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 you're 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.

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).

Development team

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

Institutional partners

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

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:

* 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
```

(continues on next page)

(continued from previous page)


```
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 }}

1.4.3 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
```

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  1.897501 -0.824858  0.261392 -0.293845
```

(continues on next page)

(continued from previous page)

```
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-05  1.175223  0.520027  1.259335  1.039387
2013-01-06  1.224282  1.432164  0.894608 -1.063357
```

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

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

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  1.897501 -0.824858  0.261392 -0.293845
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-05  1.175223  0.520027  1.259335  1.039387

In [14]: df.tail(3)
Out[14]:
                   A         B         C         D
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-05  1.175223  0.520027  1.259335  1.039387
2013-01-06  1.224282  1.432164  0.894608 -1.063357
```

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 doesn't require copying data.

```
In [17]: df.to_numpy()
Out[17]:
array([[ 1.89750056, -0.8248584 ,  0.26139167, -0.29384485],
       [ 0.47106432,  0.74834695, -0.26601476, -0.13547679],
       [ 0.75065515, -0.53013621, -1.96416288,  0.42594698],
       [ 1.65749758,  0.98016498,  1.02182883,  0.96073459],
       [ 1.17522291,  0.52002729,  1.25933545,  1.03938712],
       [ 1.22428156,  1.43216382,  0.89460796, -1.06335746]])
```

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'],
```

(continues on next page)

(continued from previous page)

```
       [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   1.196037  0.387618  0.201164  0.155565
std    0.534823  0.883427  1.198716  0.809320
min    0.471064 -0.824858 -1.964163 -1.063357
25%    0.856797 -0.267595 -0.134163 -0.254253
50%    1.199752  0.634187  0.578000  0.145235
75%    1.549194  0.922210  0.990024  0.827038
max    1.897501  1.432164  1.259335  1.039387
```

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    1.897501    0.471064    0.750655    1.657498    1.175223    1.224282
B   -0.824858    0.748347   -0.530136    0.980165    0.520027    1.432164
C    0.261392   -0.266015   -1.964163    1.021829    1.259335    0.894608
D   -0.293845   -0.135477    0.425947    0.960735    1.039387   -1.063357
```

Sorting by an axis:

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]: 
                   D         C         B         A
2013-01-01 -0.293845  0.261392 -0.824858  1.897501
2013-01-02 -0.135477 -0.266015  0.748347  0.471064
2013-01-03  0.425947 -1.964163 -0.530136  0.750655
2013-01-04  0.960735  1.021829  0.980165  1.657498
2013-01-05  1.039387  1.259335  0.520027  1.175223
2013-01-06 -1.063357  0.894608  1.432164  1.224282
```

Sorting by values:

```
In [22]: df.sort_values(by='B')
Out[22]: 
                   A         B         C         D
2013-01-01  1.897501 -0.824858  0.261392 -0.293845
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
2013-01-05  1.175223  0.520027  1.259335  1.039387
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-06  1.224282  1.432164  0.894608 -1.063357
```

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    1.897501
2013-01-02    0.471064
2013-01-03    0.750655
2013-01-04    1.657498
2013-01-05    1.175223
2013-01-06    1.224282
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  1.897501 -0.824858  0.261392 -0.293845
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947

In [25]: df['20130102':'20130104']
Out[25]:
                   A         B         C         D
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
2013-01-04  1.657498  0.980165  1.021829  0.960735
```

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    1.897501
B   -0.824858
C    0.261392
D   -0.293845
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  1.897501 -0.824858
2013-01-02  0.471064  0.748347
2013-01-03  0.750655 -0.530136
2013-01-04  1.657498  0.980165
2013-01-05  1.175223  0.520027
2013-01-06  1.224282  1.432164
```

Showing label slicing, both endpoints are *included*:

```
In [28]: df.loc['20130102':'20130104', ['A', 'B']]
Out[28]:
                   A         B
2013-01-02  0.471064  0.748347
2013-01-03  0.750655 -0.530136
2013-01-04  1.657498  0.980165
```

Reduction in the dimensions of the returned object:

```
In [29]: df.loc['20130102', ['A', 'B']]
Out[29]:
A    0.471064
B    0.748347
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], 'A']
Out[30]: 1.8975005556825981
```

