

Visualization

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt
```

The plots in this document are made using matplotlib's `ggplot` style (new in version 1.4):

```
import matplotlib
matplotlib.style.use('ggplot')
```

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.

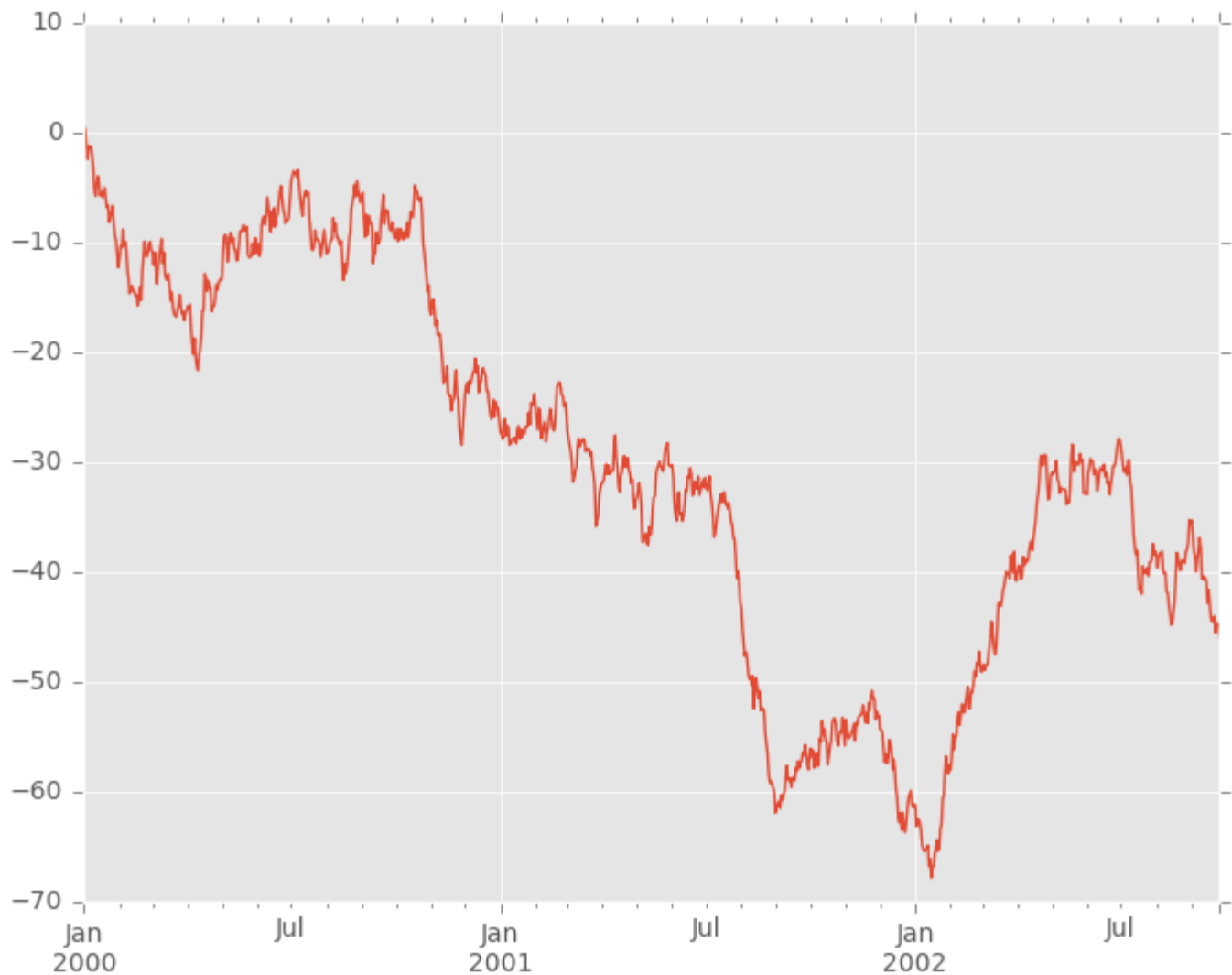
Note: All calls to `np.random` are seeded with 123456.

Basic Plotting: `plot`

See the [cookbook](#) for some advanced strategies

The `plot` method on `Series` and `DataFrame` is just a simple wrapper around `plt.plot()`:

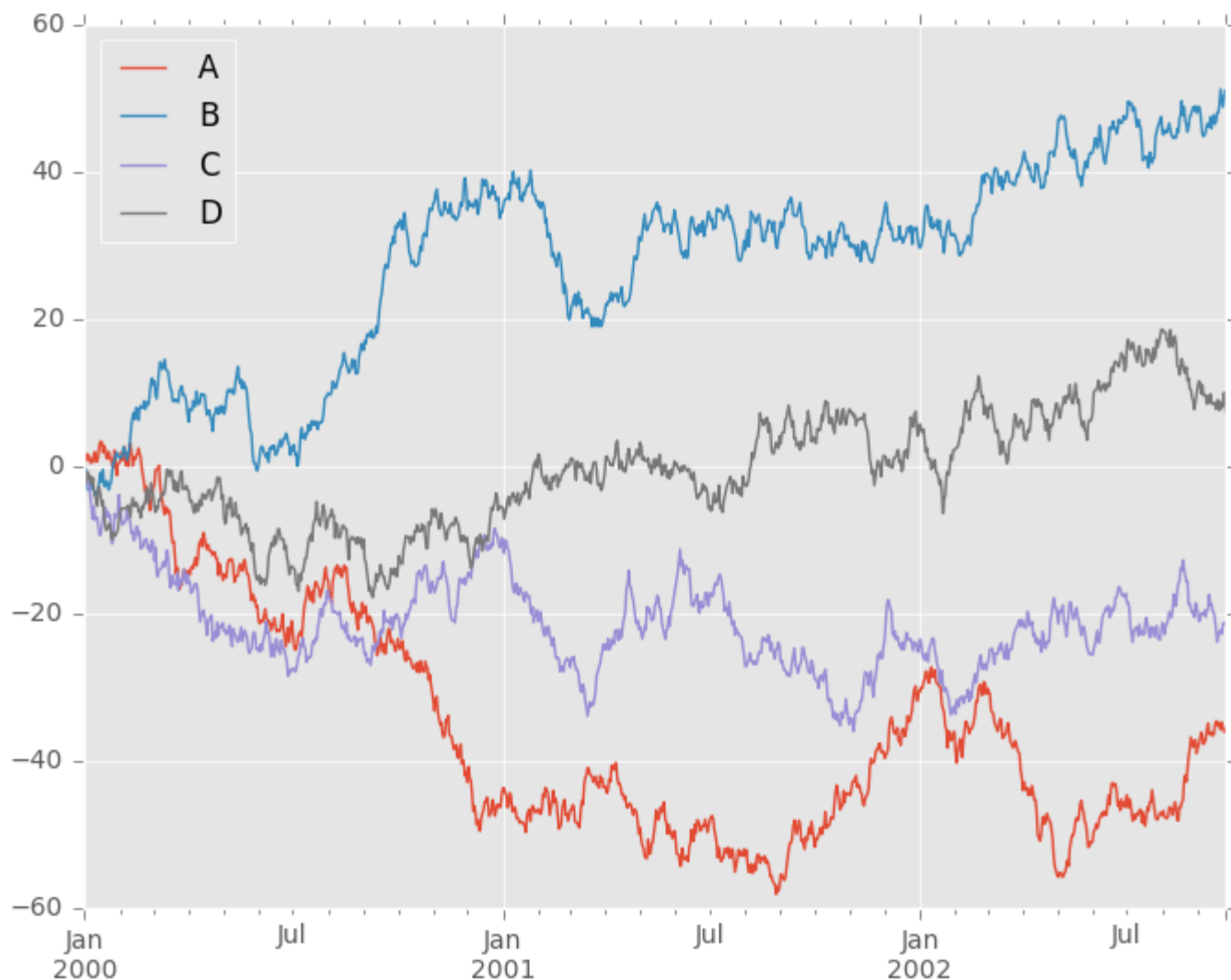
```
In [2]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [3]: ts = ts.cumsum()
In [4]: ts.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8236550>
```



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 [5]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))  
In [6]: df = df.cumsum()  
In [7]: plt.figure(); df.plot();
```



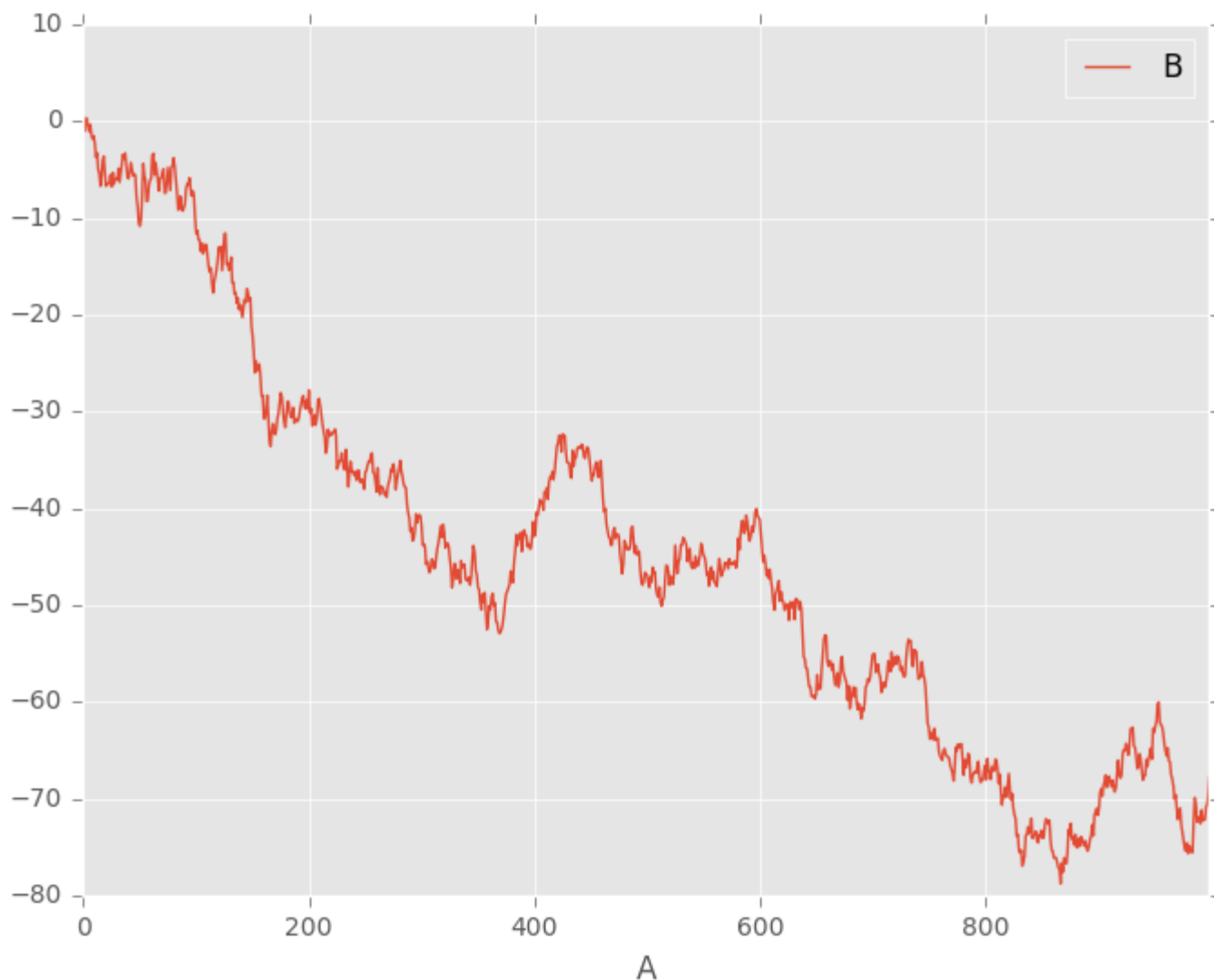
You can plot one column versus another using the `x` and `y` keywords in `plot()`:

```
In [8]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
```

```
In [9]: df3['A'] = pd.Series(list(range(len(df))))
```

```
In [10]: df3.plot(x='A', y='B')
```

```
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8e23650>
```



Note: For more formatting and styling options, see [below](#).

