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 array's `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.

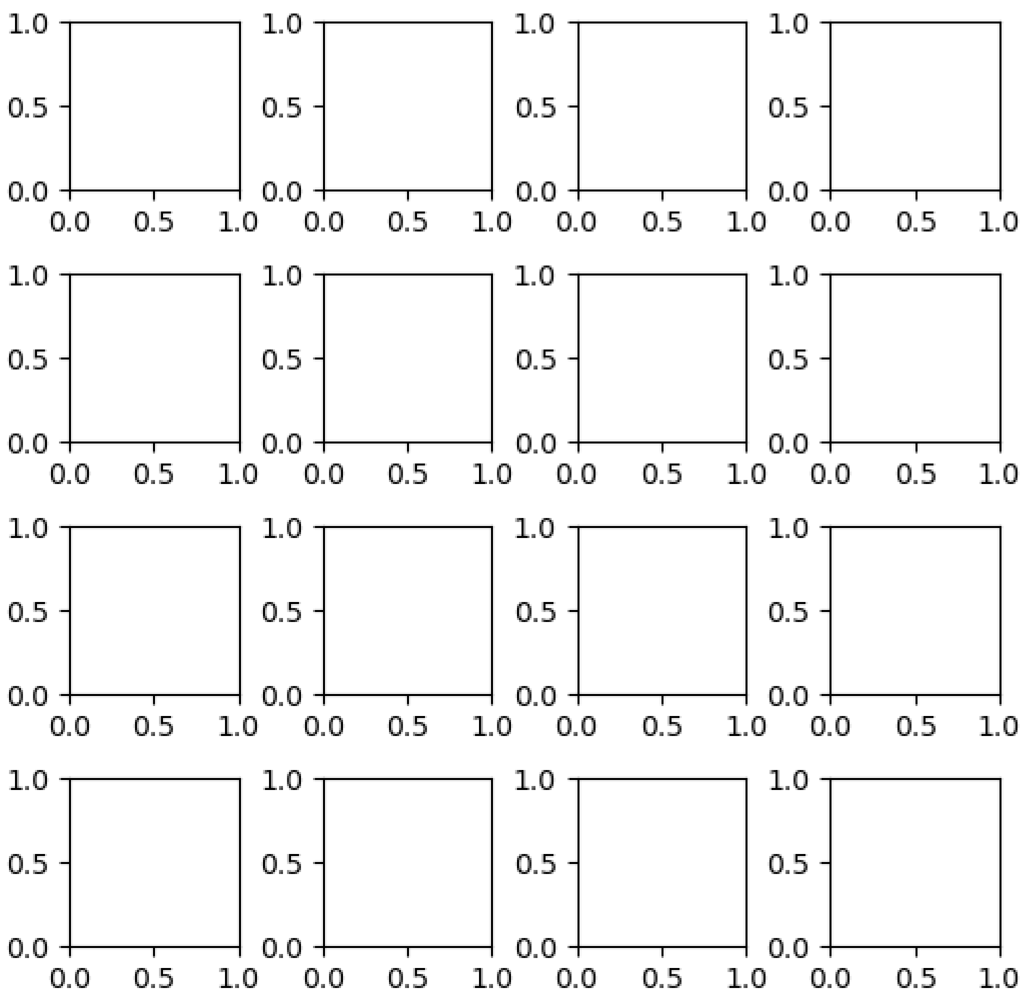
When multiple axes are passed via the `ax` keyword, `layout`, `sharex` and `sharey` keywords don't affect to the output. You should explicitly pass `sharex=False` and `sharey=False`, otherwise you will see a warning.

```
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]]

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);
```

(continues on next page)

(continued from previous page)

```
In [143]: (-df).plot(subplots=True, ax=target2, legend=False,
   .....:             sharex=False, sharey=False);
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-143-a9df76fad608> in <module>
----> 1 (-df).plot(subplots=True, ax=target2, legend=False,
      2             sharex=False, sharey=False);

NameError: name 'df' is not defined
```