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

```
In [31]: df.at[dates[0], 'A']
Out[31]: 1.8975005556825981
```

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    1.657498
B    0.980165
C    1.021829
D    0.960735
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
```

(continues on next page)

(continued from previous page)

```
2013-01-04  1.657498  0.980165
2013-01-05  1.175223  0.520027
```

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  0.471064 -0.266015
2013-01-03  0.750655 -1.964163
2013-01-05  1.175223  1.259335
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
                   A         B         C         D
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
```

For slicing columns explicitly:

```
In [36]: df.iloc[:, 1:3]
Out[36]:
                   B         C
2013-01-01 -0.824858  0.261392
2013-01-02  0.748347 -0.266015
2013-01-03 -0.530136 -1.964163
2013-01-04  0.980165  1.021829
2013-01-05  0.520027  1.259335
2013-01-06  1.432164  0.894608
```

For getting a value explicitly:

```
In [37]: df.iloc[1, 1]
Out[37]: 0.7483469482146548
```

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

```
In [38]: df.iat[1, 1]
Out[38]: 0.7483469482146548
```

Boolean indexing

Using a single column's values to select data.

```
In [39]: df[df['A'] > 0]
Out[39]:
                   A         B         C         D
2013-01-01  1.897501 -0.824858  0.261392 -0.293845
2013-01-02  0.471064  0.748347 -0.266015 -0.135477
2013-01-03  0.750655 -0.530136 -1.964163  0.425947
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-05  1.175223  0.520027  1.259335  1.039387
2013-01-06  1.224282  1.432164  0.894608 -1.063357
```

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  1.897501       NaN  0.261392       NaN
2013-01-02  0.471064  0.748347       NaN       NaN
2013-01-03  0.750655       NaN       NaN  0.425947
2013-01-04  1.657498  0.980165  1.021829  0.960735
2013-01-05  1.175223  0.520027  1.259335  1.039387
2013-01-06  1.224282  1.432164  0.894608       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  1.897501 -0.824858  0.261392 -0.293845    one
2013-01-02  0.471064  0.748347 -0.266015 -0.135477    one
2013-01-03  0.750655 -0.530136 -1.964163  0.425947    two
2013-01-04  1.657498  0.980165  1.021829  0.960735  three
2013-01-05  1.175223  0.520027  1.259335  1.039387   four
2013-01-06  1.224282  1.432164  0.894608 -1.063357  three

In [44]: df2[df2['E'].isin(['two', 'four'])]
Out[44]:
                   A         B         C         D     E
2013-01-03  0.750655 -0.530136 -1.964163  0.425947   two
2013-01-05  1.175223  0.520027  1.259335  1.039387  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  0.261392  5  NaN
2013-01-02  0.471064  0.748347 -0.266015  5  1.0
2013-01-03  0.750655 -0.530136 -1.964163  5  2.0
2013-01-04  1.657498  0.980165  1.021829  5  3.0
2013-01-05  1.175223  0.520027  1.259335  5  4.0
2013-01-06  1.224282  1.432164  0.894608  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 -0.261392 -5  NaN
2013-01-02 -0.471064 -0.748347 -0.266015 -5 -1.0
2013-01-03 -0.750655 -0.530136 -1.964163 -5 -2.0
2013-01-04 -1.657498 -0.980165 -1.021829 -5 -3.0
2013-01-05 -1.175223 -0.520027 -1.259335 -5 -4.0
2013-01-06 -1.224282 -1.432164 -0.894608 -5 -5.0
```

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  0.261392  5  NaN  1.0
2013-01-02  0.471064  0.748347 -0.266015  5  1.0  1.0
2013-01-03  0.750655 -0.530136 -1.964163  5  2.0  NaN
2013-01-04  1.657498  0.980165  1.021829  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  0.471064  0.748347 -0.266015  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  0.261392  5  5.0  1.0
2013-01-02  0.471064  0.748347 -0.266015  5  1.0  1.0
2013-01-03  0.750655 -0.530136 -1.964163  5  2.0  5.0
2013-01-04  1.657498  0.980165  1.021829  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
```