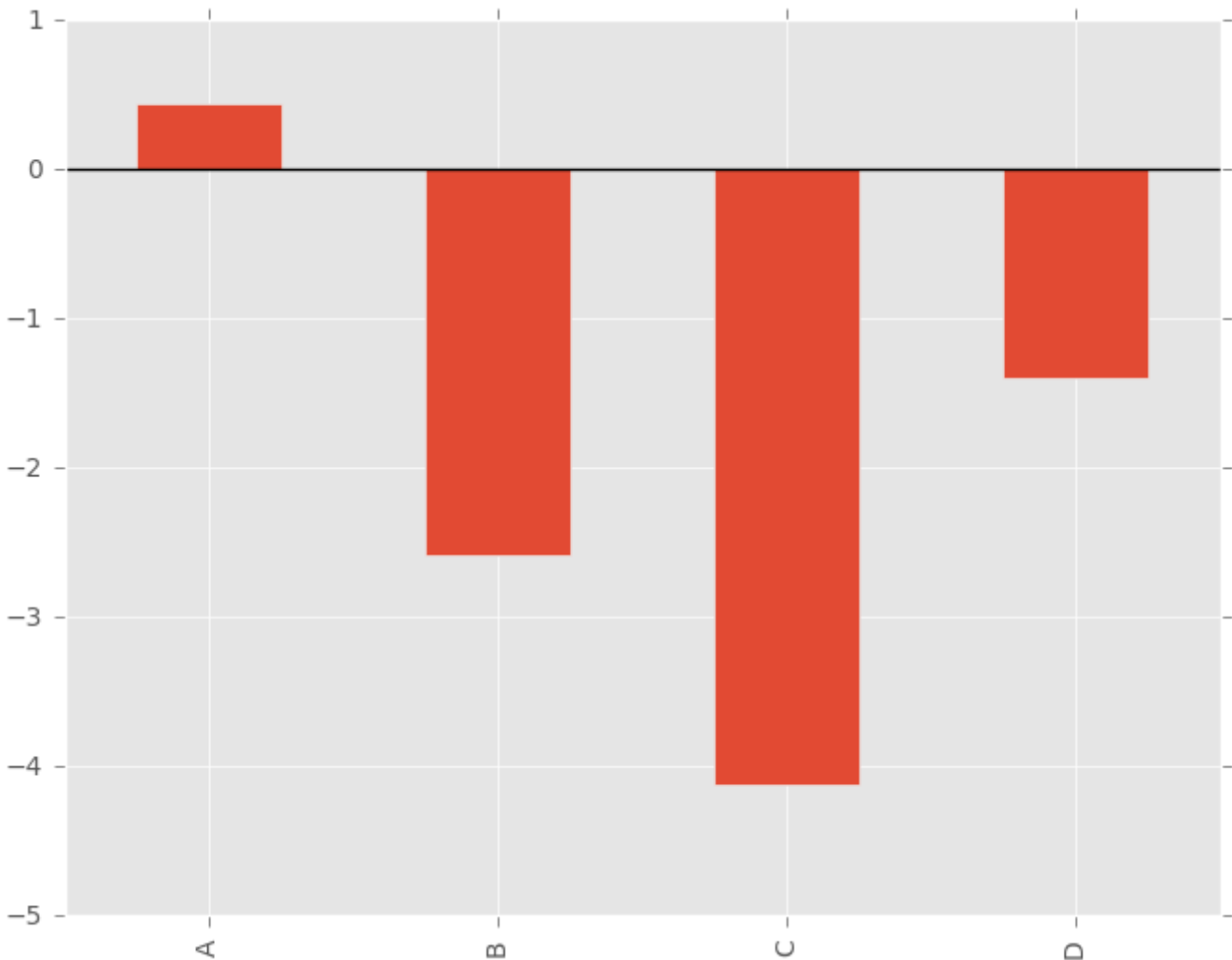
Other Plots

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()`. These 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 [11]: plt.figure();
In [12]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
Out[12]: <matplotlib.lines.Line2D at 0x7f3db938d650>
```



New in version 0.17.0.

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 [13]: df = pd.DataFrame()

In [14]: df.plot.<TAB>
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.tools.plotting` that take a [Series](#) or [DataFrame](#) as an argument. These include

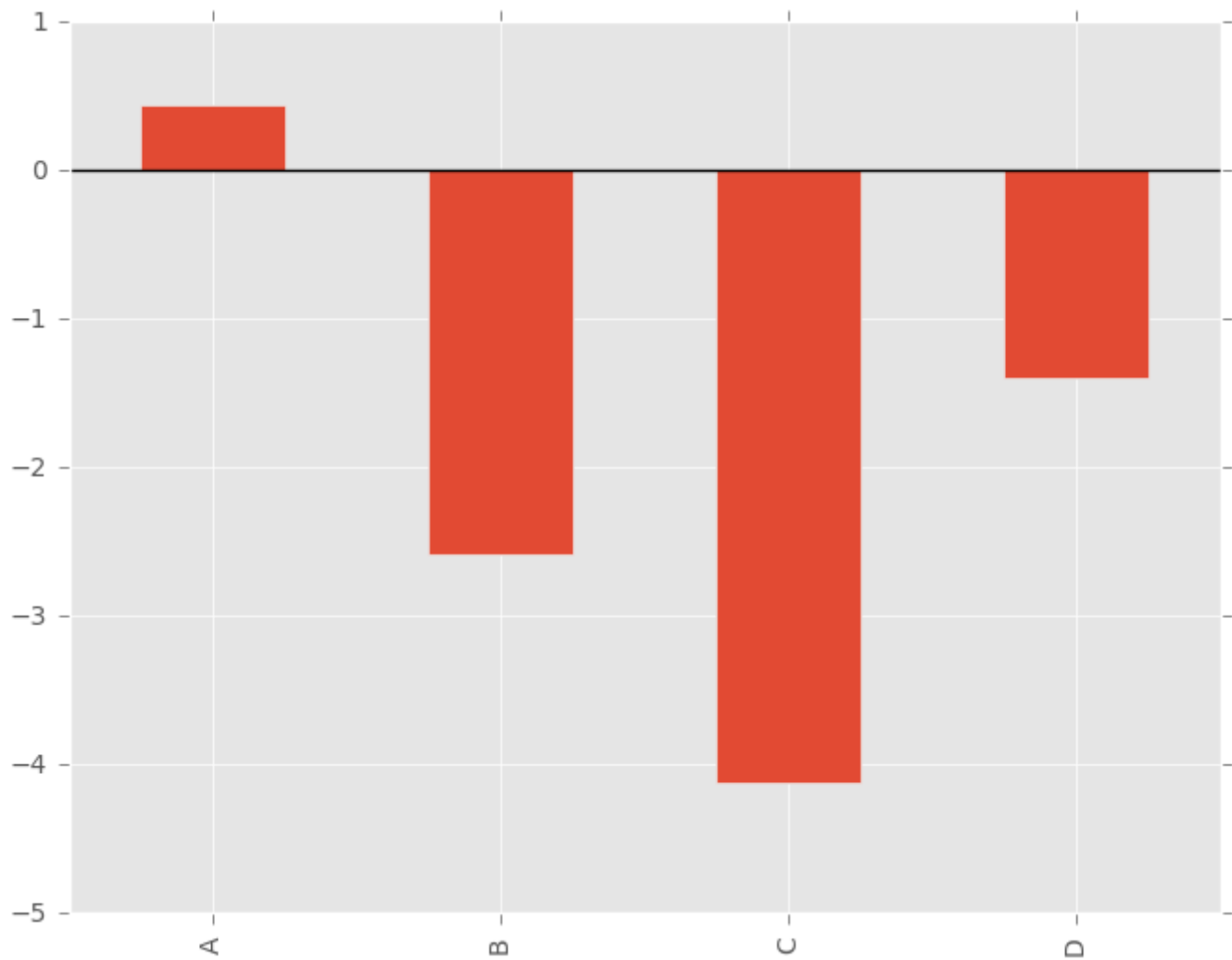
- [Scatter Matrix](#)
- [Andrews Curves](#)
- [Parallel Coordinates](#)
- [Lag Plot](#)
- [Autocorrelation Plot](#)
- [Bootstrap Plot](#)
- [RadViz](#)

Plots may also be adorned with [errorbars](#) or [tables](#).

Bar plots

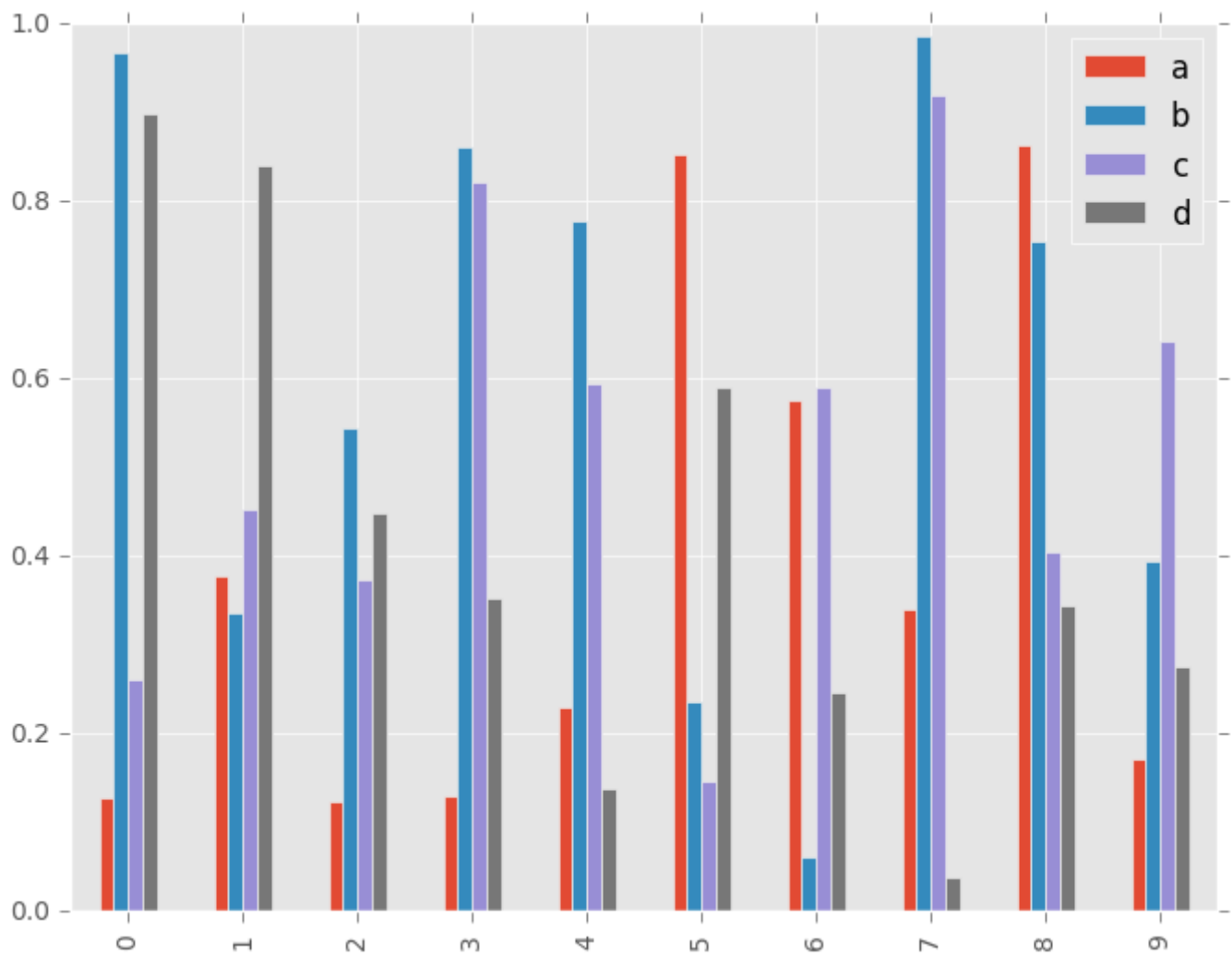
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [15]: plt.figure();  
  
In [16]: df.ix[5].plot.bar(); plt.axhline(0, color='k')  
Out[16]: <matplotlib.lines.Line2D at 0x7f3db9d26790>
```



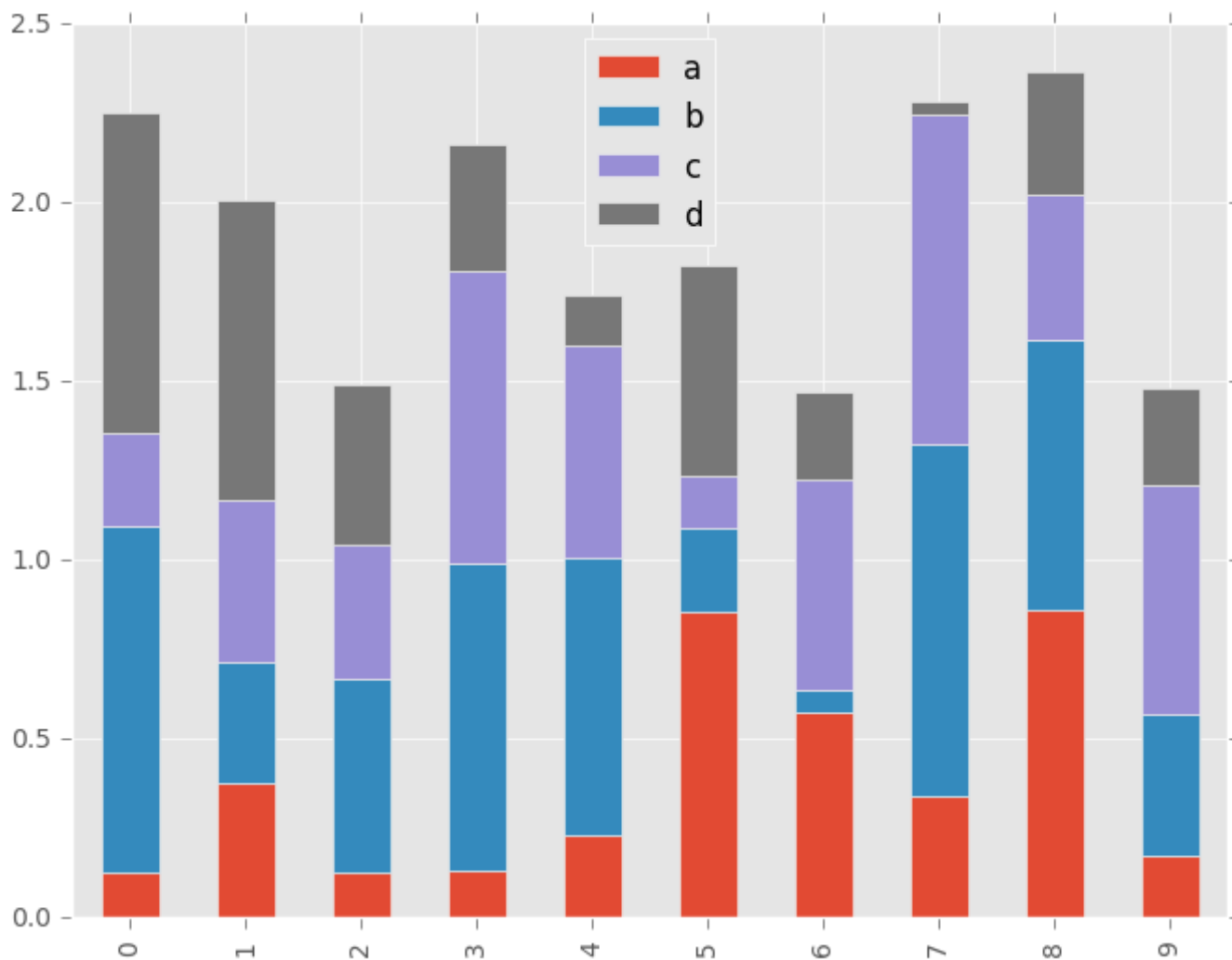
Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

```
In [17]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])  
In [18]: df2.plot.bar();
```



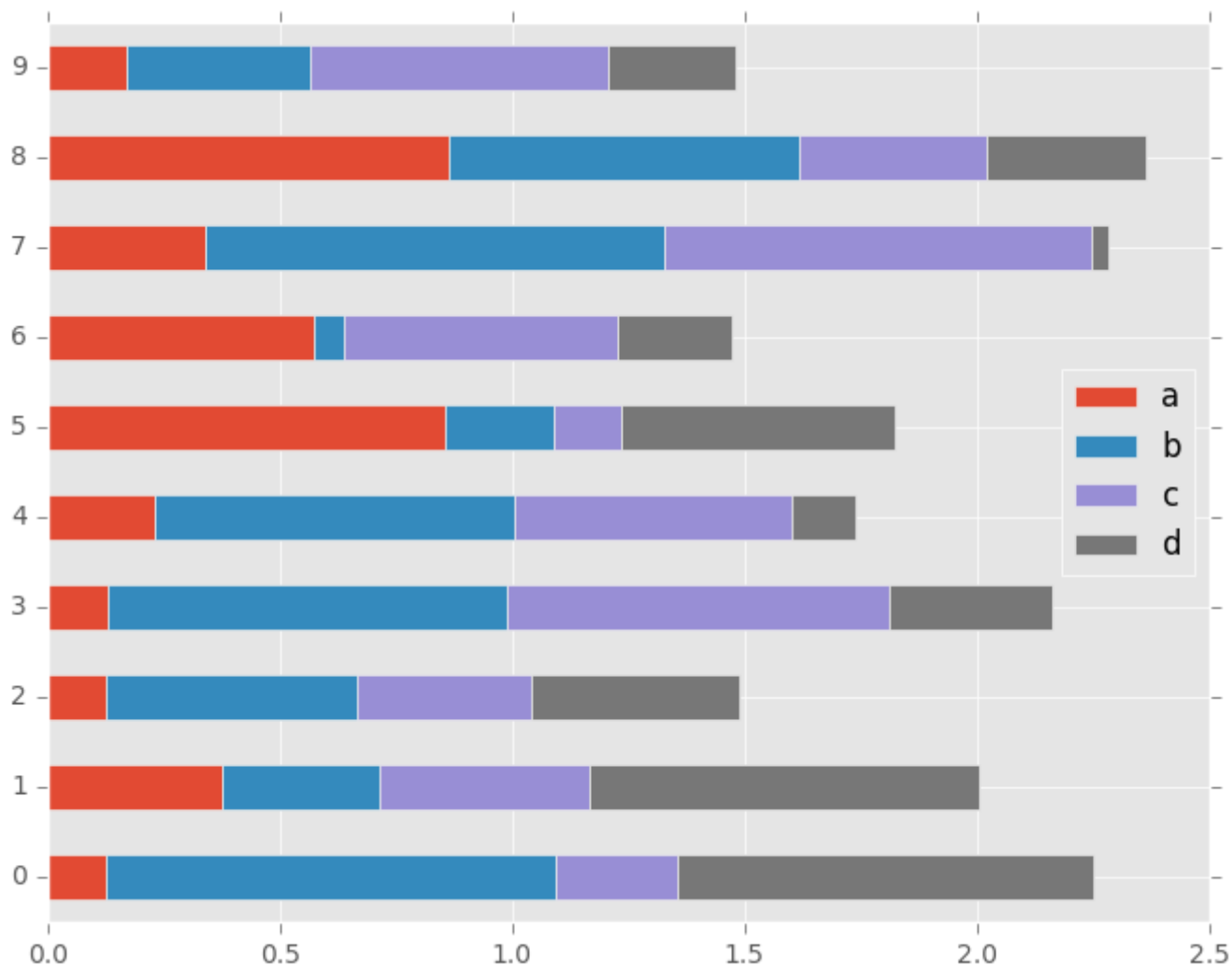
To produce a stacked bar plot, pass `stacked=True`:

```
In [19]: df2.plot.bar(stacked=True);
```

To get horizontal bar plots, use the `barh` method:

```
In [20]: df2.plot.barh(stacked=True);
```



