6  1  1
```

Enumerate group items

To see the order in which each row appears within its group, use the cumcount method:

```
In [201]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [202]: dfg
Out[202]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [203]: dfg.groupby('A').cumcount()
Out[203]:
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64

In [204]: dfg.groupby('A').cumcount(ascending=False)
Out[204]:
0    3
```

```
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

Enumerate groups

New in version 0.20.2.

To see the ordering of the groups (as opposed to the order of rows within a group given by `cumcount`) you can use `ngroup()`.

Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

```
In [205]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [206]: dfg
Out[206]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [207]: dfg.groupby('A').ngroup()
Out[207]:
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64

In [208]: dfg.groupby('A').ngroup(ascending=False)
Out[208]:
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
```

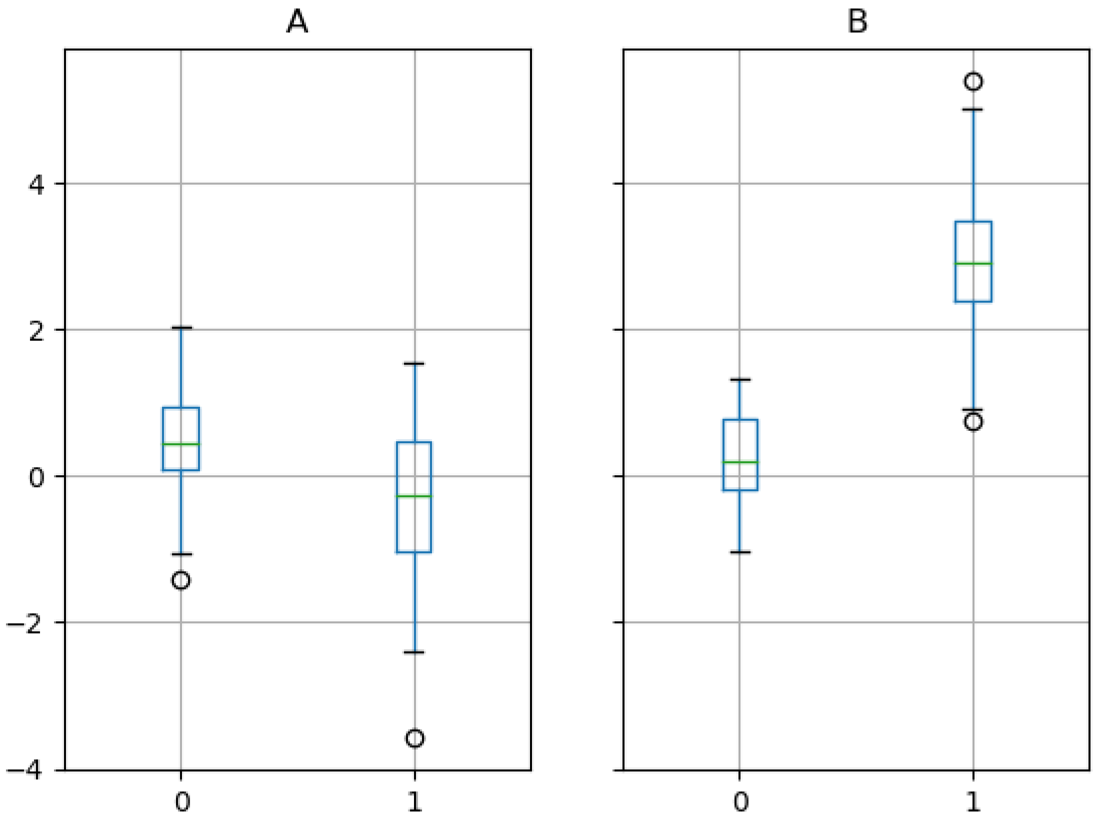
Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is B are 3 higher on average.

```
In [209]: np.random.seed(1234)

In [210]: df = pd.DataFrame(np.random.randn(50, 2))

In [211]: df['g'] = np.random.choice(['A', 'B'], size=50)

In [212]: df.loc[df['g'] == 'B', 1] += 3
```

We can easily visualize this with a boxplot:

```
In [213]: df.groupby('g').boxplot()
Out[213]:
A         AxesSubplot(0.1,0.15;0.363636x0.75)
B    AxesSubplot(0.536364,0.15;0.363636x0.75)
dtype: object
```



The result of calling `boxplot` is a dictionary whose keys are the values of our grouping column `g` (A and B). The values of the resulting dictionary can be controlled by the `return_type` keyword of `boxplot`. See the *visualization documentation* for more.

Warning: For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See *here* for an explanation.

Piping function calls

New in version 0.21.0.

Similar to the functionality provided by `DataFrame` and `Series`, functions that take `GroupBy` objects can be chained together using a `pipe` method to allow for a cleaner, more readable syntax. To read about `.pipe` in general terms, see *here*.

Combining `.groupby` and `.pipe` is often useful when you need to reuse GroupBy objects.