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    0.879787
B    0.525094
C    0.201164
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    1.315348
2013-01-02    1.390679
2013-01-03    1.051271
2013-01-04    2.331898
2013-01-05    2.390917
2013-01-06    2.710211
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 -0.249345 -1.530136 -2.964163  4.0  1.0
2013-01-04 -1.342502 -2.019835 -1.978171  2.0  0.0
2013-01-05 -3.824777 -4.479973 -3.740665  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  0.261392   5   NaN
2013-01-02  0.471064  0.748347 -0.004623  10   1.0
2013-01-03  1.221719  0.218211 -1.968786  15   3.0
2013-01-04  2.879217  1.198376 -0.946957  20   6.0
2013-01-05  4.054440  1.718403  0.312378  25  10.0
2013-01-06  5.278722  3.150567  1.206986  30  15.0

In [67]: df.apply(lambda x: x.max() - x.min())
Out[67]:
A    1.657498
B    1.962300
C    3.223498
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    0
1    0
2    0
3    6
4    2
5    2
6    4
7    1
8    5
9    0
dtype: int64

In [70]: s.value_counts()
Out[70]:
0    4
2    2
6    1
5    1
4    1
1    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
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

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  0.109215  0.889572 -0.632724 -0.033554
1 -0.598168  0.036744 -1.267156  1.468185
2 -0.977383 -0.809893 -0.374513  1.025521
3 -0.332141  2.032386  0.198440 -0.300240
4 -0.275232 -0.708336 -0.757731  1.290129
5 -1.953918 -0.765158  0.950298  0.125226
6 -0.750831 -0.331657  1.535099  2.471824
7  0.133147  0.305182 -0.645845  0.407080
8 -0.975879 -0.061448 -1.340803 -0.614017
9 -1.363160  1.161824 -0.099942 -0.527464

# 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  0.109215  0.889572 -0.632724 -0.033554
1 -0.598168  0.036744 -1.267156  1.468185
2 -0.977383 -0.809893 -0.374513  1.025521
3 -0.332141  2.032386  0.198440 -0.300240
4 -0.275232 -0.708336 -0.757731  1.290129
5 -1.953918 -0.765158  0.950298  0.125226
6 -0.750831 -0.331657  1.535099  2.471824
7  0.133147  0.305182 -0.645845  0.407080
8 -0.975879 -0.061448 -1.340803 -0.614017
9 -1.363160  1.161824 -0.099942 -0.527464
```

Note: Adding a column to a `DataFrame` is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the `DataFrame` constructor instead of building a `DataFrame` by iteratively appending records to it. See *Appending to dataframe* for more.

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
```

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 [87]: 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 [88]: df
Out[88]:
     A      B         C         D
0  foo    one  2.123933  1.809498
1  bar    one -0.512351  1.380992
2  foo    two  1.181485 -0.247795
3  bar  three  1.223647 -1.615201
4  foo    two  1.626605  0.246011
5  bar    two  1.083931  0.650619
6  foo    one -0.582405 -0.452213
7  foo  three  0.878575  1.566884
```

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

```
In [89]: df.groupby('A').sum()
Out[89]:
            C         D
A
bar  1.795227  0.416410
foo  5.228193  2.922384
```

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

```
In [90]: df.groupby(['A', 'B']).sum()
Out[90]:
                  C         D
A   B
bar one   -0.512351  1.380992
    three  1.223647 -1.615201
    two    1.083931  0.650619
foo one    1.541528  1.357285
    three  0.878575  1.566884
    two    2.808090 -0.001784
```

Reshaping

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

Stack

```
In [91]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
   ....:                      'foo', 'foo', 'qux', 'qux'],
   ....:                     ['one', 'two', 'one', 'two',
   ....:                      'one', 'two', 'one', 'two']]))
   ....: 

In [92]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [93]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [94]: df2 = df[:4]

In [95]: df2
Out[95]: 
                     A         B
first second                    
bar   one    -0.986965 -0.185302
      two    -0.354776 -0.392071
baz   one     0.758271  1.072199
      two    -1.208995  0.414066
```

The `stack()` method "compresses" a level in the DataFrame's columns.