Histograms

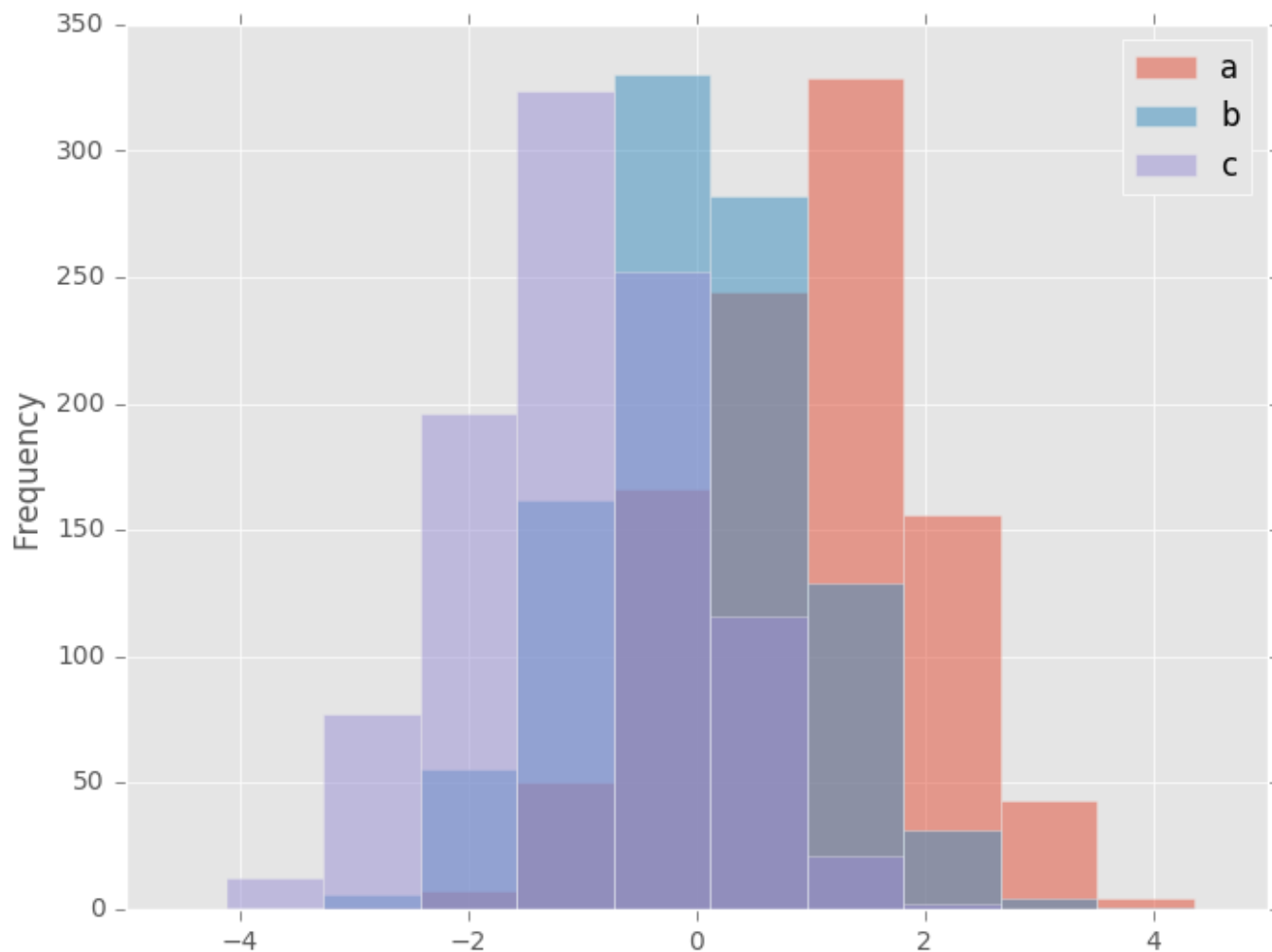
New in version 0.15.0.

Histogram can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [21]: 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 [22]: plt.figure();

In [23]: df4.plot.hist(alpha=0.5)
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dba553b10>
```

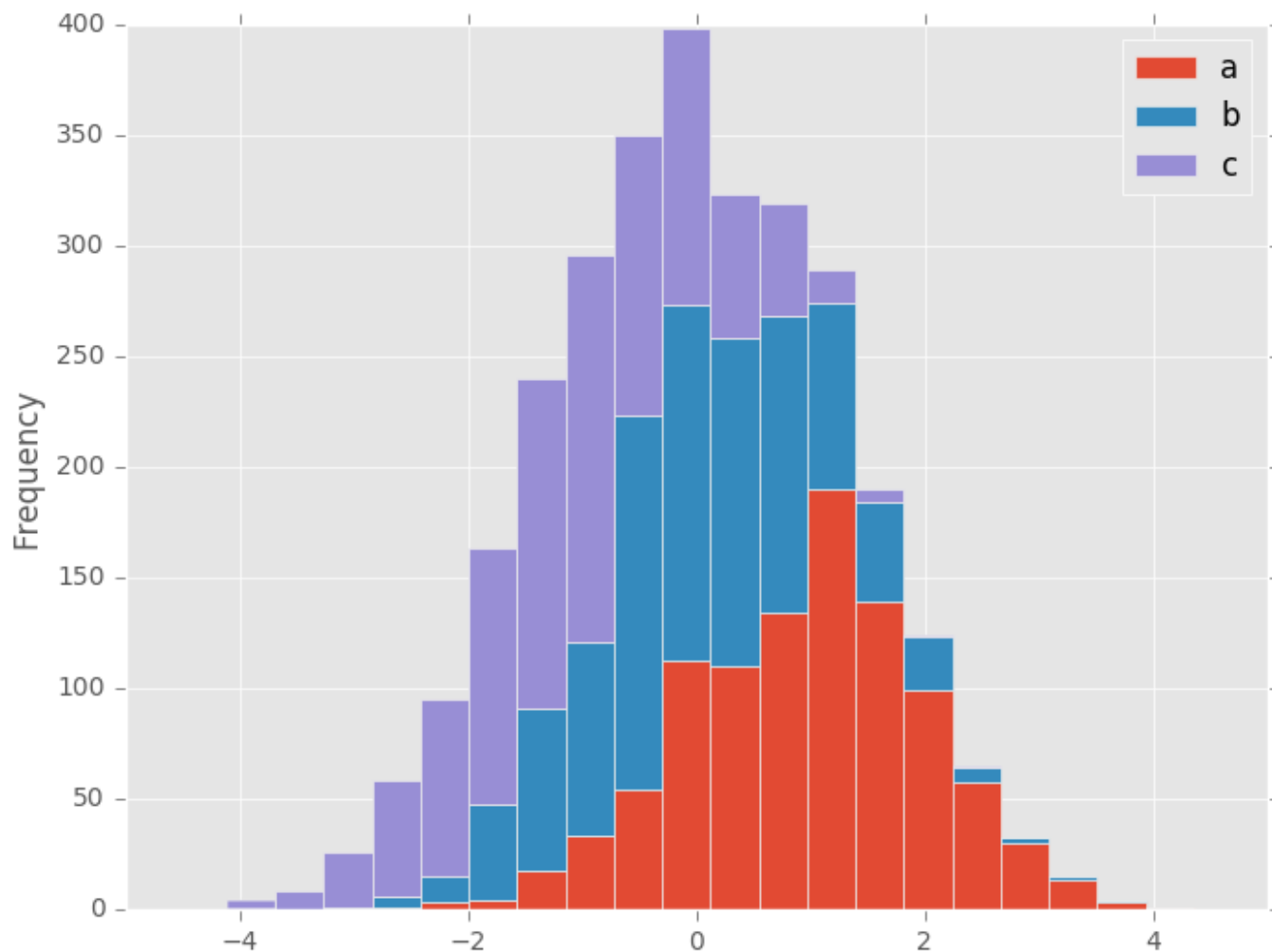


Histogram can be stacked by `stacked=True`. Bin size can be changed by `bins` keyword.

```
In [24]: plt.figure();
```

```
In [25]: df4.plot.hist(stacked=True, bins=20)
```

```
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbd3fd3d0>
```

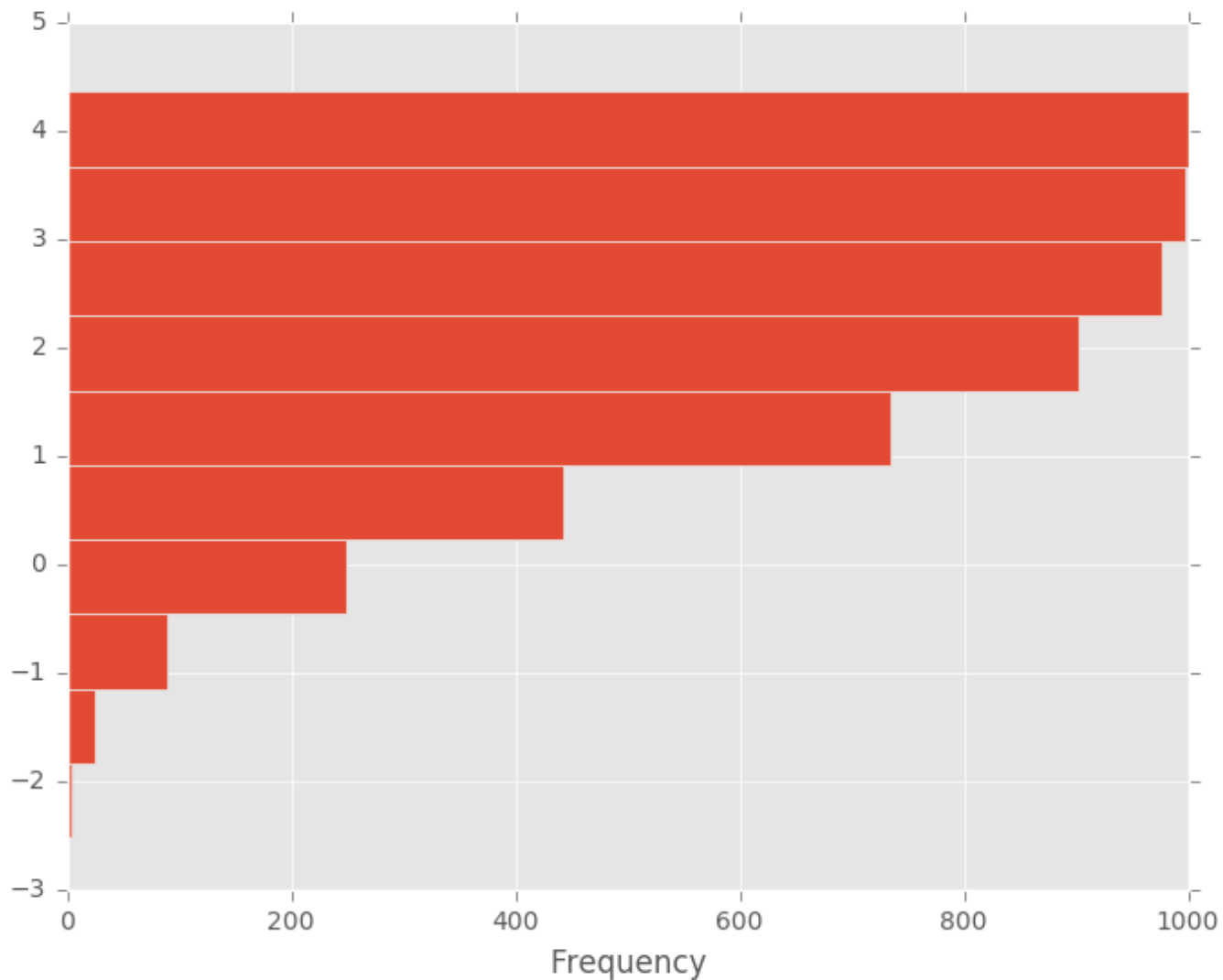


You can pass other keywords supported by matplotlib `hist`. For example, horizontal and cumulative histogram can be drawn by `orientation='horizontal'` and `cumulative=True`.

```
In [26]: plt.figure();
```

```
In [27]: df4['a'].plot.hist(orientation='horizontal', cumulative=True)
```

```
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db3c7ec90>
```



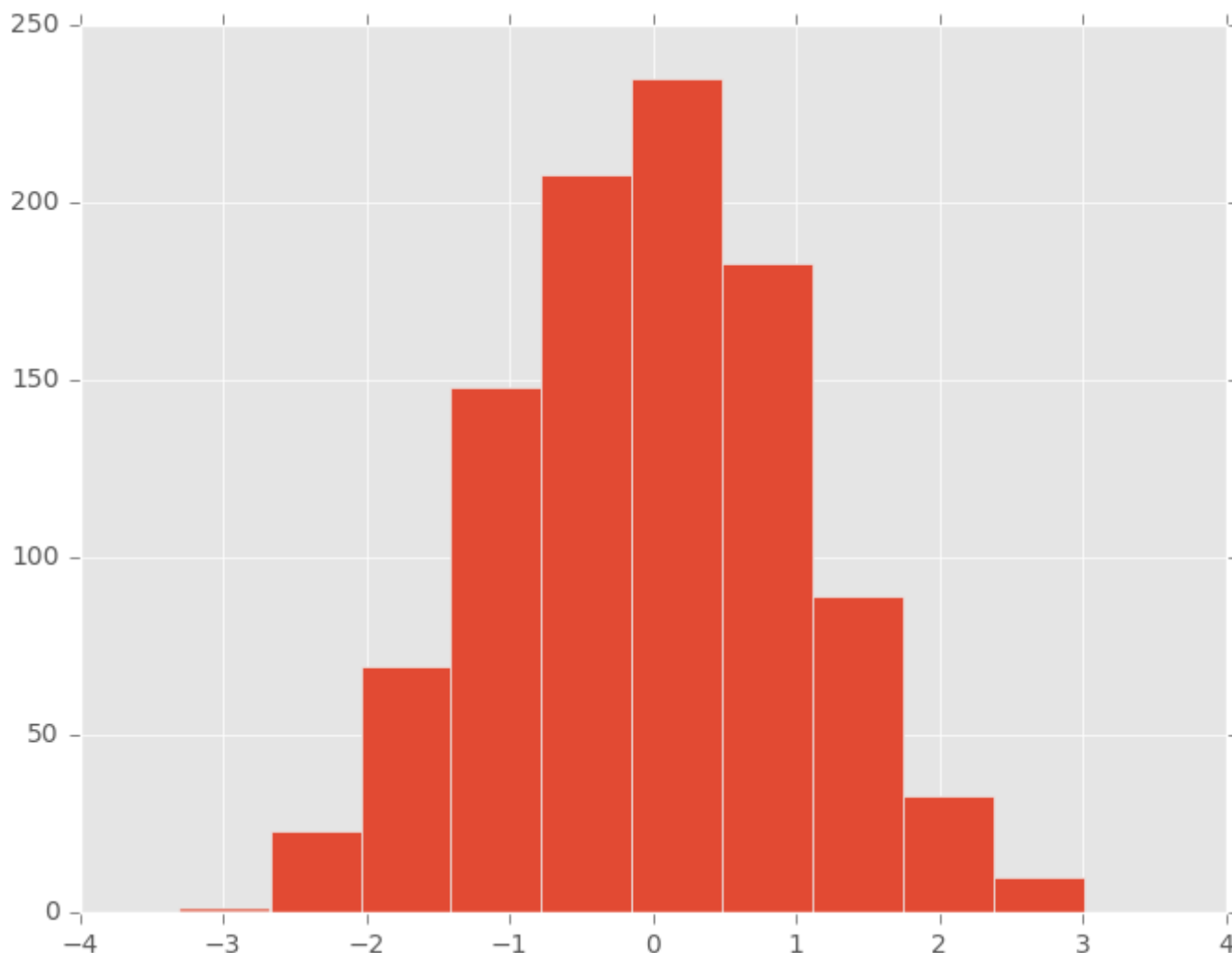
See the [hist](#) method and the [matplotlib hist documentation](#) for more.

The existing interface `DataFrame.hist` to plot histogram still can be used.