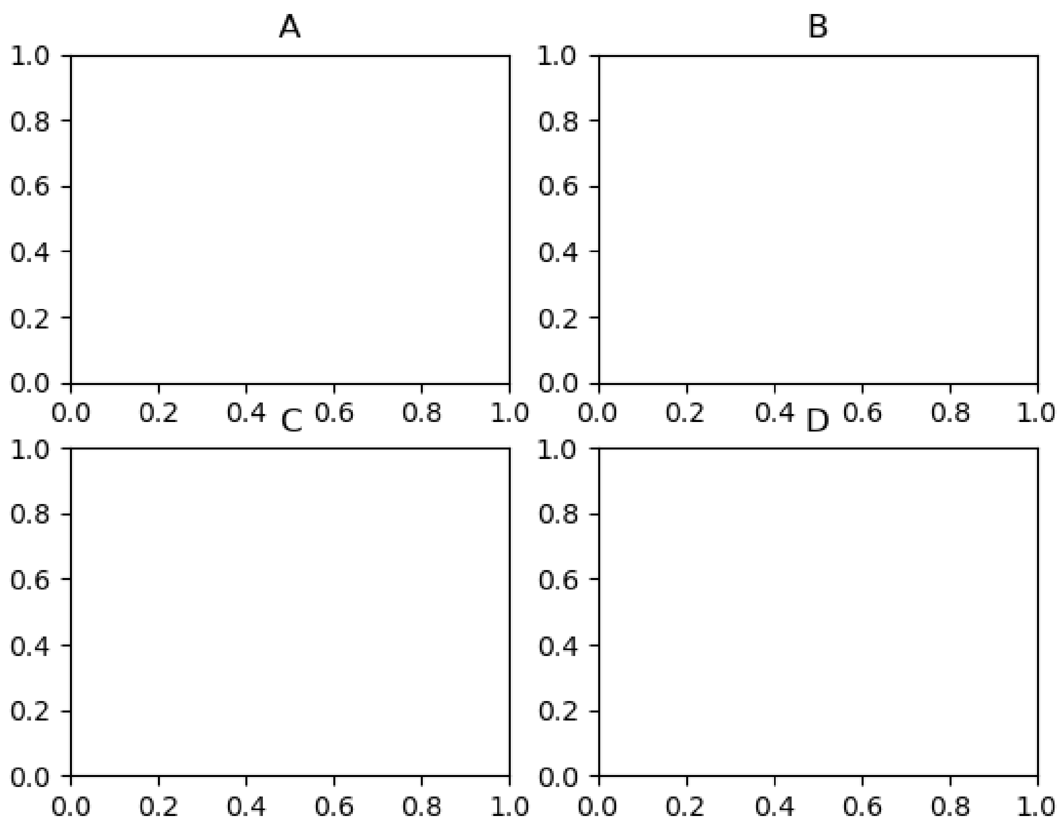
Another option is passing an `ax` argument to `Series.plot()` to plot on a particular axis:

```
In [144]: fig, axes = plt.subplots(nrows=2, ncols=2)
```

(continues on next page)

(continued from previous page)

```
In [145]: df['A'].plot(ax=axes[0, 0]);

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.

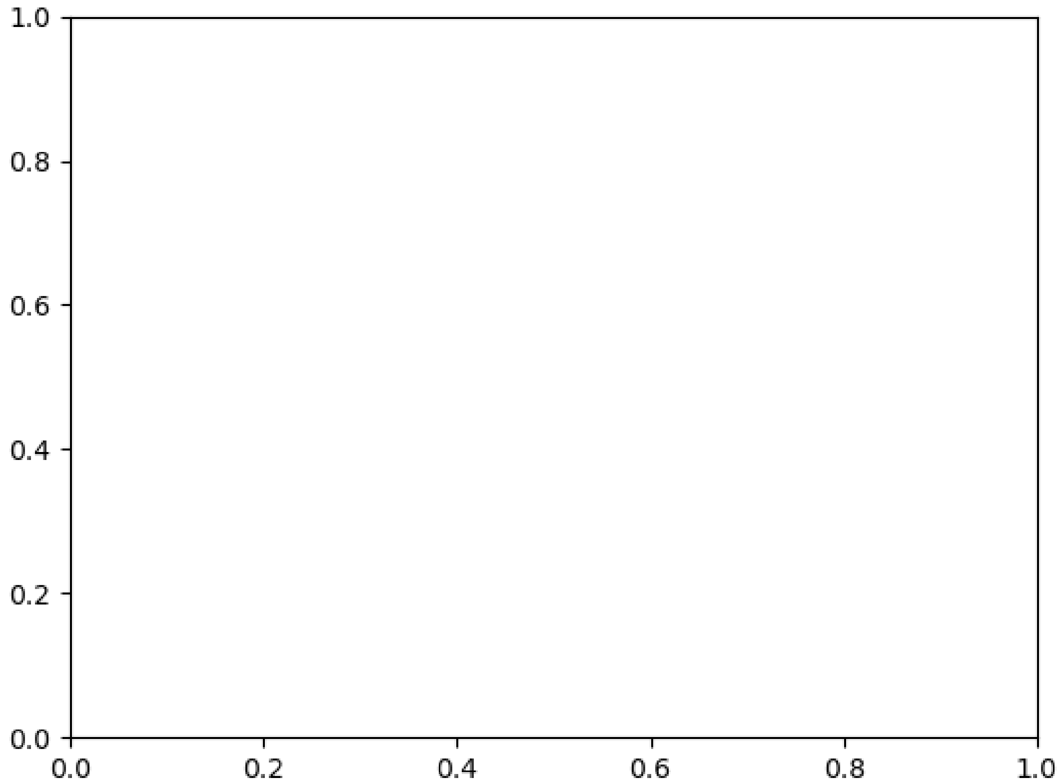
Here is an example of one way to easily plot group means with standard deviations from the raw data.