```
In [96]: stacked = df2.stack()

In [97]: stacked
Out[97]: 
first  second   
bar    one     A   -0.986965
               B   -0.185302
       two     A   -0.354776
               B   -0.392071
baz    one     A    0.758271
               B    1.072199
       two     A   -1.208995
               B    0.414066
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 [98]: stacked.unstack()
Out[98]: 
                     A         B
first second                    
bar   one    -0.986965 -0.185302
      two    -0.354776 -0.392071
baz   one     0.758271  1.072199
      two    -1.208995  0.414066
```

(continues on next page)

(continued from previous page)

```
In [99]: stacked.unstack(1)
Out[99]:
second        one       two
first
bar    A -0.986965 -0.354776
       B -0.185302 -0.392071
baz    A  0.758271 -1.208995
       B  1.072199  0.414066

In [100]: stacked.unstack(0)
Out[100]:
first          bar       baz
second
one     A -0.986965  0.758271
        B -0.185302  1.072199
two     A -0.354776 -1.208995
        B -0.392071  0.414066
```

Pivot tables

See the section on *Pivot Tables*.

```
In [101]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
   .....:                    'B': ['A', 'B', 'C'] * 4,
   .....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
   .....:                    'D': np.random.randn(12),
   .....:                    'E': np.random.randn(12)})
   .....:

In [102]: df
Out[102]:
        A  B    C         D         E
0     one  A  foo -2.118899 -0.504792
1     one  B  foo -0.131137  0.135539
2     two  C  foo -1.915548  2.348321
3   three  A  bar -0.394289  2.068605
4     one  B  bar -0.535694  1.069768
5     one  C  bar -1.157419  2.057720
6     two  A  foo -1.145990 -1.737902
7   three  B  foo -0.830670 -0.627602
8     one  C  foo -0.946261  1.220538
9     one  A  bar  0.893534  1.283678
10    two  B  bar  2.321785  2.182541
11  three  C  bar  0.220623 -0.549491
```

We can produce pivot tables from this data very easily:

```
In [103]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[103]:
C             bar       foo
A     B
one   A  0.893534 -2.118899
      B -0.535694 -0.131137
      C -1.157419 -0.946261
three A -0.394289       NaN
```

(continues on next page)

(continued from previous page)

```
      B       NaN -0.830670
      C  0.220623       NaN
two   A       NaN -1.145990
      B  2.321785       NaN
      C       NaN -1.915548
```

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

In [105]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [106]: ts.resample('5Min').sum()
Out[106]:
2012-01-01    23108
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [107]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [108]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [109]: ts
Out[109]:
2012-03-06   -0.309668
2012-03-07   -0.414286
2012-03-08    1.271311
2012-03-09    1.015239
2012-03-10   -0.221046
Freq: D, dtype: float64

In [110]: ts_utc = ts.tz_localize('UTC')

In [111]: ts_utc
Out[111]:
2012-03-06 00:00:00+00:00   -0.309668
2012-03-07 00:00:00+00:00   -0.414286
2012-03-08 00:00:00+00:00    1.271311
2012-03-09 00:00:00+00:00    1.015239
2012-03-10 00:00:00+00:00   -0.221046
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [112]: ts_utc.tz_convert('US/Eastern')
Out[112]:
2012-03-05 19:00:00-05:00   -0.309668
2012-03-06 19:00:00-05:00   -0.414286
2012-03-07 19:00:00-05:00    1.271311
2012-03-08 19:00:00-05:00    1.015239
```

(continues on next page)

(continued from previous page)

```
2012-03-09 19:00:00-05:00   -0.221046
Freq: D, dtype: float64
```

Converting between time span representations:

```
In [113]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [114]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [115]: ts
Out[115]: 
2012-01-31    1.317178
2012-02-29   -0.900960
2012-03-31   -0.635198
2012-04-30   -0.882834
2012-05-31    0.346617
Freq: M, dtype: float64

In [116]: ps = ts.to_period()

In [117]: ps
Out[117]: 
2012-01    1.317178
2012-02   -0.900960
2012-03   -0.635198
2012-04   -0.882834
2012-05    0.346617
Freq: M, dtype: float64

In [118]: ps.to_timestamp()
Out[118]: 
2012-01-01    1.317178
2012-02-01   -0.900960
2012-03-01   -0.635198
2012-04-01   -0.882834
2012-05-01    0.346617
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 [119]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [120]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [121]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [122]: ts.head()
Out[122]: 
1990-03-01 09:00    0.514201
1990-06-01 09:00   -0.583787
1990-09-01 09:00    1.106193
1990-12-01 09:00   -0.293941
1991-03-01 09:00   -0.069979
Freq: H, dtype: float64
```