As an example, imagine having a DataFrame with columns for stores, products, revenue and quantity sold. Wed like to do a groupwise calculation of *prices* (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable. First we set the data:

```
In [214]: n = 1000

In [215]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
   .....:                    'Product': np.random.choice(['Product_1',
   .....:                                                 'Product_2'], n),
   .....:                    'Revenue': (np.random.random(n) * 50 + 10).round(2),
   .....:                    'Quantity': np.random.randint(1, 10, size=n)})
   .....:

In [216]: df.head(2)
Out[216]:
     Store    Product  Revenue  Quantity
0  Store_2  Product_1    26.12         1
1  Store_2  Product_1    28.86         1
```

Now, to find prices per store/product, we can simply do:

```
In [217]: (df.groupby(['Store', 'Product'])
   .....:    .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   .....:    .unstack().round(2))
   .....:
Out[217]:
Product  Product_1  Product_2
Store
Store_1       6.82       7.05
Store_2       6.30       6.64
```

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

```
In [218]: def mean(groupby):
   .....:     return groupby.mean()
   .....:

In [219]: df.groupby(['Store', 'Product']).pipe(mean)
Out[219]:
                     Revenue  Quantity
Store   Product
Store_1 Product_1  34.622727  5.075758
        Product_2  35.482815  5.029630
Store_2 Product_1  32.972837  5.237589
        Product_2  34.684360  5.224000
```

where `mean` takes a GroupBy object and finds the mean of the Revenue and Quantity columns respectively for each Store-Product combination. The `mean` function can be any function that takes in a GroupBy object; the `.pipe` will pass the GroupBy object as a parameter into the function you specify.

4.12.10 Examples

Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

```
In [220]: df = pd.DataFrame({'a': [1, 0, 0], 'b': [0, 1, 0],
   .....:                    'c': [1, 0, 0], 'd': [2, 3, 4]})
   .....:

In [221]: df
Out[221]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4

In [222]: df.groupby(df.sum(), axis=1).sum()
Out[222]:
   1  9
0  2  2
1  1  3
2  0  4
```

Multi-column factorization

By using `ngroup()`, we can extract information about the groups in a way similar to `factorize()` (as described further in the *reshaping API*) but which applies naturally to multiple columns of mixed type and different sources. This can be useful as an intermediate categorical-like step in processing, when the relationships between the group rows are more important than their content, or as input to an algorithm which only accepts the integer encoding. (For more information about support in pandas for full categorical data, see the *Categorical introduction* and the *API documentation*.)

```
In [223]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})

In [224]: dfg
Out[224]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a

In [225]: dfg.groupby(["A", "B"]).ngroup()
Out[225]:
0    0
1    0
2    1
3    2
4    1
dtype: int64

In [226]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
```

```
Out[226]:
0    0
1    0
2    1
3    3
4    2
dtype: int64
```

Groupby by indexer to resample data

Resampling produces new hypothetical samples (resamples) from already existing observed data or from a model that generates data. These new samples are similar to the pre-existing samples.

In order to resample to work on indices that are non-datetimelike, the following procedure can be utilized.

In the following examples, **df.index // 5** returns a binary array which is used to determine what gets selected for the groupby operation.

Note: The below example shows how we can downsample by consolidation of samples into fewer samples. Here by using **df.index // 5**, we are aggregating the samples in bins. By applying **std()** function, we aggregate the information contained in many samples into a small subset of values which is their standard deviation thereby reducing the number of samples.

```
In [227]: df = pd.DataFrame(np.random.randn(10, 2))

In [228]: df
Out[228]:
          0         1
0 -0.793893  0.321153
1  0.342250  1.618906
2 -0.975807  1.918201
3 -0.810847 -1.405919
4 -1.977759  0.461659
5  0.730057 -1.316938
6 -0.751328  0.528290
7 -0.257759 -1.081009
8  0.505895 -1.701948
9 -1.006349  0.020208

In [229]: df.index // 5
Out[229]: Int64Index([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype='int64')

In [230]: df.groupby(df.index // 5).std()
Out[230]:
          0         1
0  0.823647  1.312912
1  0.760109  0.942941
```

Returning a Series to propagate names

Group DataFrame columns, compute a set of metrics and return a named Series. The Series name is used as the name for the column index. This is especially useful in conjunction with reshaping operations such as stacking in which the column index name will be used as the name of the inserted column:

```
In [231]: df = pd.DataFrame({'a': [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2],
   .....:                    'b': [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
   .....:                    'c': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
   .....:                    'd': [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]})
   .....:

In [232]: def compute_metrics(x):
   .....:     result = {'b_sum': x['b'].sum(), 'c_mean': x['c'].mean()}
   .....:     return pd.Series(result, name='metrics')
   .....:

In [233]: result = df.groupby('a').apply(compute_metrics)

In [234]: result
Out[234]:
metrics  b_sum  c_mean
a
0          2.0     0.5
1          2.0     0.5
2          2.0     0.5

In [235]: result.stack()
Out[235]:
a  metrics
0  b_sum      2.0
   c_mean     0.5
1  b_sum      2.0
   c_mean     0.5
2  b_sum      2.0
   c_mean     0.5
dtype: float64
```

{{ header }}

4.13 Time series / date functionality

pandas contains extensive capabilities and features for working with time series data for all domains. Using the NumPy `datetime64` and `timedelta64` dtypes, pandas has 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.