```
# 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'])
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-153-9f015fa171f2> in <module>
----> 1 ix3 = pd.MultiIndex.from_arrays([
      2     ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
      3     ['foo', 'foo', 'bar', 'bar', 'foo', 'foo', 'bar', 'bar']],
      4     names=['letter', 'word'])

NameError: name 'pd' is not defined

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)
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-154-a2b5068f0300> in <module>
----> 1 df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2],
      2                     'data2': [6, 5, 7, 5, 4, 5, 6, 5]}, index=ix3)

NameError: name 'pd' is not defined

# Group by index labels and take the means and standard deviations
# for each group
In [155]: gp3 = df3.groupby(level=('letter', 'word'))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-155-3f049b0a1791> in <module>
----> 1 gp3 = df3.groupby(level=('letter', 'word'))

NameError: name 'df3' is not defined

In [156]: means = gp3.mean()
```

(continues on next page)

(continued from previous page)

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-156-e6327c2cbb3d> in <module>
----> 1 means = gp3.mean()

NameError: name 'gp3' is not defined

In [157]: errors = gp3.std()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-157-e9e61accc58e> in <module>
----> 1 errors = gp3.std()

NameError: name 'gp3' is not defined

In [158]: means
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-158-88030acd958e> in <module>
----> 1 means

NameError: name 'means' is not defined

In [159]: errors
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-159-ab14c7b75346> in <module>
----> 1 errors

NameError: name 'errors' is not defined

# Plot
In [160]: fig, ax = plt.subplots()

In [161]: means.plot.bar(yerr=errors, ax=ax, capsize=4)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-161-60abb17bbd0c> in <module>
----> 1 means.plot.bar(yerr=errors, ax=ax, capsize=4)

NameError: name 'means' is not defined
```



Plotting tables

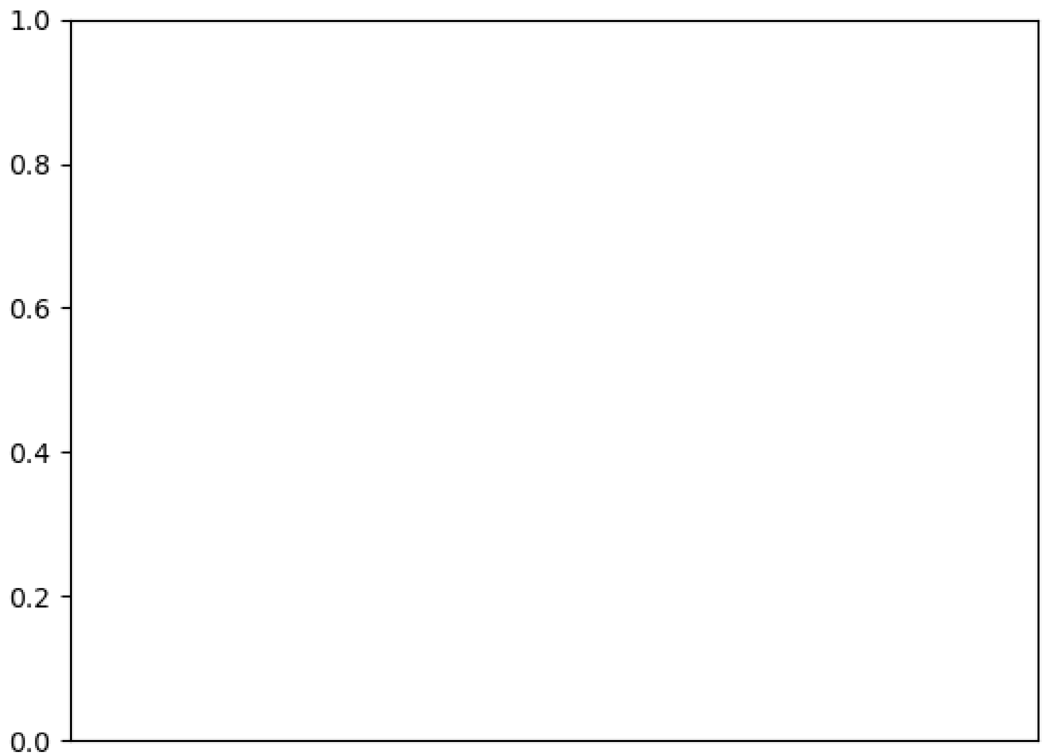
Plotting with matplotlib table is now supported in `DataFrame.plot()` and `Series.plot()` with a `table` keyword. The `table` keyword can accept `bool`, `DataFrame` or `Series`. The simple way to draw a table is to specify `table=True`. Data will be transposed to meet matplotlib's default layout.

```
In [162]: fig, ax = plt.subplots(1, 1)

In [163]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-163-05c0fbdb11a1> in <module>
----> 1 df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

NameError: name 'pd' is not defined

In [164]: ax.get_xaxis().set_visible(False)   # Hide Ticks

In [165]: df.plot(table=True, ax=ax)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-165-8624f4234fa9> in <module>
----> 1 df.plot(table=True, ax=ax)
```