```
In [28]: plt.figure();
```

```
In [29]: df['A'].diff().hist()
```

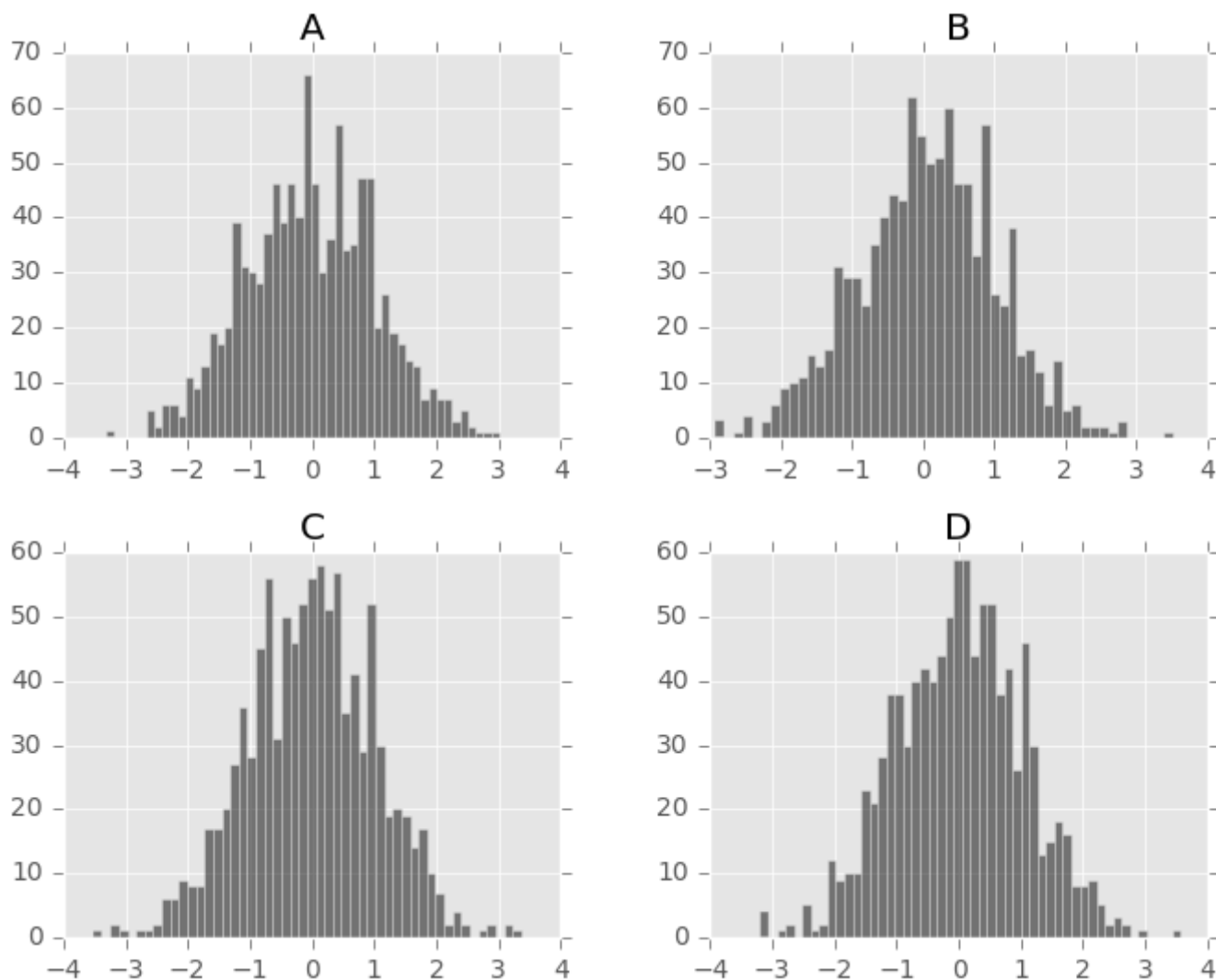
```
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db3c55ed0>
```



`DataFrame.hist()` plots the histograms of the columns on multiple subplots:

```
In [30]: plt.figure()
Out[30]: <matplotlib.figure.Figure at 0x7f3db3dfaed0>

In [31]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[31]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f3db3c11950>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db96ebbd0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f3db8539750>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db8a95b50>]], dtype=object)
```

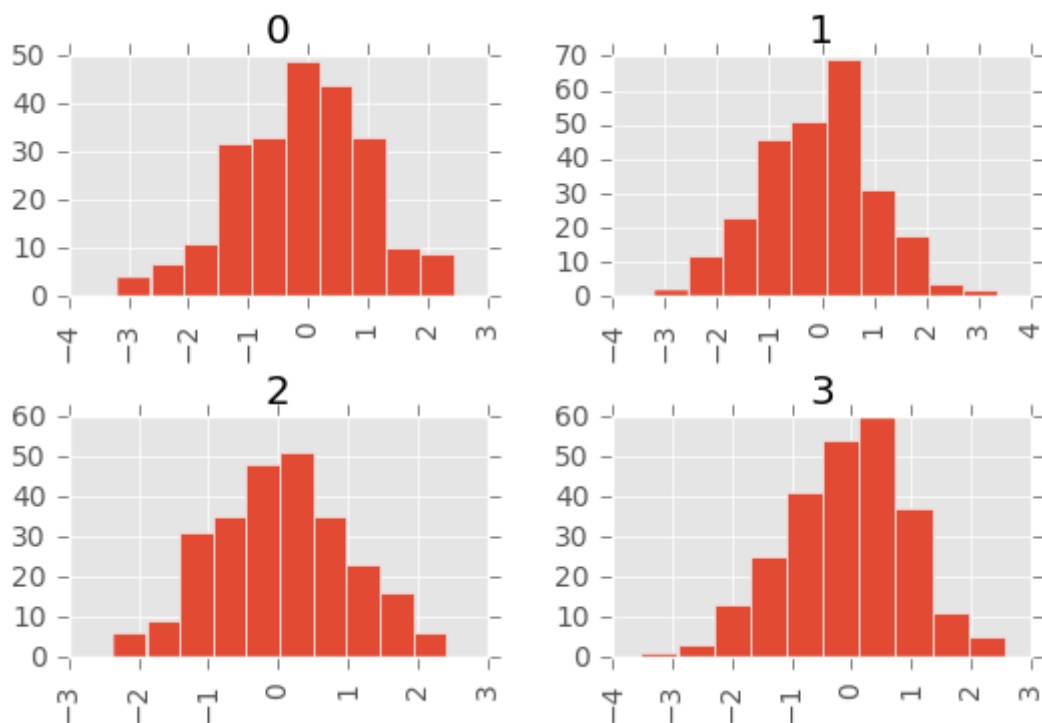


New in version 0.10.0.

The `by` keyword can be specified to plot grouped histograms:

```
In [32]: data = pd.Series(np.random.randn(1000))

In [33]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[33]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbadedb90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbab94890>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x7f3dba302190>,
         <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbac87710>]], dtype=object)
```



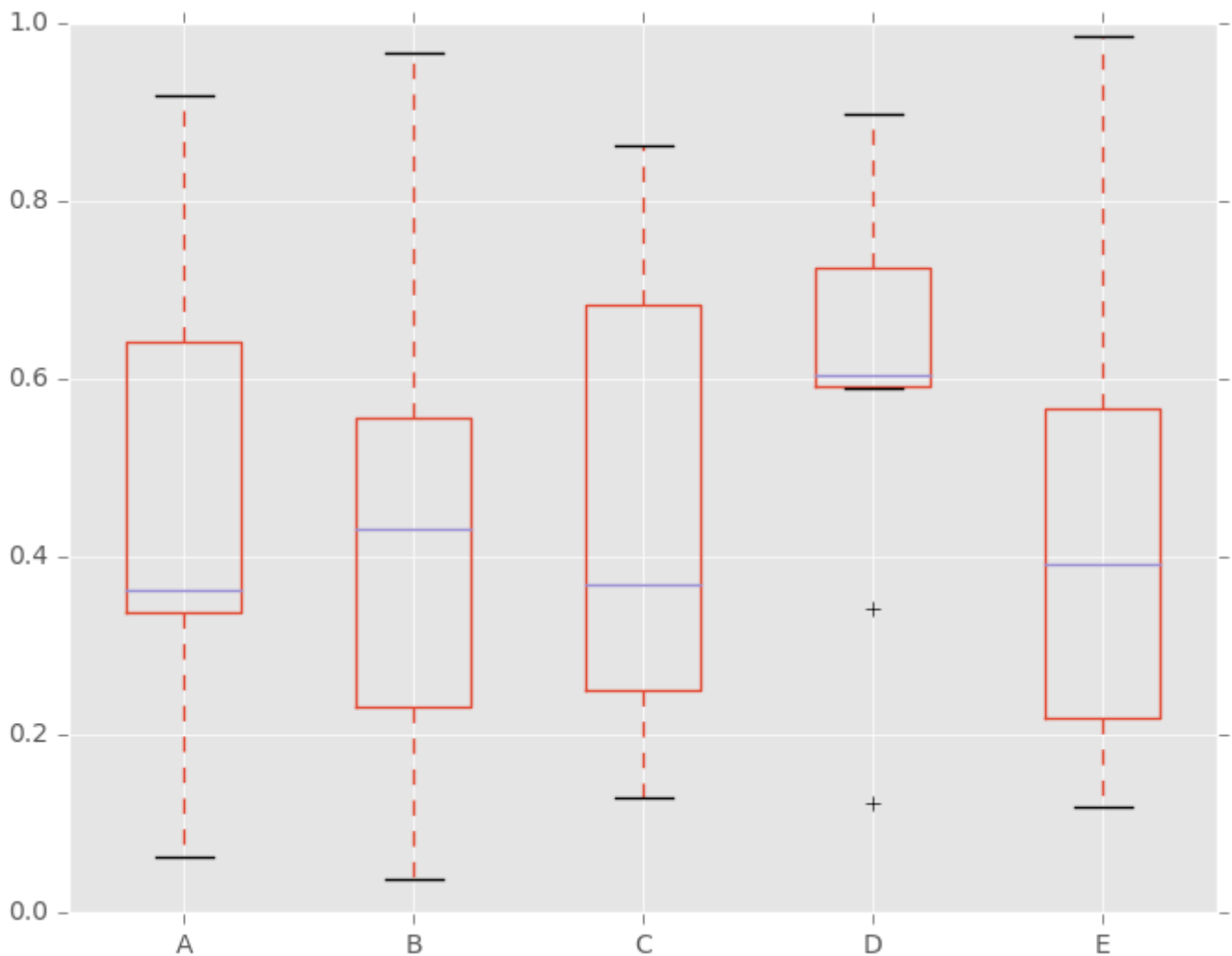
Box Plots

New in version 0.15.0.

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 [34]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])  
In [35]: df.plot.box()  
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dba861b50>
```

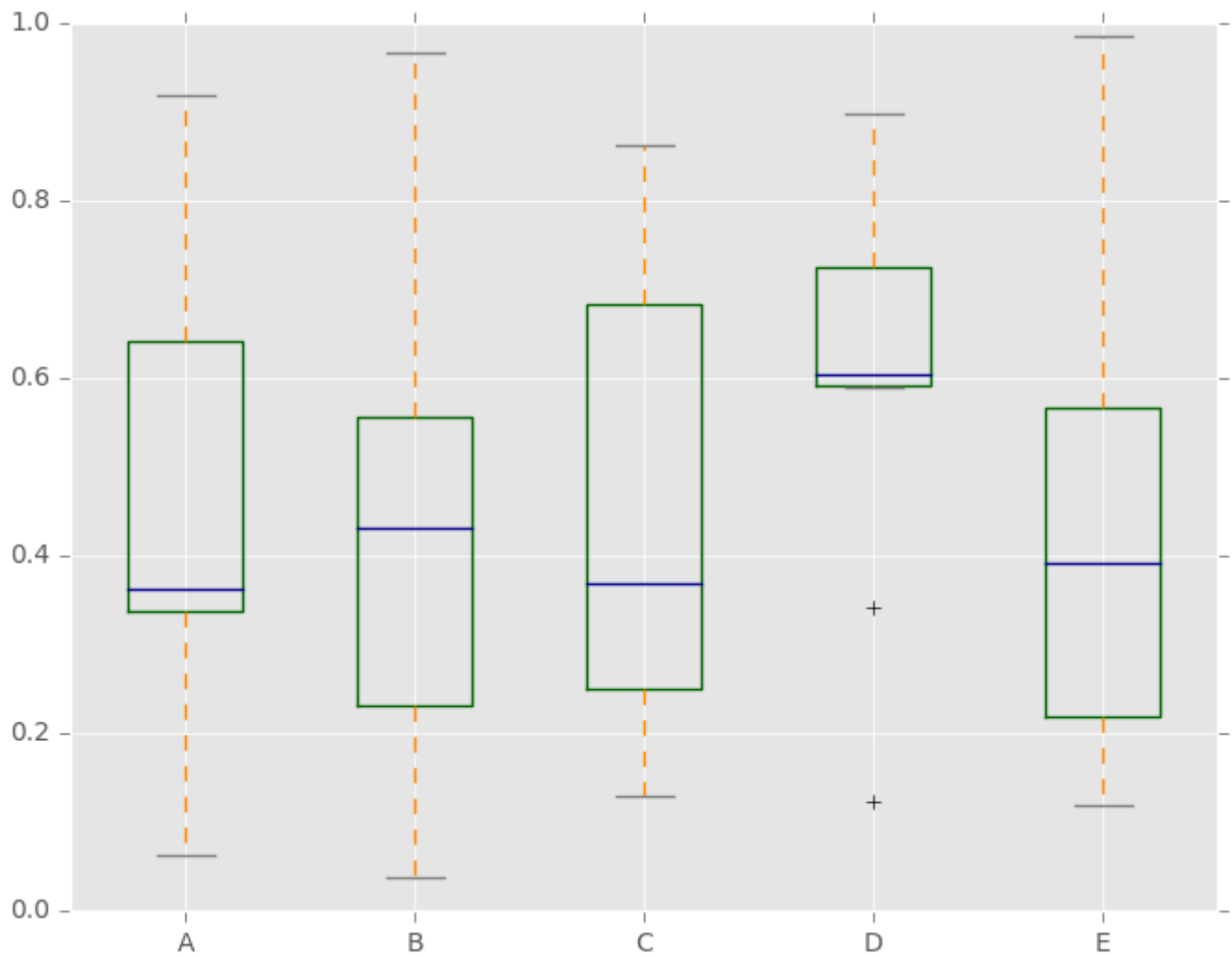
Boxplot can be colored by passing `color` keyword. You can pass a dict whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the dict, default colors are used for the corresponding artists. Also, boxplot has `sym` keyword to specify fliers style.

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`.

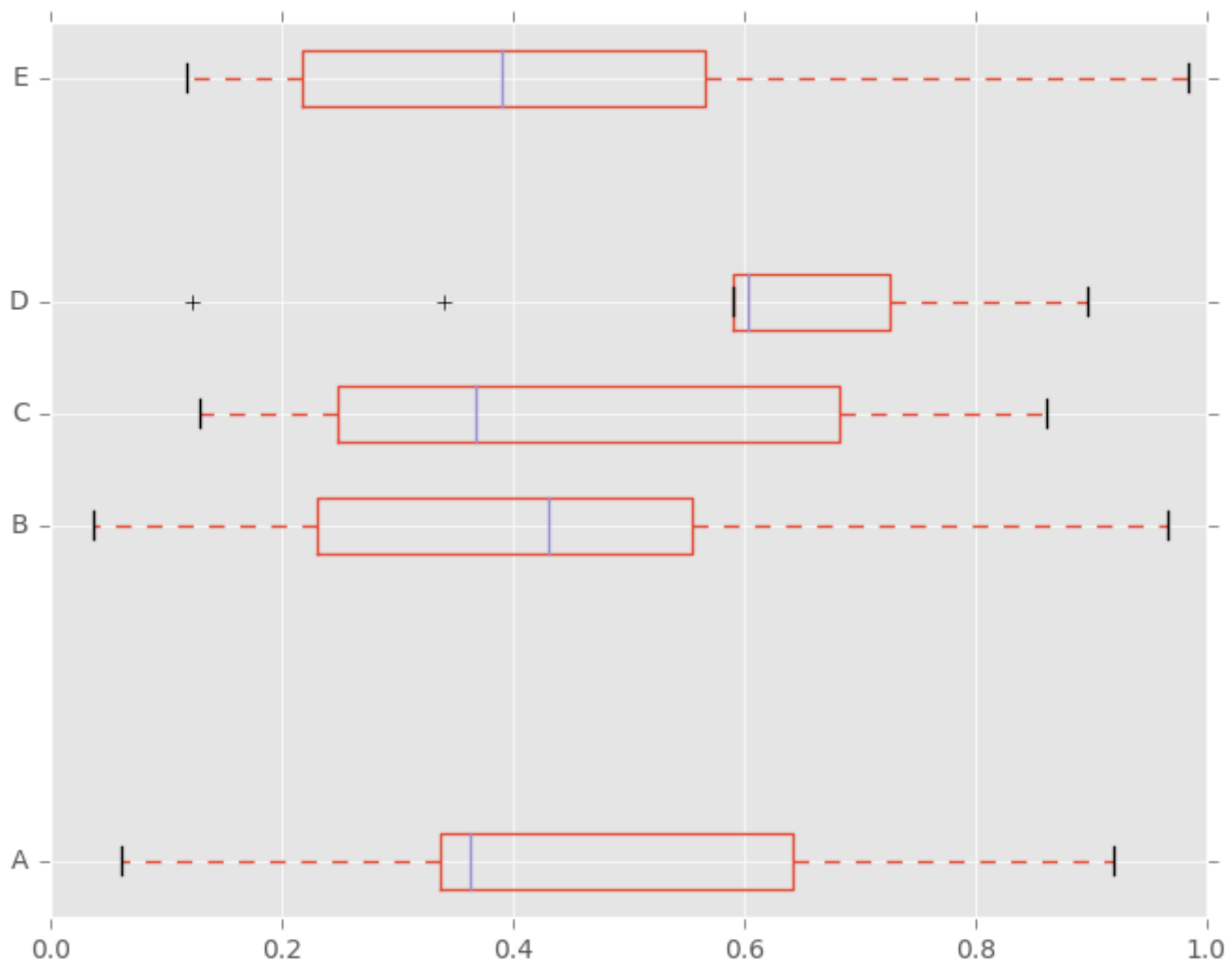
```
In [36]: color = dict(boxes='DarkGreen', whiskers='DarkOrange',
.....:               medians='DarkBlue', caps='Gray')
.....:

In [37]: df.plot.box(color=color, sym='r+')
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbe082c10>
```