For example, pandas supports:

Parsing time series information from various sources and formats

```
In [1]: import datetime

In [2]: dti = pd.to_datetime(['1/1/2018', np.datetime64('2018-01-01'),
   ...:                       datetime.datetime(2018, 1, 1)])
   ...:

In [3]: dti
Out[3]: DatetimeIndex(['2018-01-01', '2018-01-01', '2018-01-01'], dtype=
↪'datetime64[ns]', freq=None)
```

Generate sequences of fixed-frequency dates and time spans

```
In [4]: dti = pd.date_range('2018-01-01', periods=3, freq='H')

In [5]: dti
Out[5]:
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 01:00:00',
               '2018-01-01 02:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Manipulating and converting date times with timezone information

```
In [6]: dti = dti.tz_localize('UTC')

In [7]: dti
Out[7]:
DatetimeIndex(['2018-01-01 00:00:00+00:00', '2018-01-01 01:00:00+00:00',
               '2018-01-01 02:00:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='H')

In [8]: dti.tz_convert('US/Pacific')
Out[8]:
DatetimeIndex(['2017-12-31 16:00:00-08:00', '2017-12-31 17:00:00-08:00',
               '2017-12-31 18:00:00-08:00'],
              dtype='datetime64[ns, US/Pacific]', freq='H')
```

Resampling or converting a time series to a particular frequency

```
In [9]: idx = pd.date_range('2018-01-01', periods=5, freq='H')

In [10]: ts = pd.Series(range(len(idx)), index=idx)

In [11]: ts
Out[11]:
2018-01-01 00:00:00    0
2018-01-01 01:00:00    1
2018-01-01 02:00:00    2
2018-01-01 03:00:00    3
2018-01-01 04:00:00    4
Freq: H, dtype: int64

In [12]: ts.resample('2H').mean()
Out[12]:
2018-01-01 00:00:00    0.5
2018-01-01 02:00:00    2.5
2018-01-01 04:00:00    4.0
Freq: 2H, dtype: float64
```

Performing date and time arithmetic with absolute or relative time increments

```
In [13]: friday = pd.Timestamp('2018-01-05')

In [14]: friday.day_name()
Out[14]: 'Friday'

# Add 1 day
In [15]: saturday = friday + pd.Timedelta('1 day')
```

(continues on next page)

(continued from previous page)

```
In [16]: saturday.day_name()
Out[16]: 'Saturday'

# Add 1 business day (Friday --> Monday)
In [17]: monday = friday + pd.offsets.BDay()

In [18]: monday.day_name()
Out[18]: 'Monday'
```

pandas provides a relatively compact and self-contained set of tools for performing the above tasks and more.

4.13.1 Overview

pandas captures 4 general time related concepts:

1. Date times: A specific date and time with timezone support. Similar to `datetime.datetime` from the standard library.
2. Time deltas: An absolute time duration. Similar to `datetime.timedelta` from the standard library.
3. Time spans: A span of time defined by a point in time and its associated frequency.
4. Date offsets: A relative time duration that respects calendar arithmetic. Similar to `dateutil.relativedelta.relativedelta` from the `dateutil` package.

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`
Time spans	`Period`	`PeriodIndex`	`period[freq]`	`Period` or `period_range`
Date offsets	`DateOffset`	`None`	`None`	`DateOffset`

For time series data, its conventional to represent the time component in the index of a `Series` or `DataFrame` so manipulations can be performed with respect to the time element.

```
In [19]: pd.Series(range(3), index=pd.date_range('2000', freq='D', periods=3))
Out[19]:
2000-01-01    0
2000-01-02    1
2000-01-03    2
Freq: D, dtype: int64
```

However, `Series` and `DataFrame` can directly also support the time component as data itself.

```
In [20]: pd.Series(pd.date_range('2000', freq='D', periods=3))
Out[20]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
dtype: datetime64[ns]
```

Series and DataFrame have extended data type support and functionality for datetime, timedelta and Period data when passed into those constructors. DateOffset data however will be stored as object data.

```
In [21]: pd.Series(pd.period_range('1/1/2011', freq='M', periods=3))
Out[21]:
0    2011-01
1    2011-02
2    2011-03
dtype: period[M]

In [22]: pd.Series([pd.DateOffset(1), pd.DateOffset(2)])
Out[22]:
0         <DateOffset>
1    <2 * DateOffsets>
dtype: object

In [23]: pd.Series(pd.date_range('1/1/2011', freq='M', periods=3))
Out[23]:
0   2011-01-31
1   2011-02-28
2   2011-03-31
dtype: datetime64[ns]
```

Lastly, pandas represents null date times, time deltas, and time spans as NaT which is useful for representing missing or null date like values and behaves similar as np.nan does for float data.

```
In [24]: pd.Timestamp(pd.NaT)
Out[24]: NaT

In [25]: pd.Timedelta(pd.NaT)
Out[25]: NaT

In [26]: pd.Period(pd.NaT)
Out[26]: NaT

# Equality acts as np.nan would
In [27]: pd.NaT == pd.NaT
Out[27]: False
```

4.13.2 Timestamps vs. Time Spans

Timestamped data is the most basic type of time series data that associates values with points in time. For pandas objects it means using the points in time.

```
In [28]: pd.Timestamp(datetime.datetime(2012, 5, 1))
Out[28]: Timestamp('2012-05-01 00:00:00')

In [29]: pd.Timestamp('2012-05-01')
Out[29]: Timestamp('2012-05-01 00:00:00')

In [30]: pd.Timestamp(2012, 5, 1)
Out[30]: Timestamp('2012-05-01 00:00:00')
```

However, in many cases it is more natural to associate things like change variables with a time span instead. The span represented by Period can be specified explicitly, or inferred from datetime string format.

For example:

```
In [31]: pd.Period('2011-01')
Out[31]: Period('2011-01', 'M')

In [32]: pd.Period('2012-05', freq='D')
Out[32]: Period('2012-05-01', 'D')
```

Timestamp and Period can serve as an index. Lists of Timestamp and Period are automatically coerced to DatetimeIndex and PeriodIndex respectively.