(continues on next page)

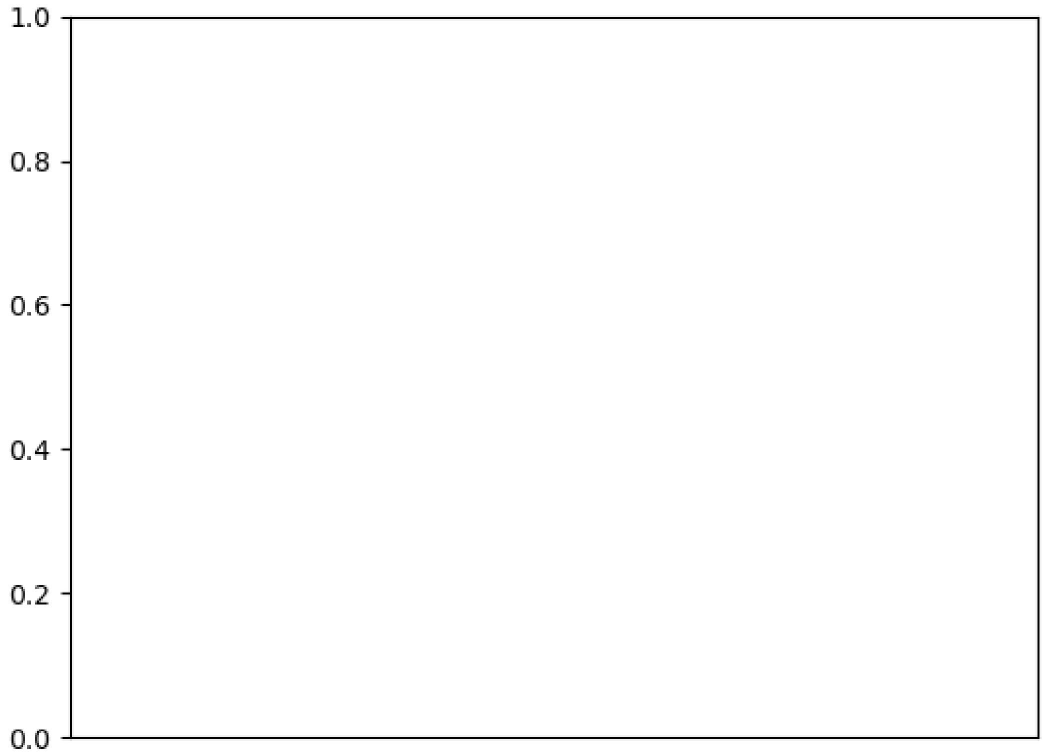
(continued from previous page)

```
NameError: name 'df' is not defined
```



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)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-168-d7698dba6372> in <module>
----> 1 df.plot(table=np.round(df.T, 2), ax=ax)