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 [38]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8])
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbddc90d0>
```



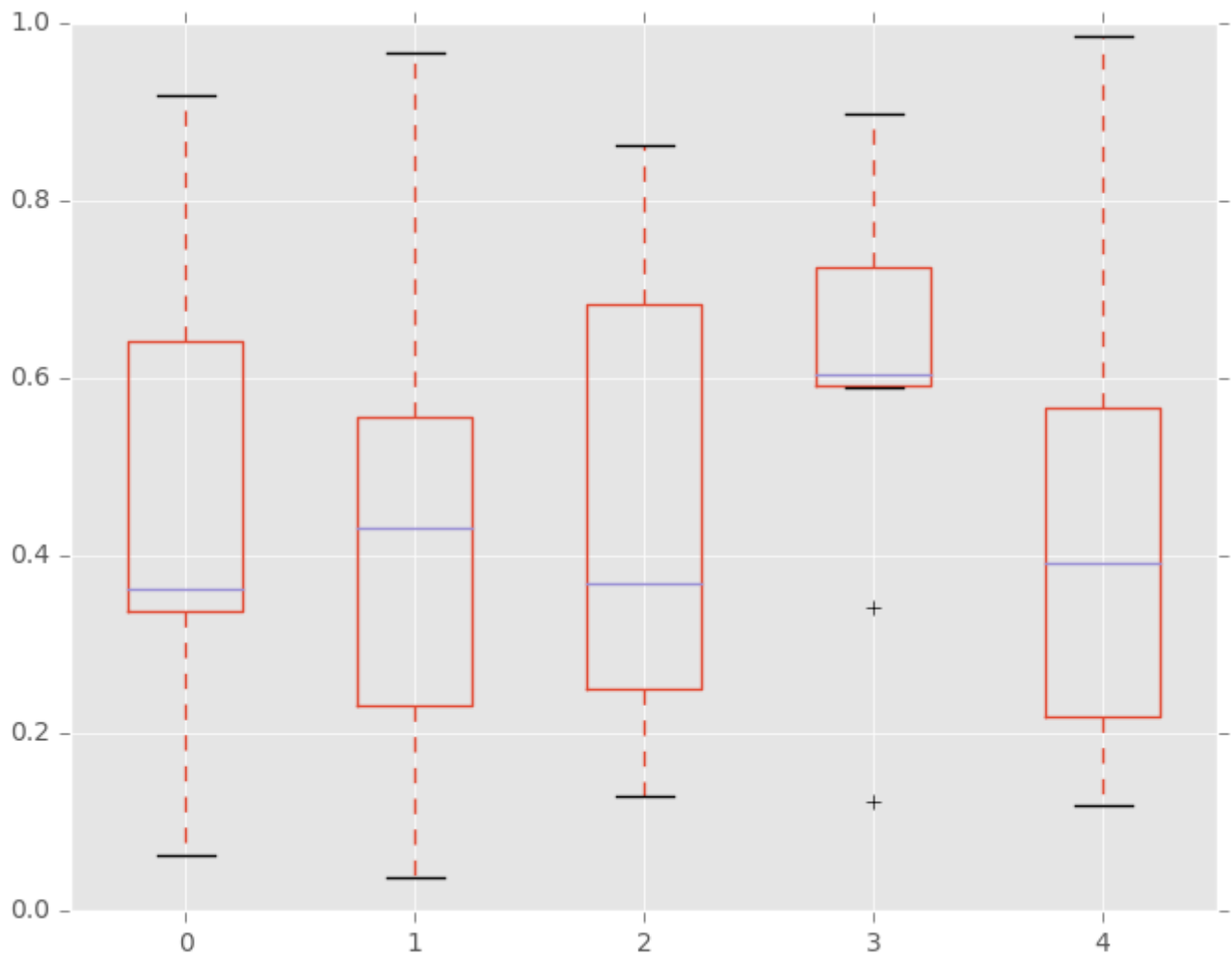
See the [boxplot](#) method and the [matplotlib boxplot documentation](#) for more.

The existing interface `DataFrame.boxplot` to plot boxplot still can be used.

```
In [39]: df = pd.DataFrame(np.random.rand(10,5))
```

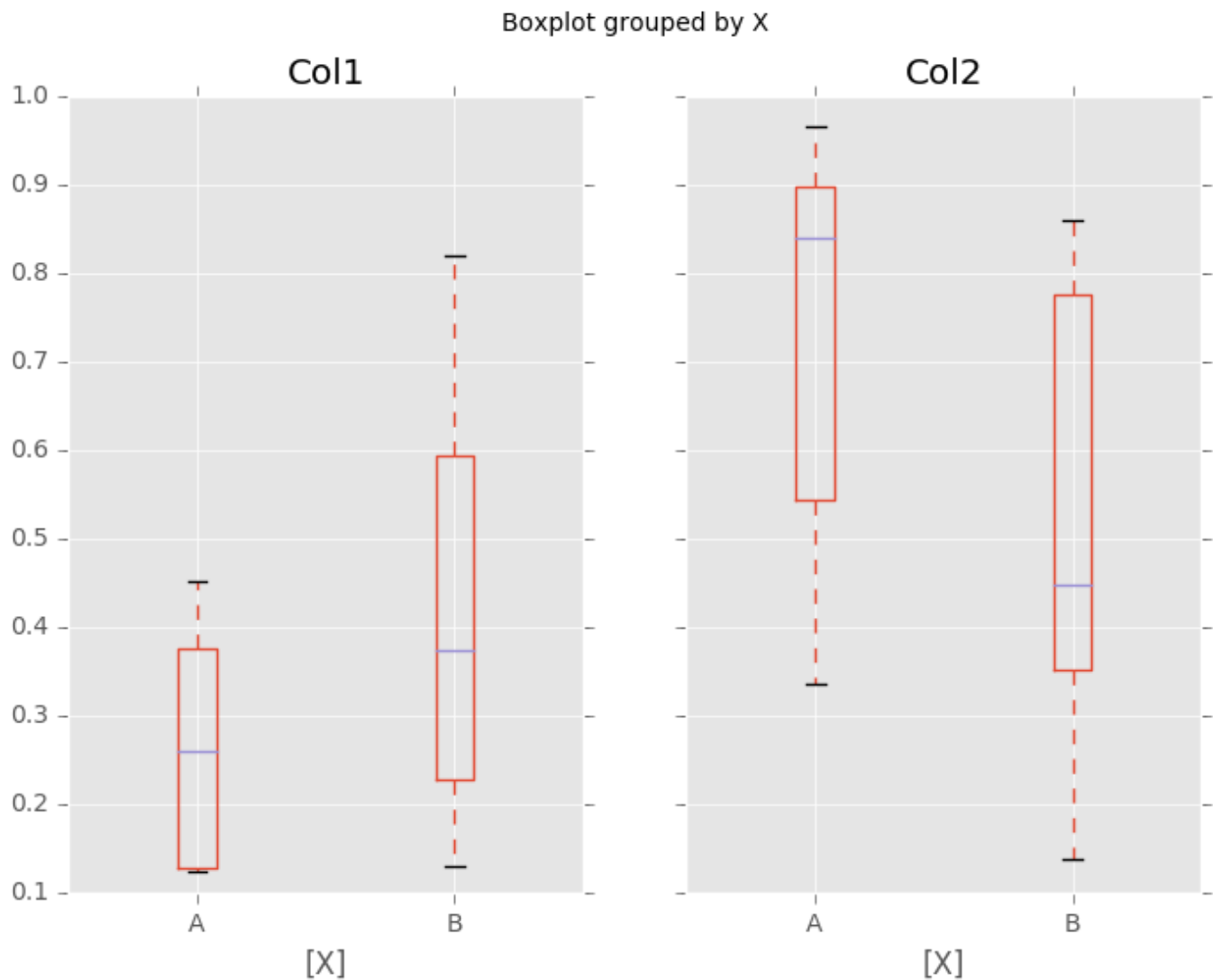
```
In [40]: plt.figure();
```

```
In [41]: bp = df.boxplot()
```



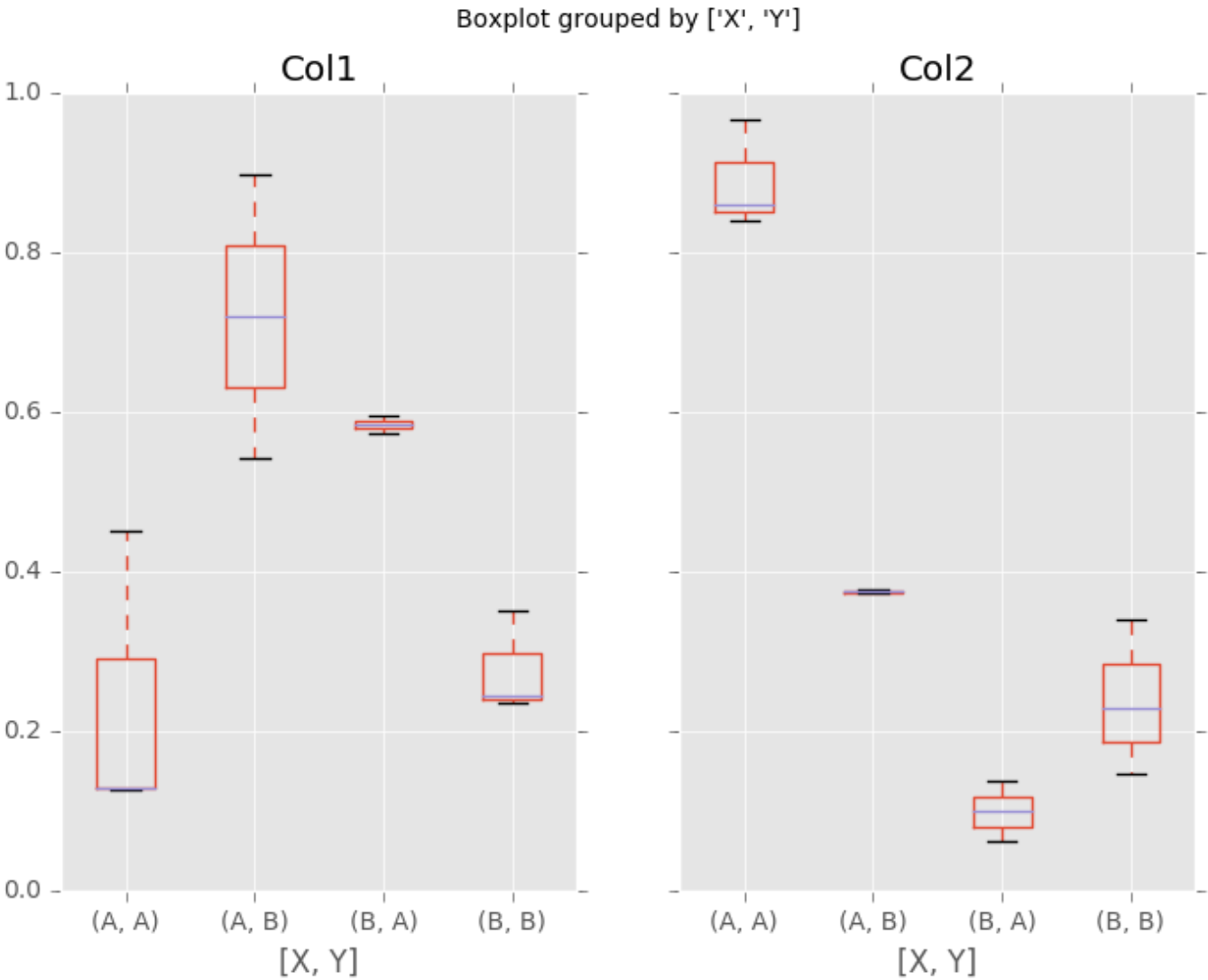
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [42]: df = pd.DataFrame(np.random.rand(10,2), columns=['Col1', 'Col2'] )
In [43]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [44]: plt.figure();
In [45]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [46]: df = pd.DataFrame(np.random.rand(10,3), columns=['Col1', 'Col2', 'Col3'])
In [47]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [48]: df['Y'] = pd.Series(['A','B','A','B','A','B','A','B','A','B'])
In [49]: plt.figure();
In [50]: bp = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```



Warning: The default changed from 'dict' to 'axes' in version 0.19.0.

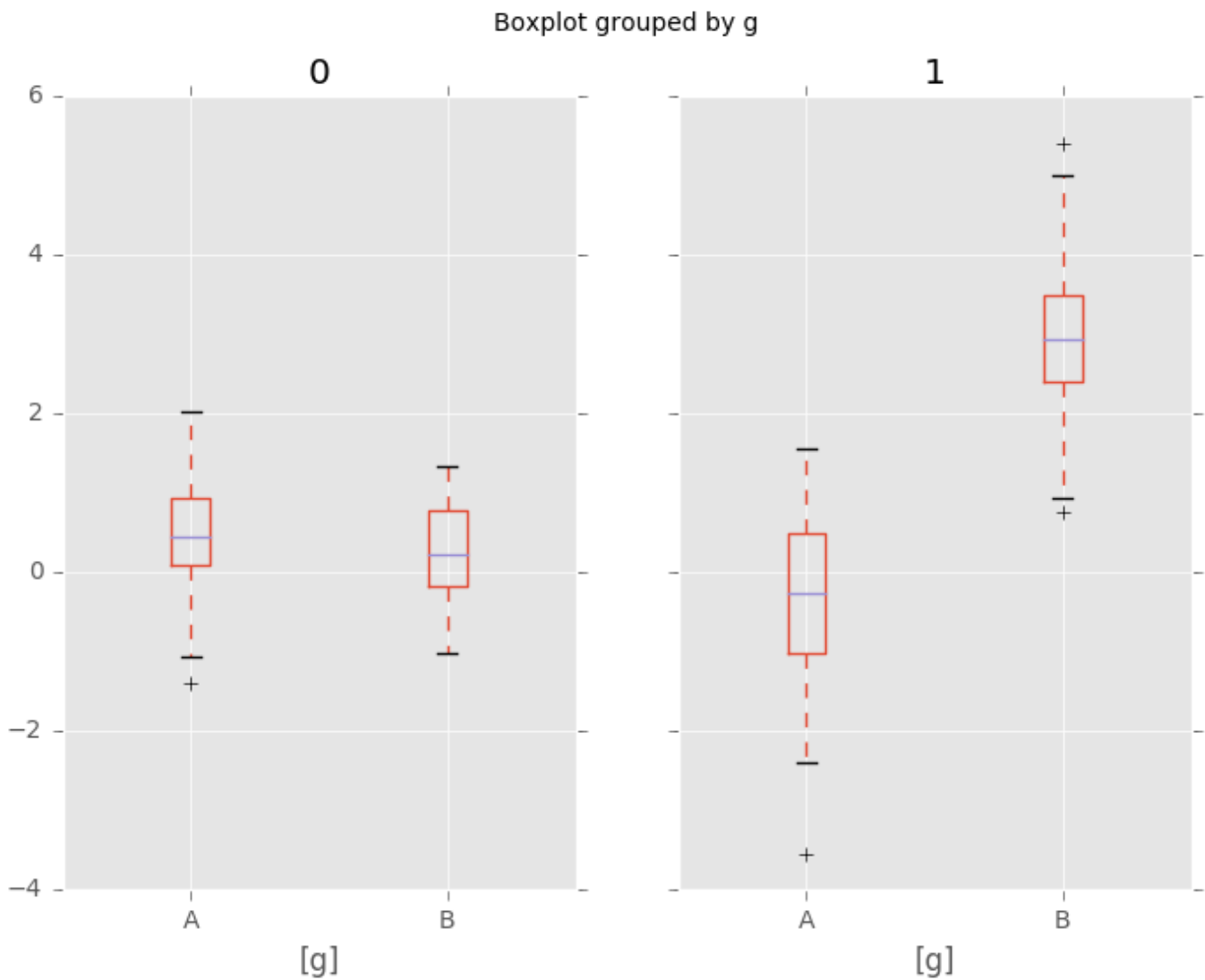
In `boxplot`, the return type can be controlled by the `return_type`, keyword. The valid choices are {"axes", "dict", "both", None}. Faceting, created by `DataFrame.boxplot` with the `by` keyword, will affect the output type as well:

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`.

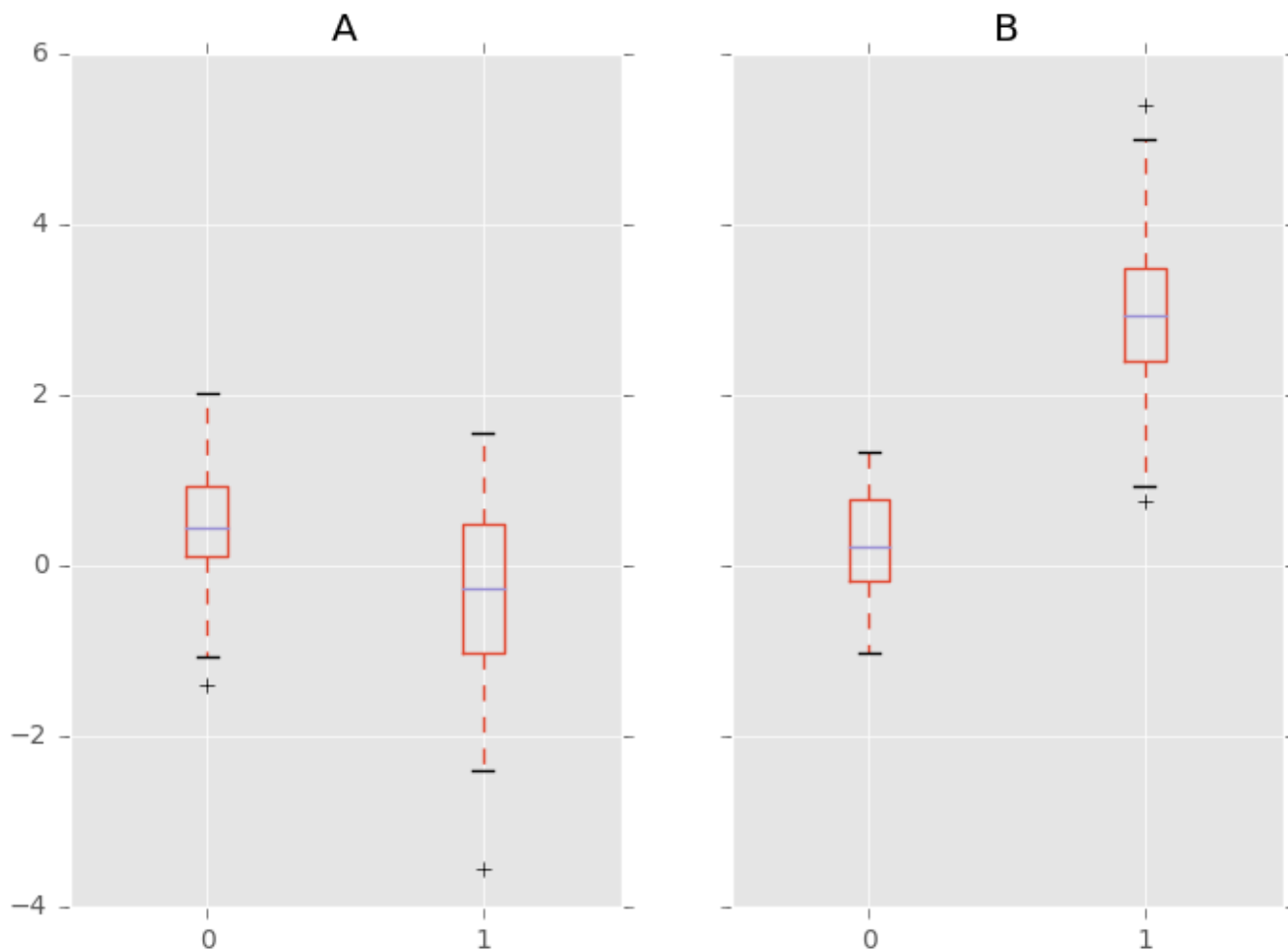
In [51]: `np.random.seed(1234)`