```
In [33]: dates = [pd.Timestamp('2012-05-01'),
   ....:          pd.Timestamp('2012-05-02'),
   ....:          pd.Timestamp('2012-05-03')]
   ....:

In [34]: ts = pd.Series(np.random.randn(3), dates)

In [35]: type(ts.index)
Out[35]: pandas.core.indexes.datetimes.DatetimeIndex

In [36]: ts.index
Out[36]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'],␣
↪dtype='datetime64[ns]', freq=None)

In [37]: ts
Out[37]:
2012-05-01    1.212707
2012-05-02   -2.219105
2012-05-03    0.930394
dtype: float64

In [38]: periods = [pd.Period('2012-01'), pd.Period('2012-02'), pd.
↪Period('2012-03')]

In [39]: ts = pd.Series(np.random.randn(3), periods)

In [40]: type(ts.index)
Out[40]: pandas.core.indexes.period.PeriodIndex

In [41]: ts.index
Out[41]: PeriodIndex(['2012-01', '2012-02', '2012-03'], dtype='period[M]',␣
↪freq='M')

In [42]: ts
Out[42]:
2012-01    0.703754
2012-02    0.580511
2012-03   -0.135776
Freq: M, dtype: float64
```

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.

4.13.3 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 [43]: pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))
Out[43]:
0   2009-07-31
1   2010-01-10
2          NaT
dtype: datetime64[ns]

In [44]: pd.to_datetime(['2005/11/23', '2010.12.31'])
Out[44]: DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]',␣
↪freq=None)
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```
In [45]: pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)
Out[45]: DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]',␣
↪freq=None)

In [46]: pd.to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
Out[46]: DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]',␣
↪freq=None)
```

Warning: You see in the above example that `dayfirst` isnt strict, so if a date cant be parsed with the day being first it will be parsed as if `dayfirst` were False.

If you pass a single string to `to_datetime`, it returns a single `Timestamp`. `Timestamp` can also accept string input, but it doesnt accept string parsing options like `dayfirst` or `format`, so use `to_datetime` if these are required.

```
In [47]: pd.to_datetime('2010/11/12')
Out[47]: Timestamp('2010-11-12 00:00:00')

In [48]: pd.Timestamp('2010/11/12')
Out[48]: Timestamp('2010-11-12 00:00:00')
```

You can also use the `DatetimeIndex` constructor directly:

```
In [49]: pd.DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'])
Out[49]: DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], dtype=
↪'datetime64[ns]', freq=None)
```

The string infer can be passed in order to set the frequency of the index as the inferred frequency upon creation:

```
In [50]: pd.DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], freq='infer')
Out[50]: DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], dtype=
↪'datetime64[ns]', freq='2D')
```

Providing a format argument

In addition to the required datetime string, a `format` argument can be passed to ensure specific parsing. This could also potentially speed up the conversion considerably.

```
In [51]: pd.to_datetime('2010/11/12', format='%Y/%m/%d')
Out[51]: Timestamp('2010-11-12 00:00:00')

In [52]: pd.to_datetime('12-11-2010 00:00', format='%d-%m-%Y %H:%M')
Out[52]: Timestamp('2010-11-12 00:00:00')
```

For more information on the choices available when specifying the `format` option, see the Python datetime documentation.

Assembling datetime from multiple DataFrame columns

New in version 0.18.1.

You can also pass a `DataFrame` of integer or string columns to assemble into a `Series` of `Timestamps`.

```
In [53]: df = pd.DataFrame({'year': [2015, 2016],
   ....:                    'month': [2, 3],
   ....:                    'day': [4, 5],
   ....:                    'hour': [2, 3]})
   ....:

In [54]: pd.to_datetime(df)
Out[54]:
0   2015-02-04 02:00:00
1   2016-03-05 03:00:00
dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```
In [55]: pd.to_datetime(df[['year', 'month', 'day']])
Out[55]:
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

`pd.to_datetime` looks for standard designations of the datetime component in the column names, including:

- required: `year`, `month`, `day`
- optional: `hour`, `minute`, `second`, `millisecond`, `microsecond`, `nanosecond`

Invalid data

The default behavior, `errors='raise'`, is to raise when unparseable:

```
In [2]: pd.to_datetime(['2009/07/31', 'asd'], errors='raise')
ValueError: Unknown string format
```

Pass `errors='ignore'` to return the original input when unparseable:

```
In [56]: pd.to_datetime(['2009/07/31', 'asd'], errors='ignore')
Out[56]: Index(['2009/07/31', 'asd'], dtype='object')
```

Pass `errors='coerce'` to convert unparseable data to `NaT` (not a time):

```
In [57]: pd.to_datetime(['2009/07/31', 'asd'], errors='coerce')
Out[57]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

Epoch timestamps

pandas supports converting integer or float epoch times to `Timestamp` and `DatetimeIndex`. The default unit is nanoseconds, since that is how `Timestamp` objects are stored internally. However, epochs are often stored in another `unit` which can be specified. These are computed from the starting point specified by the `origin` parameter.

```
In [58]: pd.to_datetime([1349720105, 1349806505, 1349892905,
   ....:                 1349979305, 1350065705], unit='s')
   ....:
Out[58]:
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05',
               '2012-10-12 18:15:05'],
              dtype='datetime64[ns]', freq=None)

In [59]: pd.to_datetime([1349720105100, 1349720105200, 1349720105300,
   ....:                 1349720105400, 1349720105500], unit='ms')
   ....:
Out[59]:
DatetimeIndex(['2012-10-08 18:15:05.100000', '2012-10-08 18:15:05.200000',
               '2012-10-08 18:15:05.300000', '2012-10-08 18:15:05.400000',
               '2012-10-08 18:15:05.500000'],
              dtype='datetime64[ns]', freq=None)
```