Categoricals

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

```
In [123]: 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 [124]: df["grade"] = df["raw_grade"].astype("category")

In [125]: df["grade"]
Out[125]:
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 [126]: 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 [127]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                                "good", "very good"])
   .....:

In [128]: df["grade"]
Out[128]:
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 [129]: df.sort_values(by="grade")
Out[129]:
   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 [130]: df.groupby("grade").size()
Out[130]:
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

Plotting

See the *Plotting* docs.

We use the standard convention for referencing the matplotlib API:

```
In [131]: import matplotlib.pyplot as plt

In [132]: plt.close('all')
```

```
In [133]: ts = pd.Series(np.random.randn(1000),
   .....:                index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [134]: ts = ts.cumsum()

In [135]: ts.plot()
Out[135]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5bf513e690>
```

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

```
In [136]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
   .....:                   columns=['A', 'B', 'C', 'D'])
   .....:

In [137]: df = df.cumsum()

In [138]: plt.figure()
Out[138]: <Figure size 640x480 with 0 Axes>

In [139]: df.plot()
Out[139]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5bf4e0db50>

In [140]: plt.legend(loc='best')
Out[140]: <matplotlib.legend.Legend at 0x7f5bf4df6e50>
```

Getting data in/out

CSV

Writing to a csv file.

```
In [141]: df.to_csv('foo.csv')
```

Reading from a csv file.

```
In [142]: pd.read_csv('foo.csv')
Out[142]:
     Unnamed: 0          A          B          C          D
0    2000-01-01   0.728743   1.655598   1.671244   1.087989
1    2000-01-02  -0.003647   1.610951   1.840507   0.607367
2    2000-01-03  -0.311393   2.139550   2.883461   0.518210
3    2000-01-04   0.631263   3.179374   2.606323   1.320533
4    2000-01-05  -0.270184   3.103292   3.451395   1.256303
..          ...        ...        ...        ...        ...
995  2002-09-22   9.300506 -52.115524  20.594501   9.509217
996  2002-09-23  10.388986 -51.112260  20.300889   8.093781
997  2002-09-24  11.442359 -50.309056  21.913125   7.942181
998  2002-09-25  11.113233 -50.307356  22.129366   7.970033
```

(continues on next page)

(continued from previous page)

```
999  2002-09-26  11.440539 -50.553522  21.762509  7.829262

[1000 rows x 5 columns]
```

HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store.

```
In [143]: df.to_hdf('foo.h5', 'df')
```

Reading from a HDF5 Store.

```
In [144]: pd.read_hdf('foo.h5', 'df')
Out[144]:
                    A          B          C         D
2000-01-01   0.728743   1.655598   1.671244  1.087989
2000-01-02  -0.003647   1.610951   1.840507  0.607367
2000-01-03  -0.311393   2.139550   2.883461  0.518210
2000-01-04   0.631263   3.179374   2.606323  1.320533
2000-01-05  -0.270184   3.103292   3.451395  1.256303
...               ...        ...        ...       ...
2002-09-22   9.300506 -52.115524  20.594501  9.509217
2002-09-23  10.388986 -51.112260  20.300889  8.093781
2002-09-24  11.442359 -50.309056  21.913125  7.942181
2002-09-25  11.113233 -50.307356  22.129366  7.970033
2002-09-26  11.440539 -50.553522  21.762509  7.829262

[1000 rows x 4 columns]
```

Excel

Reading and writing to *MS Excel*.

Writing to an excel file.

```
In [145]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

Reading from an excel file.

```
In [146]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
Out[146]:
    Unnamed: 0          A          B          C         D
0   2000-01-01   0.728743   1.655598   1.671244  1.087989
1   2000-01-02  -0.003647   1.610951   1.840507  0.607367
2   2000-01-03  -0.311393   2.139550   2.883461  0.518210
3   2000-01-04   0.631263   3.179374   2.606323  1.320533
4   2000-01-05  -0.270184   3.103292   3.451395  1.256303
..         ...        ...        ...        ...       ...
995 2002-09-22   9.300506 -52.115524  20.594501  9.509217
996 2002-09-23  10.388986 -51.112260  20.300889  8.093781
997 2002-09-24  11.442359 -50.309056  21.913125  7.942181
998 2002-09-25  11.113233 -50.307356  22.129366  7.970033
```

(continues on next page)