NameError: name 'df' is not defined
```



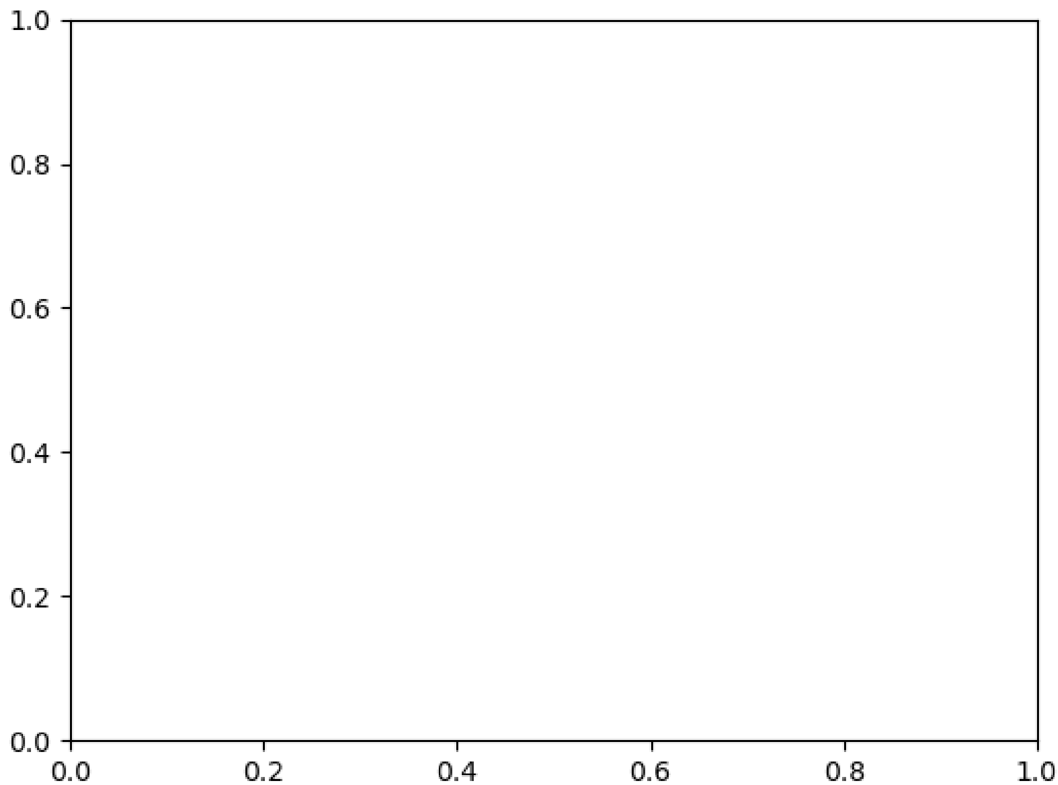
There also exists a helper function `pandas.plotting.table`, which creates a table from `DataFrame` or `Series`, and adds it to an `matplotlib.Axes` instance. This function can accept keywords which the matplotlib table has.

```
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])
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-171-c7487f392c87> in <module>
----> 1 table(ax, np.round(df.describe(), 2),
      2       loc='upper right', colWidths=[0.2, 0.2, 0.2])

NameError: name 'df' is not defined

In [172]: df.plot(ax=ax, ylim=(0, 2), legend=None)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-172-068255ff0f3e> in <module>
----> 1 df.plot(ax=ax, ylim=(0, 2), legend=None)
```

(continues on next page)

(continued from previous page)

```
NameError: name 'df' is not defined
```



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)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-173-73acce6fcaeb> in <module>
----> 1 df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
```

(continues on next page)

(continued from previous page)

```
NameError: name 'pd' is not defined

In [174]: df = df.cumsum()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-174-08208d45ae16> in <module>
----> 1 df = df.cumsum()

NameError: name 'df' is not defined

In [175]: plt.figure()
Out[175]: <Figure size 640x480 with 0 Axes>

In [176]: df.plot(colormap='cubehelix')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-176-0aab5a23aeee> in <module>
----> 1 df.plot(colormap='cubehelix')

NameError: name 'df' is not defined
```

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)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-179-7cdc1499f1cb> in <module>
----> 1 df.plot(colormap=cm.cubehelix)

NameError: name 'df' is not defined
```

Colormaps can also be used other plot types, like bar charts:

```
In [180]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-180-2d4edaa33d2e> in <module>
----> 1 dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)

NameError: name 'pd' is not defined

In [181]: dd = dd.cumsum()
---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
NameError                                 Traceback (most recent call last)
<ipython-input-181-cf596e929dc1> in <module>
----> 1 dd = dd.cumsum()