Constructing a `Timestamp` or `DatetimeIndex` with an epoch timestamp with the `tz` argument specified will currently localize the epoch timestamps to UTC first then convert the result to the specified time zone. However, this behavior is *deprecated*, and if you have epochs in wall time in another timezone, it is recommended to read the epochs as timezone-naive timestamps and then localize to the appropriate timezone:

```
In [60]: pd.Timestamp(1262347200000000000).tz_localize('US/Pacific')
Out[60]: Timestamp('2010-01-01 12:00:00-0800', tz='US/Pacific')

In [61]: pd.DatetimeIndex([1262347200000000000]).tz_localize('US/Pacific')
Out[61]: DatetimeIndex(['2010-01-01 12:00:00-08:00'], dtype='datetime64[ns,␣
↪US/Pacific]', freq=None)
```

Note: Epoch times will be rounded to the nearest nanosecond.

Warning: Conversion of float epoch times can lead to inaccurate and unexpected results. Python floats have about 15 digits precision in decimal. Rounding during conversion from float to high precision `Timestamp` is unavoidable. The only way to achieve exact precision is to use a fixed-width types (e.g. an int64).

```
In [62]: pd.to_datetime([1490195805.433, 1490195805.433502912], unit='s')
Out[62]: DatetimeIndex(['2017-03-22 15:16:45.433000088', '2017-03-22␣
↪15:16:45.433502913'], dtype='datetime64[ns]', freq=None)

In [63]: pd.to_datetime(1490195805433502912, unit='ns')
```

```
Out[63]: Timestamp('2017-03-22 15:16:45.433502912')
```

See also:

Using the origin Parameter

From timestamps to epoch

To invert the operation from above, namely, to convert from a Timestamp to a unix epoch:

```
In [64]: stamps = pd.date_range('2012-10-08 18:15:05', periods=4, freq='D')

In [65]: stamps
Out[65]: 
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05'],
              dtype='datetime64[ns]', freq='D')
```

We subtract the epoch (midnight at January 1, 1970 UTC) and then floor divide by the unit (1 second).

```
In [66]: (stamps - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
Out[66]: Int64Index([1349720105, 1349806505, 1349892905, 1349979305], dtype='int64')
```

Using the origin Parameter

New in version 0.20.0.

Using the origin parameter, one can specify an alternative starting point for creation of a DatetimeIndex. For example, to use 1960-01-01 as the starting date:

```
In [67]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[67]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

The default is set at origin='unix', which defaults to 1970-01-01 00:00:00. Commonly called unix epoch or POSIX time.

```
In [68]: pd.to_datetime([1, 2, 3], unit='D')
Out[68]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

4.13.4 Generating ranges of timestamps

To generate an index with timestamps, you can use either the DatetimeIndex or Index constructor and pass in a list of datetime objects:

```
In [69]: dates = [datetime.datetime(2012, 5, 1),
   ....:          datetime.datetime(2012, 5, 2),
   ....:          datetime.datetime(2012, 5, 3)]
   ....: 

# Note the frequency information
```

(continues on next page)

(continued from previous page)

```
In [70]: index = pd.DatetimeIndex(dates)

In [71]: index
Out[71]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↪'datetime64[ns]', freq=None)

# Automatically converted to DatetimeIndex
In [72]: index = pd.Index(dates)

In [73]: index
Out[73]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↪'datetime64[ns]', freq=None)
```

In practice 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 `date_range()` and `bdate_range()` functions to create a `DatetimeIndex`. The default frequency for `date_range` is a **calendar day** while the default for `bdate_range` is a **business day**:

```
In [74]: start = datetime.datetime(2011, 1, 1)

In [75]: end = datetime.datetime(2012, 1, 1)

In [76]: index = pd.date_range(start, end)

In [77]: index
Out[77]:
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10',
               ...
               '2011-12-23', '2011-12-24', '2011-12-25', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30',
               '2011-12-31', '2012-01-01'],
              dtype='datetime64[ns]', length=366, freq='D')

In [78]: index = pd.bdate_range(start, end)

In [79]: index
Out[79]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14',
               ...
               '2011-12-19', '2011-12-20', '2011-12-21', '2011-12-22',
               '2011-12-23', '2011-12-26', '2011-12-27', '2011-12-28',
               '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', length=260, freq='B')
```

Convenience functions like `date_range` and `bdate_range` can utilize a variety of *frequency aliases*:

```
In [80]: pd.date_range(start, periods=1000, freq='M')
Out[80]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-30',
               '2011-05-31', '2011-06-30', '2011-07-31', '2011-08-31',
               '2011-09-30', '2011-10-31',
               ...
               '2093-07-31', '2093-08-31', '2093-09-30', '2093-10-31',
               '2093-11-30', '2093-12-31', '2094-01-31', '2094-02-28',
               '2094-03-31', '2094-04-30'],
              dtype='datetime64[ns]', length=1000, freq='M')

In [81]: pd.bdate_range(start, periods=250, freq='BQS')
Out[81]: 
DatetimeIndex(['2011-01-03', '2011-04-01', '2011-07-01', '2011-10-03',
               '2012-01-02', '2012-04-02', '2012-07-02', '2012-10-01',
               '2013-01-01', '2013-04-01',
               ...
               '2071-01-01', '2071-04-01', '2071-07-01', '2071-10-01',
               '2072-01-01', '2072-04-01', '2072-07-01', '2072-10-03',
               '2073-01-02', '2073-04-03'],
              dtype='datetime64[ns]', length=250, freq='BQS-JAN')
```