```
In [52]: df_box = pd.DataFrame(np.random.randn(50, 2))  
In [53]: df_box['g'] = np.random.choice(['A', 'B'], size=50)  
In [54]: df_box.loc[df_box['g'] == 'B', 1] += 3  
In [55]: bp = df_box.boxplot(by='g')
```



Compare to:

```
In [56]: bp = df_box.groupby('g').boxplot()
```



Area Plot

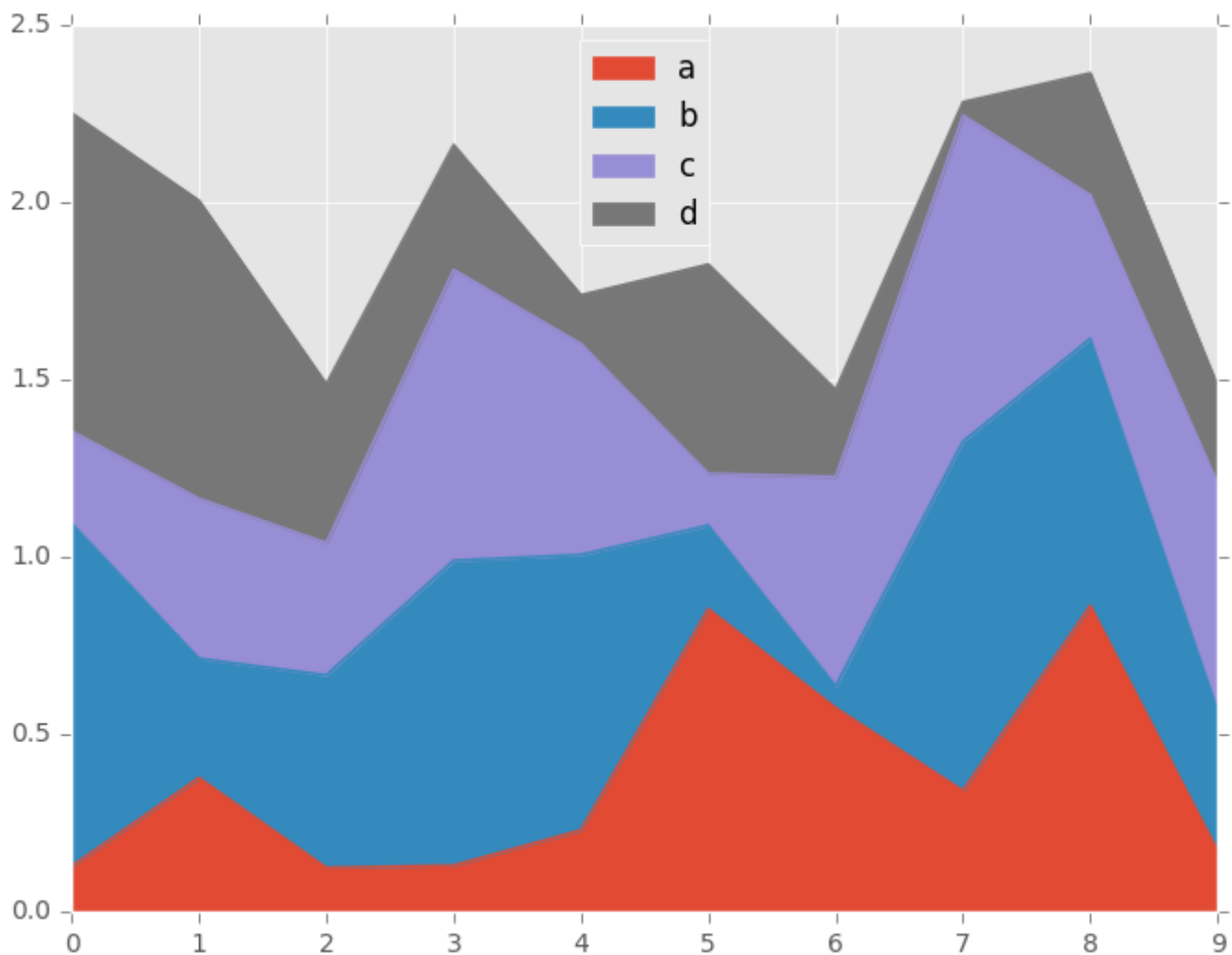
New in version 0.14.

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`.

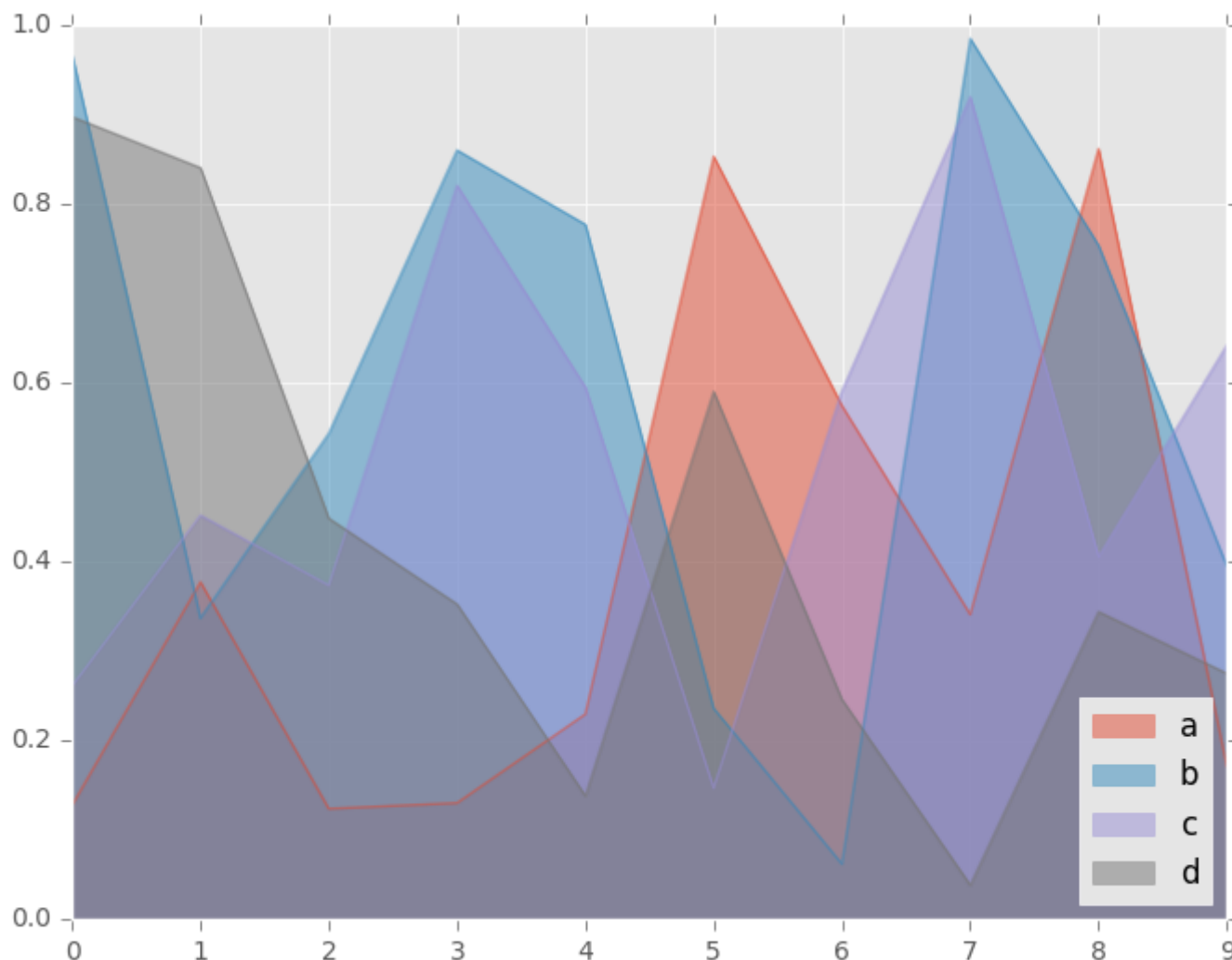
```
In [57]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [58]: df.plot.area();
```

To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