NameError: name 'dd' is not defined

In [182]: plt.figure()
Out[182]: <Figure size 640x480 with 0 Axes>

In [183]: dd.plot.bar(colormap='Greens')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-183-d5bc68809546> in <module>
----> 1 dd.plot.bar(colormap='Greens')

NameError: name 'dd' is not defined
```

Parallel coordinates charts:

```
In [184]: plt.figure()
Out[184]: <Figure size 640x480 with 0 Axes>

In [185]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
NameError                                 Traceback (most recent call last)
<ipython-input-185-a0c62c912a5a> in <module>
----> 1 parallel_coordinates(data, 'Name', colormap='gist_rainbow')

NameError: name 'data' is not defined
```

Andrews curves charts:

```
In [186]: plt.figure()
Out[186]: <Figure size 640x480 with 0 Axes>

In [187]: andrews_curves(data, 'Name', colormap='winter')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-187-3fe5a5a07312> in <module>
----> 1 andrews_curves(data, 'Name', colormap='winter')

NameError: name 'data' is not defined
```

2.11.6 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'))
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-188-c269685eca94> in <module>
----> 1 price = pd.Series(np.random.randn(150).cumsum(),
      2                   index=pd.date_range('2000-1-1', periods=150, freq='B'))

NameError: name 'pd' is not defined

In [189]: ma = price.rolling(20).mean()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-189-7dcf1e53fe5c> in <module>
----> 1 ma = price.rolling(20).mean()

NameError: name 'price' is not defined

In [190]: mstd = price.rolling(20).std()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-190-e2f8c3d51887> in <module>
----> 1 mstd = price.rolling(20).std()

NameError: name 'price' is not defined

In [191]: plt.figure()
Out[191]: <Figure size 640x480 with 0 Axes>

In [192]: plt.plot(price.index, price, 'k')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-192-7bb1b226415a> in <module>
----> 1 plt.plot(price.index, price, 'k')

NameError: name 'price' is not defined

In [193]: plt.plot(ma.index, ma, 'b')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-193-3728ccc65de7> in <module>
----> 1 plt.plot(ma.index, ma, 'b')

NameError: name 'ma' is not defined

In [194]: plt.fill_between(mstd.index, ma - 2 * mstd, ma + 2 * mstd,
   .....:                  color='b', alpha=0.2)
   .....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-194-ba00db352f3f> in <module>
----> 1 plt.fill_between(mstd.index, ma - 2 * mstd, ma + 2 * mstd,
      2                  color='b', alpha=0.2)

NameError: name 'mstd' is not defined
```

2.12 Computational tools

2.12.1 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   -1.602976
2    4.334938
3   -0.247456
4   -2.067345
5   -1.142903
6   -1.688214
7   -9.759729
dtype: float64
```

```
In [3]: df = 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 -0.218320 -1.054001  1.987147 -0.510183
4 -0.439121 -1.816454  0.649715 -4.822809
5 -0.127833 -3.042065 -5.866604 -1.776977
6 -2.596833 -1.959538 -2.111697 -3.798900
7 -0.117826 -2.169058  0.036094 -0.067696
8  2.492606 -1.357320 -1.205802 -1.558697
9 -1.012977  2.324558 -1.003744 -0.371806
```

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]: 0.0006801088174310875
```

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  1.000882 -0.003177 -0.002698 -0.006889  0.031912
b -0.003177  1.024721  0.000191  0.009212  0.000857
c -0.002698  0.000191  0.950735 -0.031743 -0.005087
d -0.006889  0.009212 -0.031743  1.002983 -0.047952
e  0.031912  0.000857 -0.005087 -0.047952  1.042487
```

`DataFrame.cov` also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```
In [10]: frame = 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  1.123670 -0.412851  0.018169
b -0.412851  1.154141  0.305260
c  0.018169  0.305260  1.301149

In [14]: frame.cov(min_periods=12)
Out[14]: 
          a         b         c
a  1.123670       NaN  0.018169
b       NaN  1.154141  0.305260
c  0.018169  0.305260  1.301149
```

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
- Spearman's 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]: 0.013479040400098775

In [18]: frame['a'].corr(frame['b'], method='spearman')
Out[18]: -0.007289885159540637
```

(continues on next page)

(continued from previous page)