`date_range` and `bdate_range` make it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`. The start and end dates are strictly inclusive, so dates outside of those specified will not be generated:

```
In [82]: pd.date_range(start, end, freq='BM')
Out[82]: 
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [83]: pd.date_range(start, end, freq='W')
Out[83]: 
DatetimeIndex(['2011-01-02', '2011-01-09', '2011-01-16', '2011-01-23',
               '2011-01-30', '2011-02-06', '2011-02-13', '2011-02-20',
               '2011-02-27', '2011-03-06', '2011-03-13', '2011-03-20',
               '2011-03-27', '2011-04-03', '2011-04-10', '2011-04-17',
               '2011-04-24', '2011-05-01', '2011-05-08', '2011-05-15',
               '2011-05-22', '2011-05-29', '2011-06-05', '2011-06-12',
               '2011-06-19', '2011-06-26', '2011-07-03', '2011-07-10',
               '2011-07-17', '2011-07-24', '2011-07-31', '2011-08-07',
               '2011-08-14', '2011-08-21', '2011-08-28', '2011-09-04',
               '2011-09-11', '2011-09-18', '2011-09-25', '2011-10-02',
               '2011-10-09', '2011-10-16', '2011-10-23', '2011-10-30',
               '2011-11-06', '2011-11-13', '2011-11-20', '2011-11-27',
               '2011-12-04', '2011-12-11', '2011-12-18', '2011-12-25',
               '2012-01-01'],
              dtype='datetime64[ns]', freq='W-SUN')

In [84]: pd.bdate_range(end=end, periods=20)
Out[84]: 
DatetimeIndex(['2011-12-05', '2011-12-06', '2011-12-07', '2011-12-08',
               '2011-12-09', '2011-12-12', '2011-12-13', '2011-12-14',
               '2011-12-15', '2011-12-16', '2011-12-19', '2011-12-20',
               '2011-12-21', '2011-12-22', '2011-12-23', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', freq='B')

In [85]: pd.bdate_range(start=start, periods=20)
Out[85]: 
```

```
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-17', '2011-01-18',
               '2011-01-19', '2011-01-20', '2011-01-21', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28'],
              dtype='datetime64[ns]', freq='B')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced dates from `start` to `end` inclusively, with `periods` number of elements in the resulting `DatetimeIndex`:

```
In [86]: pd.date_range('2018-01-01', '2018-01-05', periods=5)
Out[86]:
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05'],
              dtype='datetime64[ns]', freq=None)

In [87]: pd.date_range('2018-01-01', '2018-01-05', periods=10)
Out[87]:
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 10:40:00',
               '2018-01-01 21:20:00', '2018-01-02 08:00:00',
               '2018-01-02 18:40:00', '2018-01-03 05:20:00',
               '2018-01-03 16:00:00', '2018-01-04 02:40:00',
               '2018-01-04 13:20:00', '2018-01-05 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Custom frequency ranges

`bdate_range` can also generate a range of custom frequency dates by using the `weekmask` and `holidays` parameters. These parameters will only be used if a custom frequency string is passed.

```
In [88]: weekmask = 'Mon Wed Fri'

In [89]: holidays = [datetime.datetime(2011, 1, 5), datetime.datetime(2011, 3,
↪ 14)]

In [90]: pd.bdate_range(start, end, freq='C', weekmask=weekmask,␣
↪holidays=holidays)
Out[90]:
DatetimeIndex(['2011-01-03', '2011-01-07', '2011-01-10', '2011-01-12',
               '2011-01-14', '2011-01-17', '2011-01-19', '2011-01-21',
               '2011-01-24', '2011-01-26',
               ...
               '2011-12-09', '2011-12-12', '2011-12-14', '2011-12-16',
               '2011-12-19', '2011-12-21', '2011-12-23', '2011-12-26',
               '2011-12-28', '2011-12-30'],
              dtype='datetime64[ns]', length=154, freq='C')

In [91]: pd.bdate_range(start, end, freq='CBMS', weekmask=weekmask)
Out[91]:
DatetimeIndex(['2011-01-03', '2011-02-02', '2011-03-02', '2011-04-01',
               '2011-05-02', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-02', '2011-10-03', '2011-11-02', '2011-12-02'],
              dtype='datetime64[ns]', freq='CBMS')
```

See also:

Custom business days

4.13.5 Timestamp limitations

Since pandas represents timestamps in nanosecond resolution, the time span that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [92]: pd.Timestamp.min
Out[92]: Timestamp('1677-09-21 00:12:43.145225')

In [93]: pd.Timestamp.max
Out[93]: Timestamp('2262-04-11 23:47:16.854775807')
```

See also:

Representing out-of-bounds spans

4.13.6 Indexing

One of the main uses for `DatetimeIndex` is as an index for pandas objects. The `DatetimeIndex` class contains many time series 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 have all the basic functionality of regular `Index` objects, and a smorgasbord of advanced time series 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.

`DatetimeIndex` can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```
In [94]: rng = pd.date_range(start, end, freq='BM')

In [95]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [96]: ts.index
Out[96]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [97]: ts[:5].index
Out[97]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31'],
              dtype='datetime64[ns]', freq='BM')

In [98]: ts[::2].index
Out[98]:
DatetimeIndex(['2011-01-31', '2011-03-31', '2011-05-31', '2011-07-29',
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```

Partial string indexing

Dates and strings that parse to timestamps can be passed as indexing parameters:

```
In [99]: ts['1/31/2011']
Out[99]: 0.3135034497740378

In [100]: ts[datetime.datetime(2011, 12, 25):]
Out[100]:
2011-12-30   -0.62564
Freq: BM, dtype: float64

In [101]: ts['10/31/2011':'12/31/2011']
Out[101]:
2011-10-31   -0.042894
2011-11-30    1.321441
2011-12-30   -0.625640
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 [102]: ts['2011']
Out[102]:
2011-01-31    0.313503
2011-02-28    1.055889
2011-03-31    0.714651
2011-04-29    0.612442
2011-05-31   -0.170038
2011-06-30   -0.828809
2011-07-29    0.458806
2011-08-31    0.764740
2011-09-30    1.917429
2011-10-31   -0.042894
2011-11-30    1.321441
2011-12-30   -0.625640
Freq: BM, dtype: float64

In [103]: ts['2011-6']
Out[103]:
2011-06-30   -0.828809
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:

```
In [104]: dft = pd.DataFrame(np.random.randn(100000, 1), columns=['A'],
   .....:                    index=pd.date_range('20130101', periods=100000,␣
↪freq='T'))
   .....:

In [105]: dft
Out[105]:
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-03-11 10:35:00  0.486569
2013-03-11 10:36:00 -0.586561
2013-03-11 10:37:00 -0.403601
2013-03-11 10:38:00 -0.228277
2013-03-11 10:39:00 -0.267201

[100000 rows x 1 columns]

In [106]: dft['2013']
Out[106]:
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-03-11 10:35:00  0.486569
2013-03-11 10:36:00 -0.586561
2013-03-11 10:37:00 -0.403601
2013-03-11 10:38:00 -0.228277
2013-03-11 10:39:00 -0.267201

[100000 rows x 1 columns]
```

This starts on the very first time in the month, and includes the last date and time for the month:

```
In [107]: dft['2013-1':'2013-2']
Out[107]:
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-02-28 23:55:00  1.819527
2013-02-28 23:56:00  0.891281
```

(continues on next page)

(continued from previous page)

```
2013-02-28 23:57:00 -0.516058
2013-02-28 23:58:00 -1.350302
2013-02-28 23:59:00  1.475049

[84960 rows x 1 columns]
```

This specifies a stop time **that includes all of the times on the last day**:

```
In [108]: dft['2013-1':'2013-2-28']
Out[108]:
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-02-28 23:55:00  1.819527
2013-02-28 23:56:00  0.891281
2013-02-28 23:57:00 -0.516058
2013-02-28 23:58:00 -1.350302
2013-02-28 23:59:00  1.475049

[84960 rows x 1 columns]
```

This specifies an **exact** stop time (and is not the same as the above):

```
In [109]: dft['2013-1':'2013-2-28 00:00:00']
Out[109]:
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-02-27 23:56:00  0.032319
2013-02-27 23:57:00  0.773067
2013-02-27 23:58:00 -1.433695
2013-02-27 23:59:00 -0.759026
2013-02-28 00:00:00  0.659128

[83521 rows x 1 columns]
```

We are stopping on the included end-point as it is part of the index:

```
In [110]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[110]:
                            A
2013-01-15 00:00:00 -2.553103
2013-01-15 00:01:00  1.310783
2013-01-15 00:02:00  0.851770
2013-01-15 00:03:00 -0.534226
2013-01-15 00:04:00 -1.030374
...                       ...
2013-01-15 12:26:00  1.679647
2013-01-15 12:27:00  0.049848
```

(continues on next page)

(continued from previous page)

```
2013-01-15 12:28:00 -0.722124
2013-01-15 12:29:00  0.870311
2013-01-15 12:30:00  0.360159

[751 rows x 1 columns]
```

New in version 0.18.0.

`DatetimeIndex` partial string indexing also works on a `DataFrame` with a `MultiIndex`:

```
In [111]: dft2 = pd.DataFrame(np.random.randn(20, 1),
   .....:                     columns=['A'],
   .....:                     index=pd.MultiIndex.from_product(
   .....:                         [pd.date_range('20130101', periods=10,␣
↪freq='12H'),
   .....:                          ['a', 'b']]))
   .....:

In [112]: dft2
Out[112]:
                              A
2013-01-01 00:00:00 a  0.030508
                    b  0.201088
2013-01-01 12:00:00 a -0.822650
                    b -0.159673
2013-01-02 00:00:00 a  0.715939
                    b  0.635435
2013-01-02 12:00:00 a  0.071542
                    b  0.539646
2013-01-03 00:00:00 a -0.743837
                    b  1.319587
2013-01-03 12:00:00 a -0.501123
                    b -0.492347
2013-01-04 00:00:00 a -0.357006
                    b -0.252463
2013-01-04 12:00:00 a -1.140700
                    b -1.367172
2013-01-05 00:00:00 a  0.048871
                    b -0.400048
2013-01-05 12:00:00 a  1.325801
                    b  0.651751

In [113]: dft2.loc['2013-01-05']
Out[113]:
                              A
2013-01-05 00:00:00 a  0.048871
                    b -0.400048
2013-01-05 12:00:00 a  1.325801
                    b  0.651751

In [114]: idx = pd.IndexSlice

In [115]: dft2 = dft2.swaplevel(0, 1).sort_index()
In [116]: dft2.loc[idx[:, '2013-01-05'], :]
Out[116]:
                              A
a 2013-01-05 00:00:00  0.048871
  2013-01-05 12:00:00  1.325801
b 2013-01-05 00:00:00 -0.400048
  2013-01-05 12:00:00  0.651751
```

New in version 0.25.0.