```
In [59]: df.plot.area(stacked=False);
```



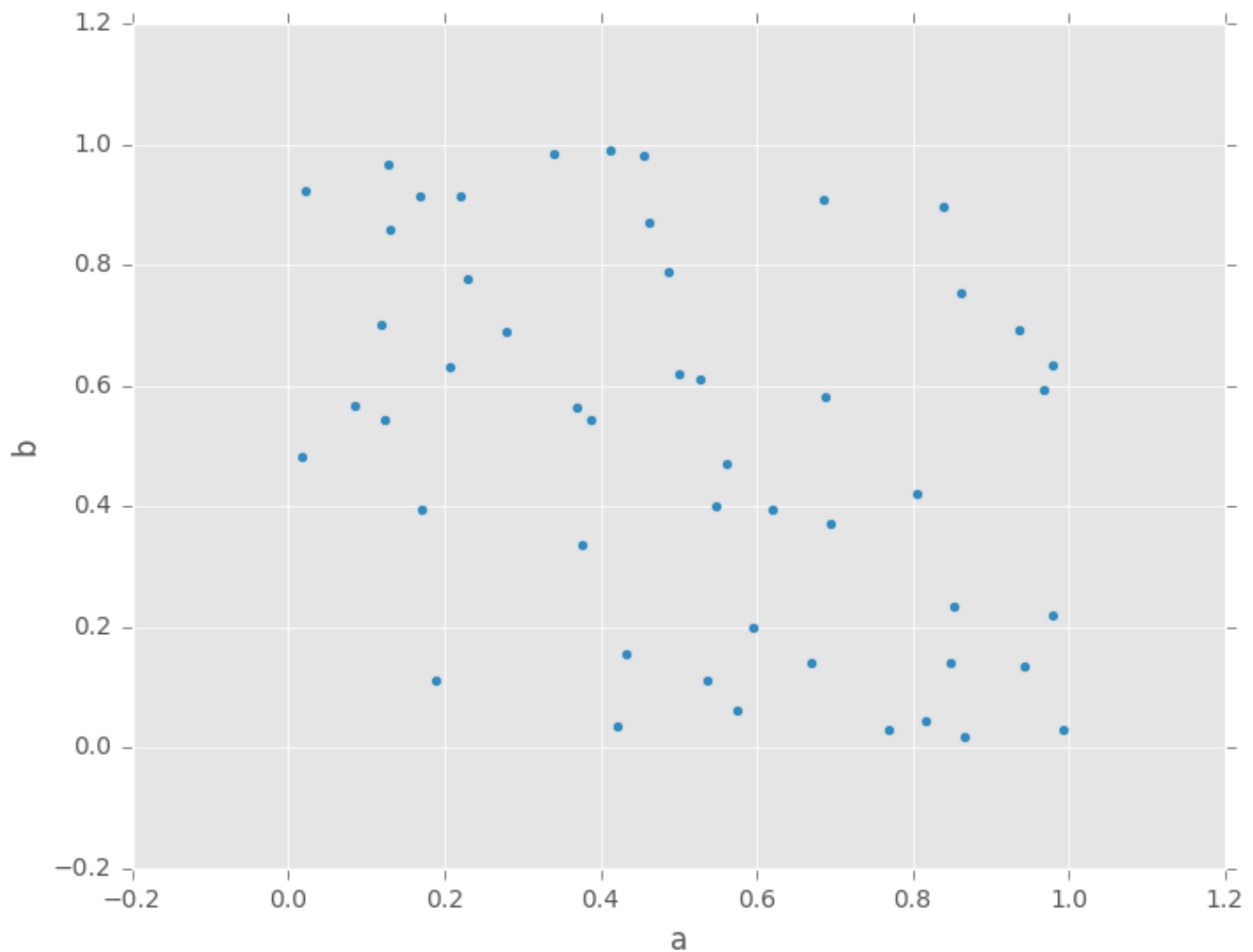
Scatter Plot

New in version 0.13.

Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for x and y axis. These can be specified by `x` and `y` keywords each.

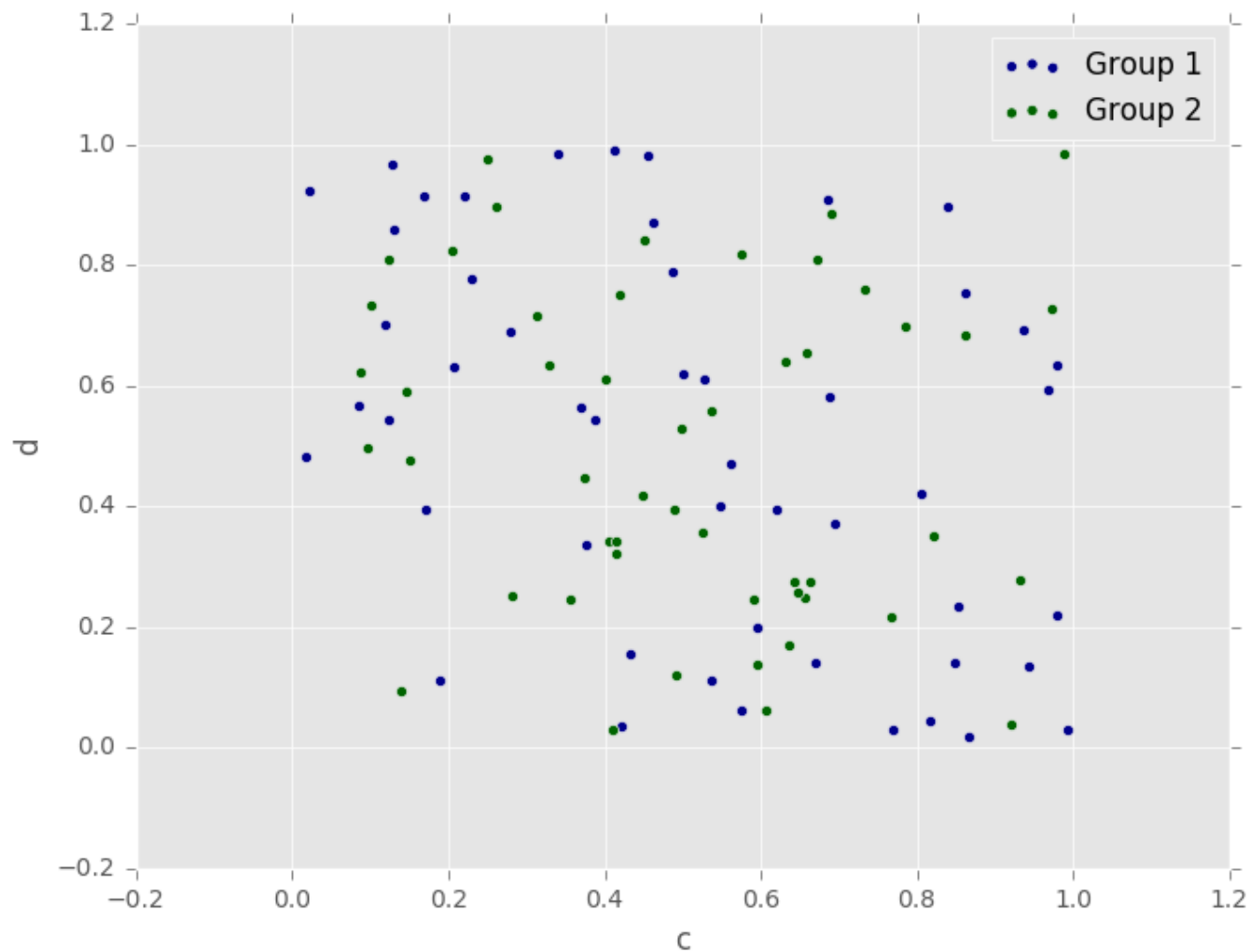
```
In [60]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [61]: df.plot.scatter(x='a', y='b');
```



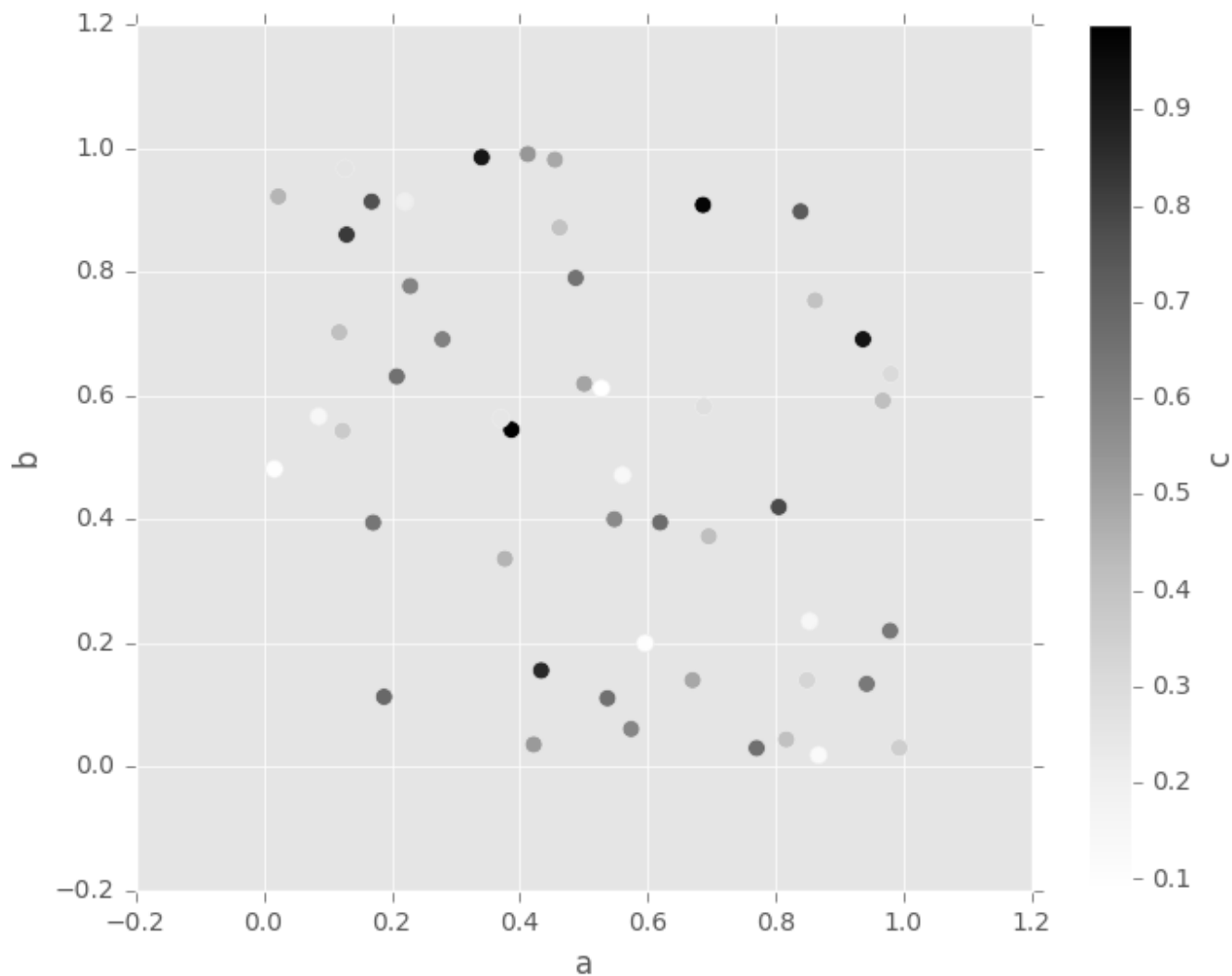
To plot multiple column groups in a single axes, repeat `plot` method specifying target `ax`. It is recommended to specify `color` and `label` keywords to distinguish each groups.

```
In [62]: ax = df.plot.scatter(x='a', y='b', color='DarkBlue', label='Group 1');  
In [63]: df.plot.scatter(x='c', y='d', color='DarkGreen', label='Group 2', ax=ax);
```



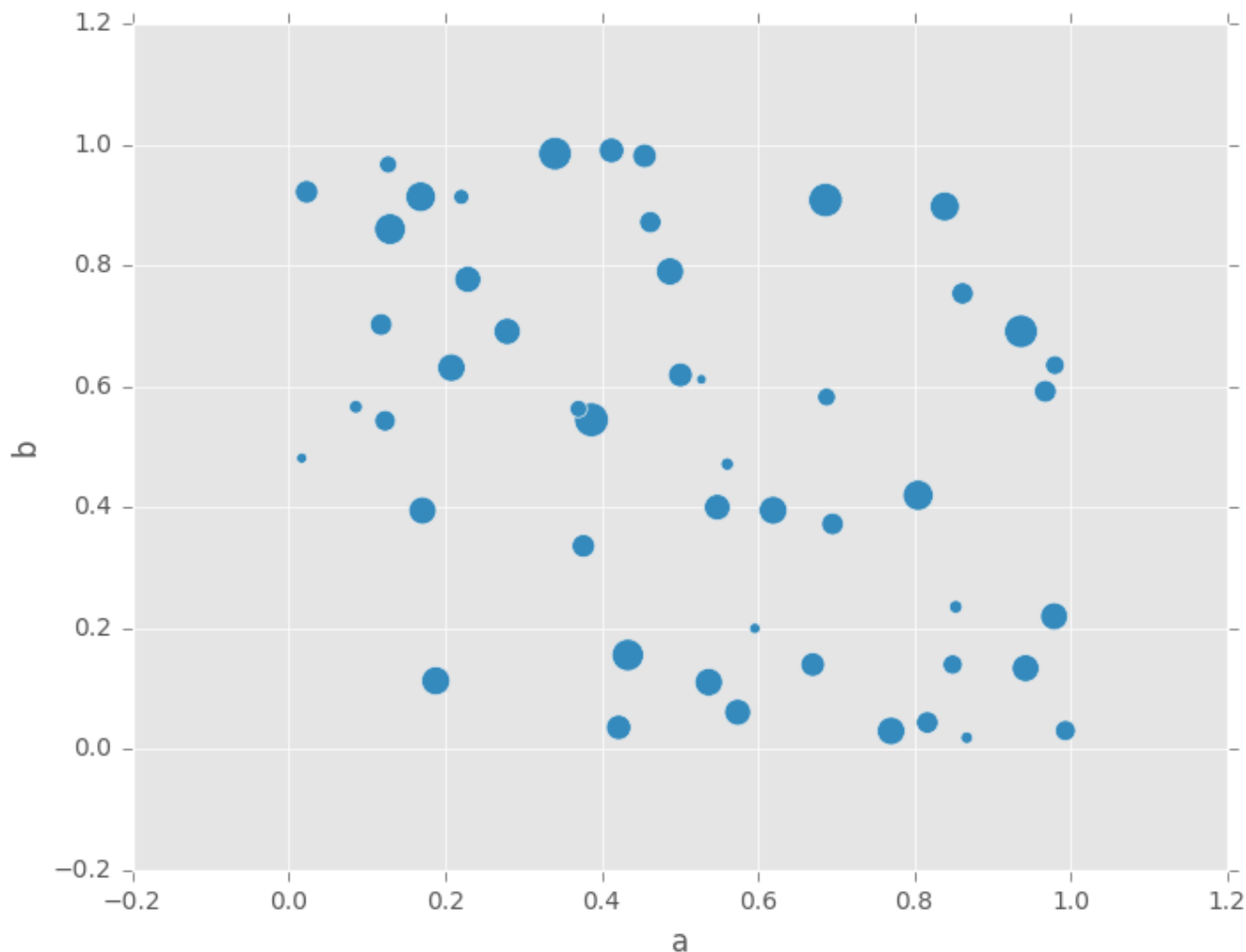
The keyword `c` may be given as the name of a column to provide colors for each point:

```
In [64]: df.plot.scatter(x='a', y='b', c='c', s=50);
```



You can pass other keywords supported by matplotlib `scatter`. Below example shows a bubble chart using a dataframe column values as bubble size.

```
In [65]: 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

New in version 0.14.

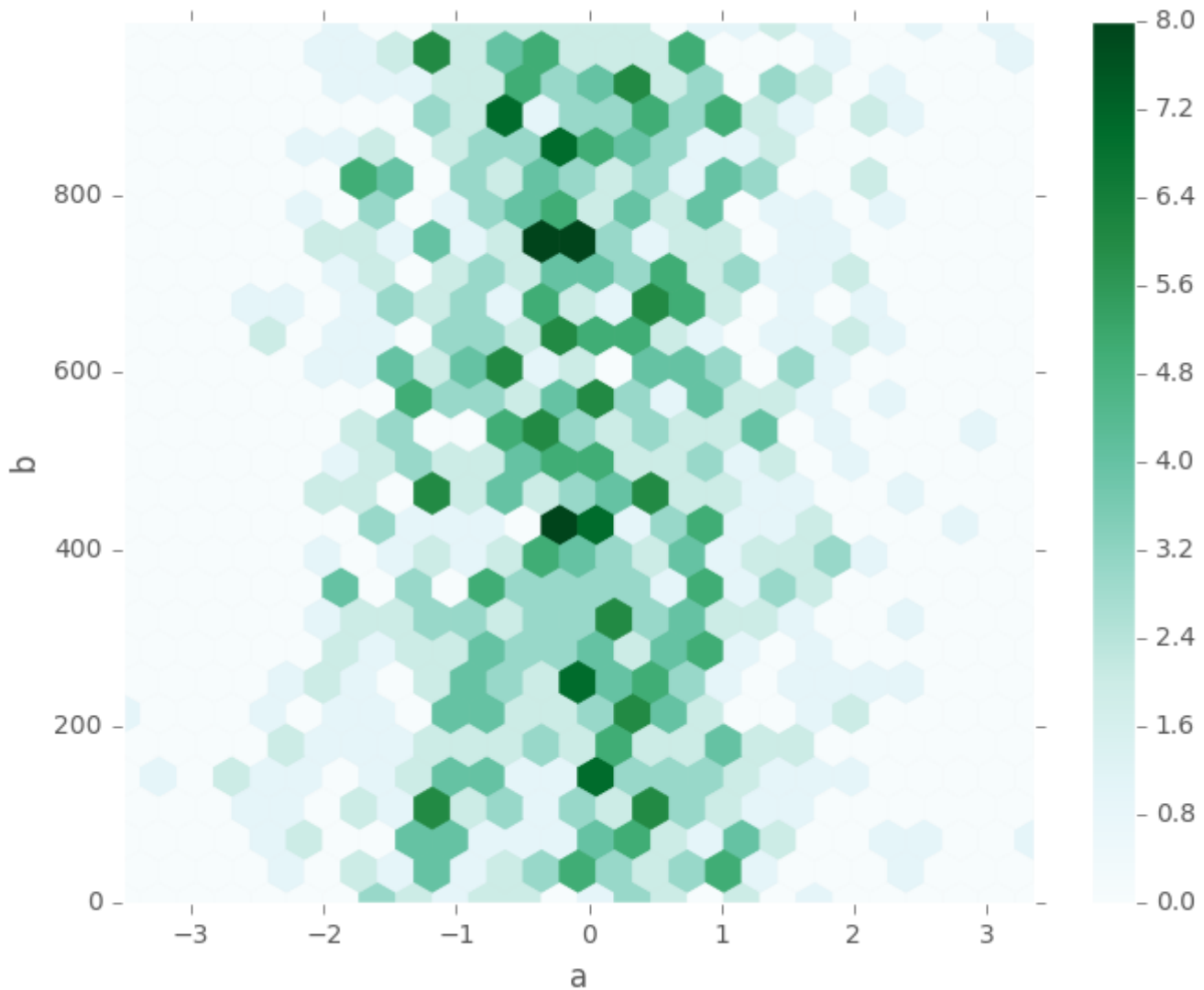
You can create hexagonal bin plots with `DataFrame.plot.hexbin()`. Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

```
In [66]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
```

```
In [67]: df['b'] = df['b'] + np.arange(1000)
```

```
In [68]: df.plot.hexbin(x='a', y='b', gridsize=25)
```

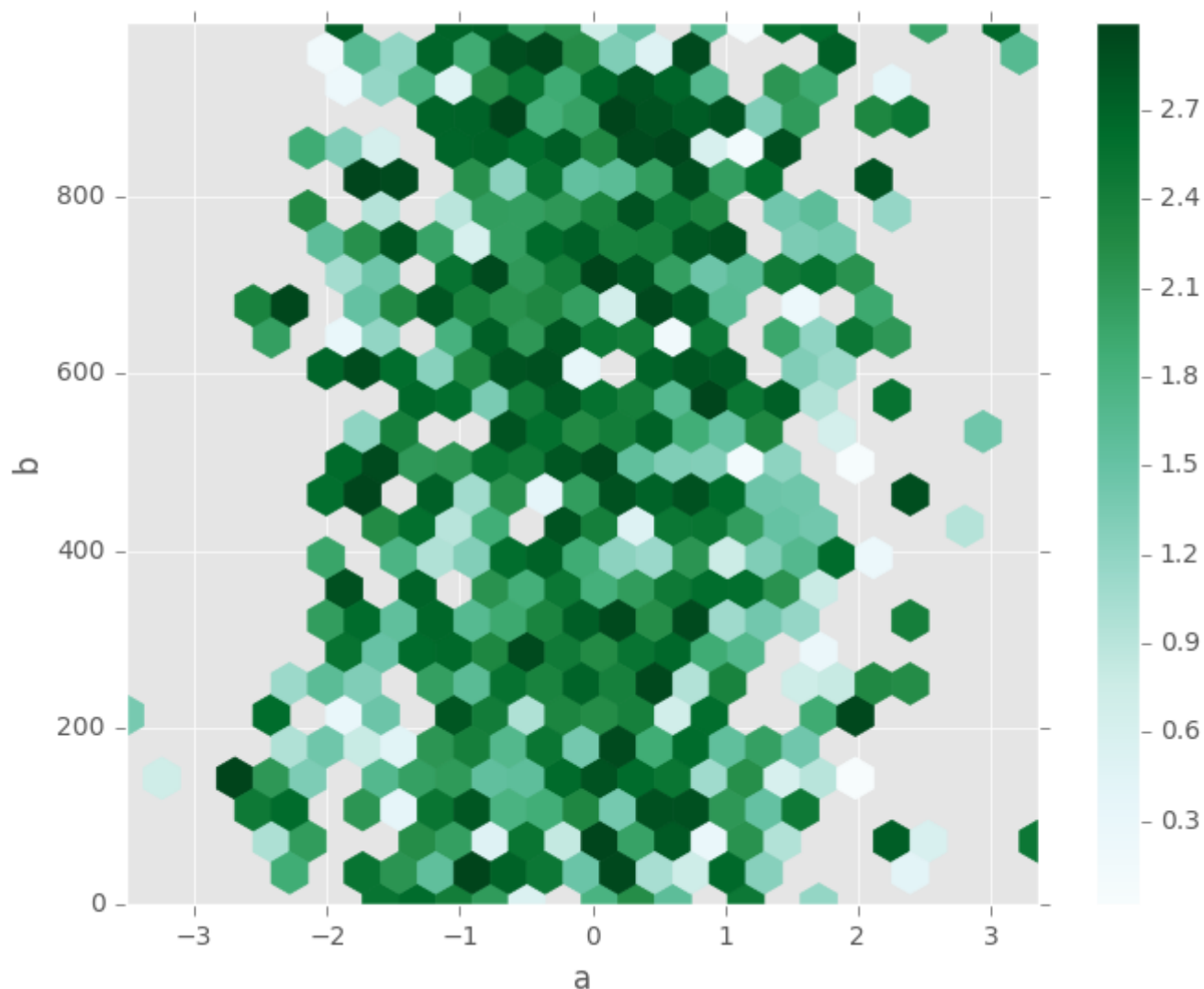
```
Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbc33cb10>
```