```
# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:
          a         b         c         d         e
a  1.000000  0.013479 -0.049269 -0.042239 -0.028525
b  0.013479  1.000000 -0.020433 -0.011139  0.005654
c -0.049269 -0.020433  1.000000  0.018587 -0.054269
d -0.042239 -0.011139  0.018587  1.000000 -0.017060
e -0.028525  0.005654 -0.054269 -0.017060  1.000000
```

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

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```
In [20]: frame = 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 -0.121111  0.069544
b -0.121111  1.000000  0.051742
c  0.069544  0.051742  1.000000

In [24]: frame.corr(min_periods=12)
Out[24]:
          a         b         c
a  1.000000       NaN  0.069544
b       NaN  1.000000  0.051742
c  0.069544  0.051742  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  -6.404882  -2.058431
b -6.404882   1.000000 -19.255743
c -2.058431 -19.255743   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     -0.125501
two     -0.493244
three    0.344056
four     0.004183
dtype: float64

In [32]: df2.corrwith(df1, axis=1)
Out[32]:
a   -0.675817
b    0.458296
c    0.190809
d   -0.186275
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

In [35]: s.rank()
Out[35]:
a    5.0
b    2.5
c    1.0
d    2.5
e    4.0
dtype: float64
```

rank() is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). `NaN` values are excluded from the ranking.

```
In [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 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
```

(continues on next page)

(continued from previous page)

```
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.204848       NaN  0.463347
6  0.376892  0.959292  0.095572 -0.593740       NaN -0.069180
7 -1.002601  1.957794 -0.120708  0.094214       NaN -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254       NaN -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477       NaN  1.375064

In [39]: df.rank(1)
Out[39]:
     0    1    2    3    4    5
0  4.0  3.0  1.5  5.0  1.5  6.0
1  2.0  6.0  4.5  1.0  4.5  3.0
2  1.0  6.0  3.5  5.0  3.5  2.0
3  4.0  5.0  1.5  3.0  1.5  6.0
4  5.0  3.0  1.5  4.0  1.5  6.0
5  1.0  2.0  5.0  3.0  NaN  4.0
6  4.0  5.0  3.0  1.0  NaN  2.0
7  2.0  5.0  3.0  4.0  NaN  1.0
8  2.0  5.0  3.0  4.0  NaN  1.0
9  2.0  3.0  1.0  4.0  NaN  5.0
```

`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

2.12.2 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()
```

(continues on next page)

(continued from previous page)

```
In [42]: s
Out[42]:
2000-01-01    -0.268824
2000-01-02    -1.771855
2000-01-03    -0.818003
2000-01-04    -0.659244
2000-01-05    -1.942133
                 ...
2002-09-22   -67.457323
2002-09-23   -69.253182
2002-09-24   -70.296818
2002-09-25   -70.844674
2002-09-26   -72.475016
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:

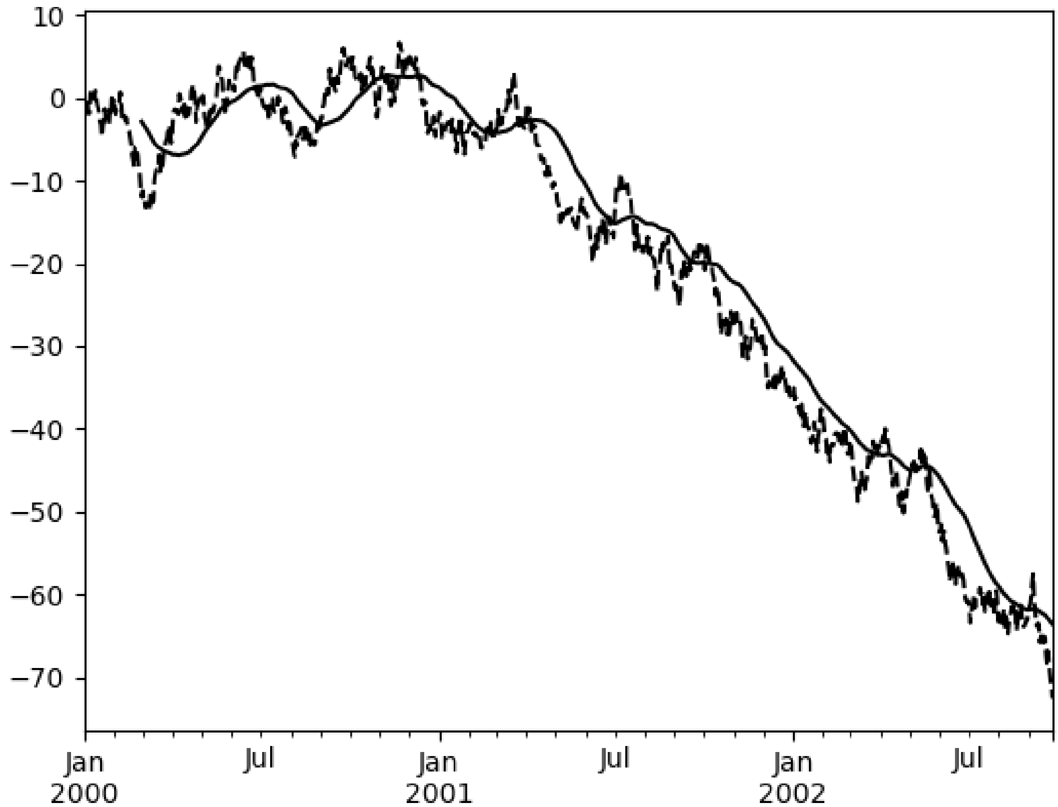
```
In [45]: r.mean()
Out[45]:
2000-01-01          NaN
2000-01-02          NaN
2000-01-03          NaN
2000-01-04          NaN
2000-01-05          NaN
                 ...
2002-09-22   -62.914971
2002-09-23   -63.061867
2002-09-24   -63.213876
2002-09-25   -63.375074
2002-09-26   -63.539734
Freq: D, Length: 1000, dtype: float64
```

```
In [46]: s.plot(style='k--')
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5340572210>
```