Slicing with string indexing also honors UTC offset.

```
In [117]: df = pd.DataFrame([0], index=pd.DatetimeIndex(['2019-01-01'],␣
↪tz='US/Pacific'))

In [118]: df
Out[118]:
                           0
2019-01-01 00:00:00-08:00  0

In [119]: df['2019-01-01 12:00:00+04:00':'2019-01-01 13:00:00+04:00']
Out[119]:
                           0
2019-01-01 00:00:00-08:00  0
```

Slice vs. exact match

Changed in version 0.20.0.

The same string used as an indexing parameter can be treated either as a slice or as an exact match depending on the resolution of the index. If the string is less accurate than the index, it will be treated as a slice, otherwise as an exact match.

Consider a `Series` object with a minute resolution index:

```
In [120]: series_minute = pd.Series([1, 2, 3],
   .....:                           pd.DatetimeIndex(['2011-12-31 23:59:00',
   .....:                                             '2012-01-01 00:00:00',
   .....:                                             '2012-01-01 00:02:00']))
   .....:

In [121]: series_minute.index.resolution
Out[121]: 'minute'
```

A timestamp string less accurate than a minute gives a `Series` object.

```
In [122]: series_minute['2011-12-31 23']
Out[122]:
2011-12-31 23:59:00    1
dtype: int64
```

A timestamp string with minute resolution (or more accurate), gives a scalar instead, i.e. it is not casted to a slice.

```
In [123]: series_minute['2011-12-31 23:59']
Out[123]: 1

In [124]: series_minute['2011-12-31 23:59:00']
Out[124]: 1
```

If index resolution is second, then the minute-accurate timestamp gives a `Series`.

```
In [125]: series_second = pd.Series([1, 2, 3],
   .....:                           pd.DatetimeIndex(['2011-12-31 23:59:59',
   .....:                                             '2012-01-01 00:00:00',
   .....:                                             '2012-01-01 00:00:01']))
   .....:

In [126]: series_second.index.resolution
Out[126]: 'second'

In [127]: series_second['2011-12-31 23:59']
Out[127]:
2011-12-31 23:59:59    1
dtype: int64
```

If the timestamp string is treated as a slice, it can be used to index `DataFrame` with `[]` as well.

```
In [128]: dft_minute = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]},
   .....:                           index=series_minute.index)
   .....:

In [129]: dft_minute['2011-12-31 23']
Out[129]:
                     a  b
2011-12-31 23:59:00  1  4
```

Warning: However, if the string is treated as an exact match, the selection in `DataFrames` `[]` will be column-wise and not row-wise, see *Indexing Basics*. For example `dft_minute['2011-12-31 23:59']` will raise `KeyError` as `'2012-12-31 23:59'` has the same resolution as the index and there is no column with such name:

To *always* have unambiguous selection, whether the row is treated as a slice or a single selection, use `.loc`.

```
In [130]: dft_minute.loc['2011-12-31 23:59']
Out[130]:
a    1
b    4
Name: 2011-12-31 23:59:00, dtype: int64
```

Note also that `DatetimeIndex` resolution cannot be less precise than day.

```
In [131]: series_monthly = pd.Series([1, 2, 3],
   .....:                            pd.DatetimeIndex(['2011-12', '2012-01',␣
↪'2012-02']))
   .....:

In [132]: series_monthly.index.resolution
Out[132]: 'day'

In [133]: series_monthly['2011-12']  # returns Series
Out[133]:
2011-12-01    1
dtype: int64
```

Exact indexing

As discussed in previous section, 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 resolution of the index. In contrast, indexing with `Timestamp` or `datetime` objects is exact, because the objects have exact meaning. These also follow the semantics of *including both endpoints*.

These `Timestamp` and `datetime` objects have exact `hours, minutes,` and `seconds`, even though they were not explicitly specified (they are `0`).

```
In [134]: dft[datetime.datetime(2013, 1, 1):datetime.datetime(2013, 2, 28)]
Out[134]: 
                            A
2013-01-01 00:00:00  0.480217
2013-01-01 00:01:00  0.791875
2013-01-01 00:02:00  1.414514
2013-01-01 00:03:00 -0.801147
2013-01-01 00:04:00  0.462656
...                       ...
2013-02-27 23:56:00  0.032319
2013-02-27 23:57:00  0.773067
2013-02-27 23:58:00 -1.433695
2013-02-27 23:59:00 -0.759026
2013-02-28 00:00:00  0.659128

[83521 rows x 1 columns]
```

With no defaults.

```
In [135]: dft[datetime.datetime(2013, 1, 1, 10, 12, 0):
   .....:     datetime.datetime(2013, 2, 28, 10, 12, 0)]
   .....: 
Out[135]: 
                            A
2013-01-01 10:12:00  0.827464
2013-01-01 10:13:00 -2.923827
2013-01-01 10:14:00 -2.109236
2013-01-01 10:15:00 -1.088176
2013-01-01 10:16:00  0.788969
...                       ...
2013-02-28 10:08:00 -1.473233
2013-02-28 10:09:00  0.090886
2013-02-28 10:10:00  0.211966
2013-02-28 10:11:00 -1.521579
2013-02-28 10:12:00 -0.318755

[83521 rows x 1 columns]
```

Truncating & fancy indexing