(continues on next page)

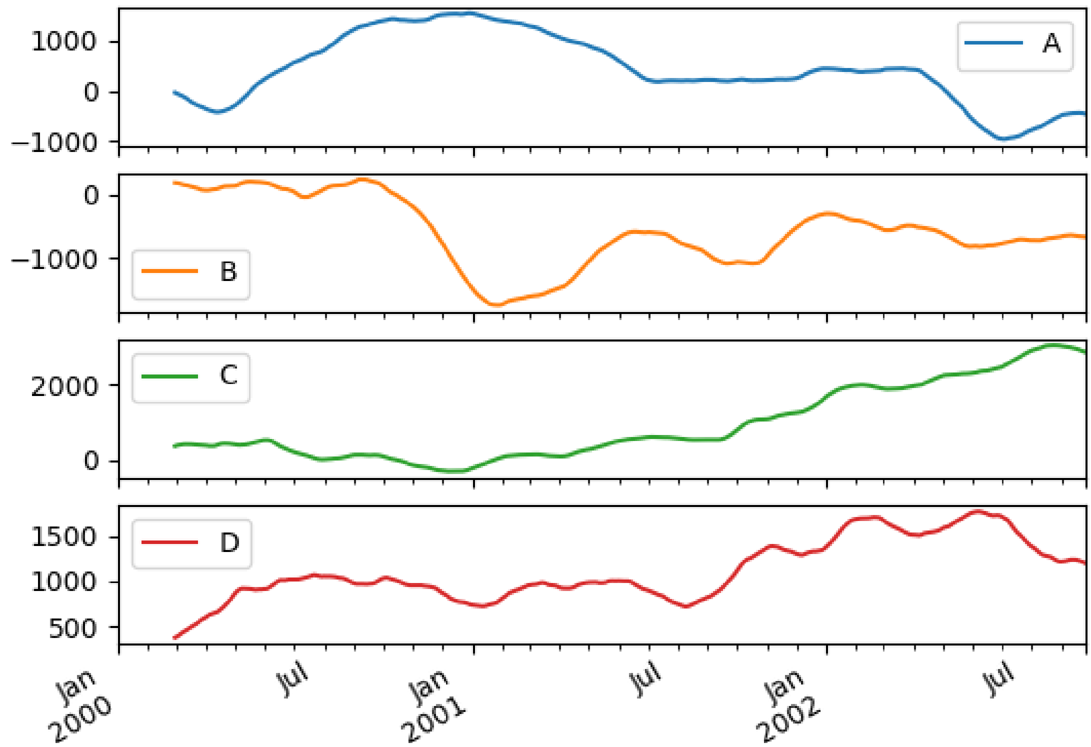
(continued from previous page)

```
In [47]: r.mean().plot(style='k')
Out[47]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5340572210>
```



They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrame's columns:

```
In [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 0x7f533d9754d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f533d925290>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f533d97efd0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f533d8e79d0>],
      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)