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 numpy's `max` function.

```
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['z'] = np.random.uniform(0, 3, 1000)
In [72]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max,
.....:                 gridsize=25)
.....:
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbed61850>
```



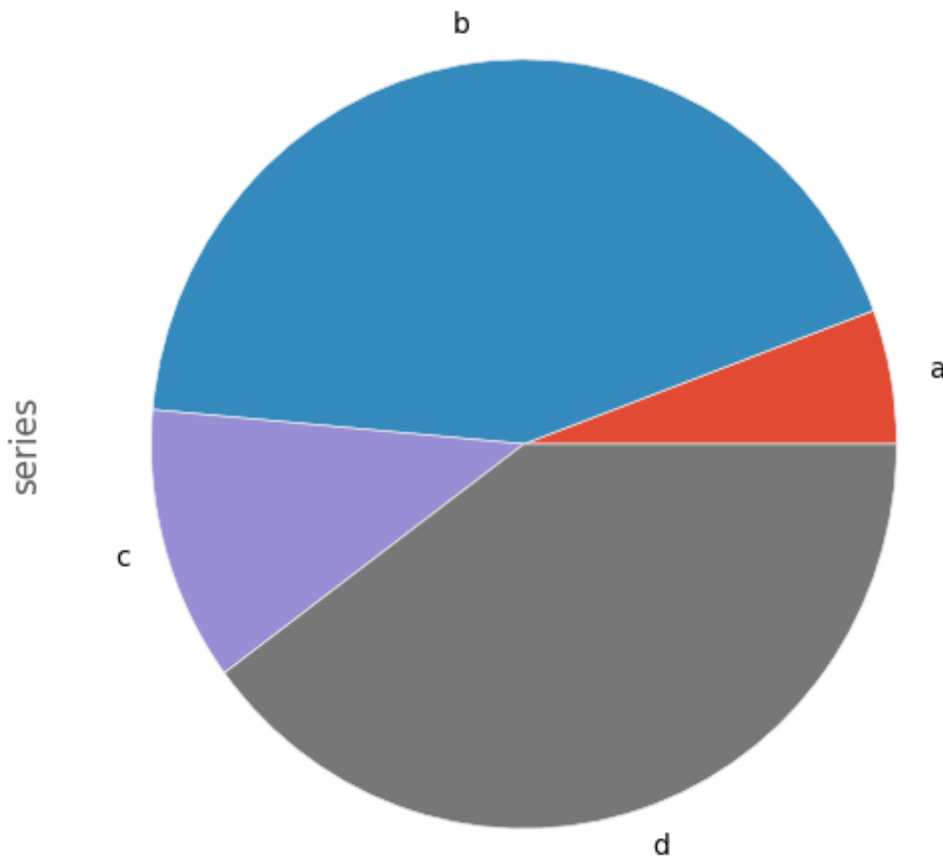
See the [hexbin](#) method and the [matplotlib hexbin documentation](#) for more.

Pie plot

New in version 0.14.

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.

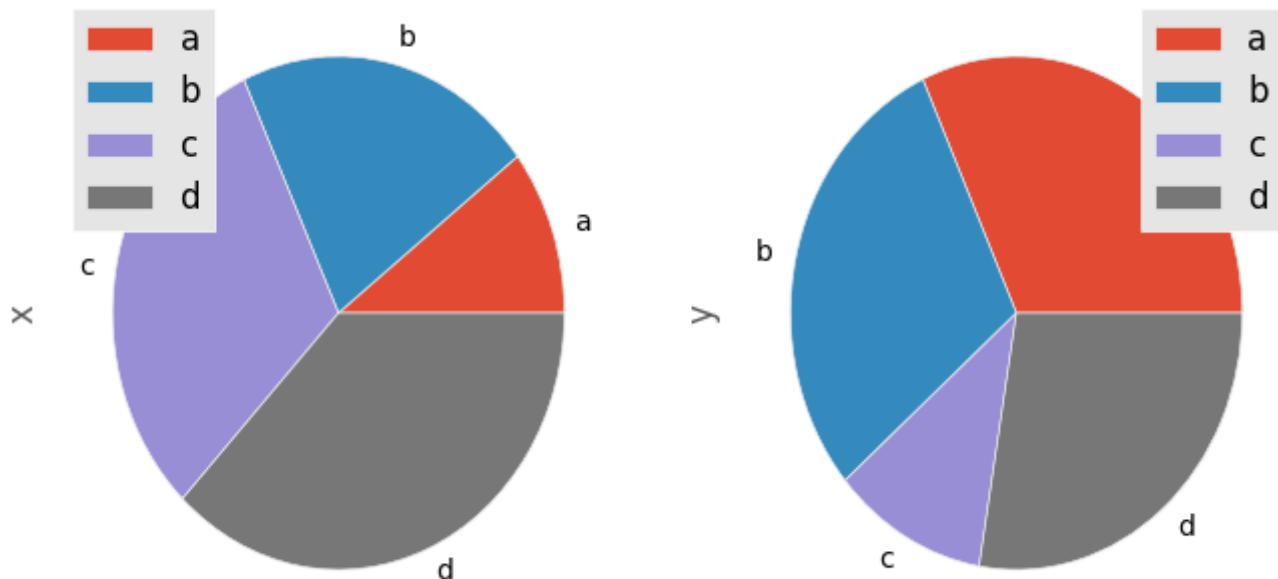
```
In [73]: series = pd.Series(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], name='series')
In [74]: series.plot.pie(figsize=(6, 6))
Out[74]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbe9a5190>
```

For pie plots it's best to use square figures, one's with an equal aspect ratio. 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 [75]: df = pd.DataFrame(3 * np.random.rand(4, 2), index=['a', 'b', 'c', 'd'], columns=['x',  
In [76]: df.plot.pie(subplots=True, figsize=(8, 4))  
Out[76]:  
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f3da974fbd0>,  
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dba8f9910>], dtype=object)
```

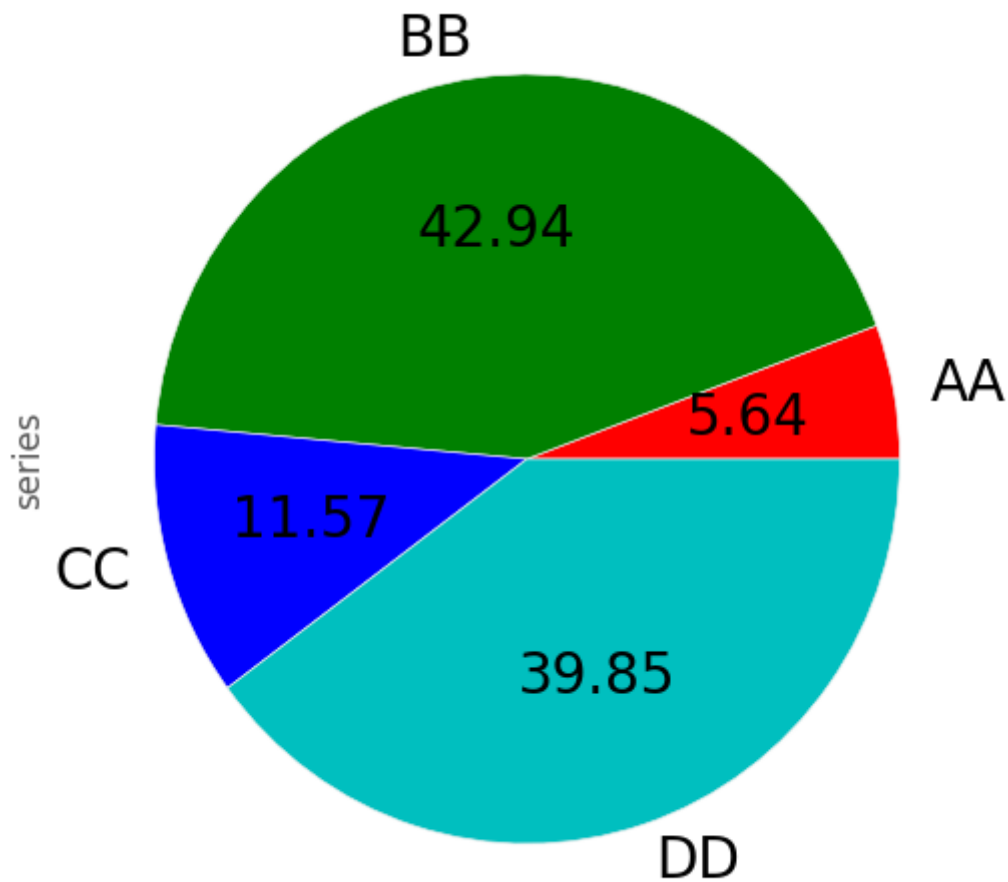


You can use the `labels` and `colors` keywords to specify the labels and colors of each wedge.

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`.

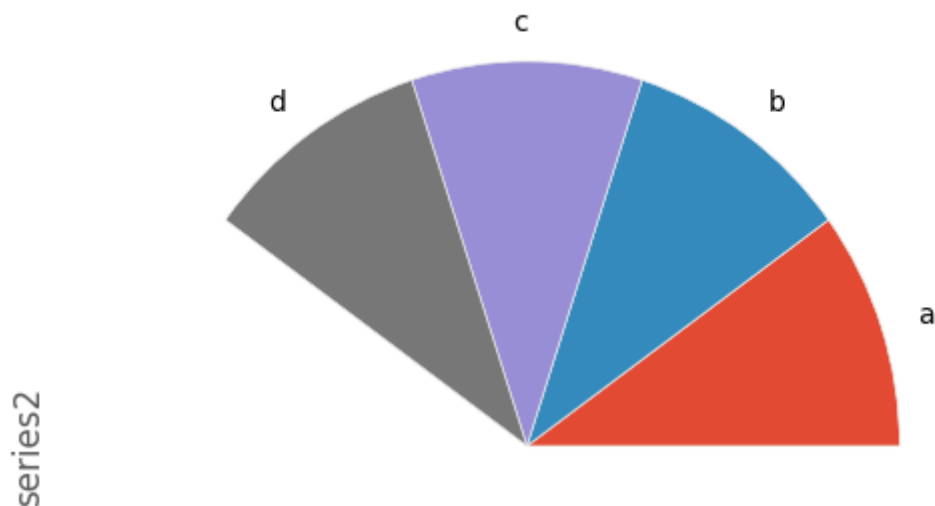
If you want to hide wedge labels, specify `labels=None`. If `fontsize` is specified, the value will be applied to wedge labels. Also, other keywords supported by `matplotlib.pyplot.pie()` can be used.

```
In [77]: series.plot.pie(labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
.....:                  autopct='%.2f', fontsize=20, figsize=(6, 6))
.....:
Out[77]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbd742e10>
```



If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```
In [78]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')  
In [79]: series.plot.pie(figsize=(6, 6))  
Out[79]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db840e550>
```



See the [matplotlib pie documentation](#) for more.

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 0's
Bar	Fill 0's
Scatter	Drop NaNs
Histogram	Drop NaNs (column-wise)
Box	Drop NaNs (column-wise)
Area	Fill 0's
KDE	Drop NaNs (column-wise)
Hexbin	Drop NaNs
Pie	Fill 0's

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.

Plotting Tools

These functions can be imported from `pandas.tools.plotting` and take a [Series](#) or [DataFrame](#) as an argument.

Scatter Matrix Plot

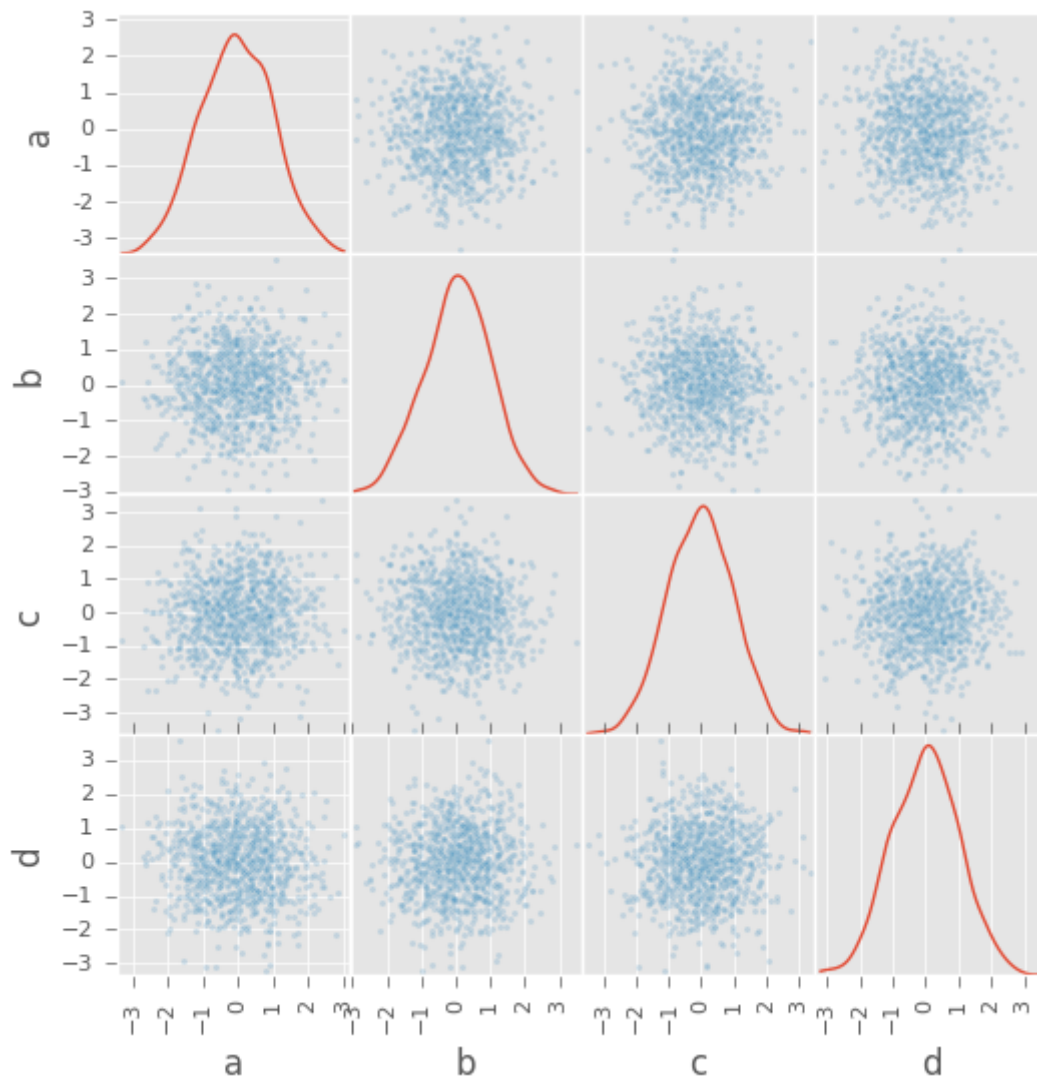
New in version 0.7.3.

You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.tools.plotting`:

```
In [80]: from pandas.tools.plotting import scatter_matrix

In [81]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [82]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
Out[82]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f3db8633a50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3da9be1fd0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3daa6ad650>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dba570f90>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x7f3db9753450>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dba9f4850>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db833f8d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db979b690>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbf65c710>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db954b8d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3da9ac2510>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbadf0810>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x7f3db3f91950>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3dbe8408d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3db83616d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f3da9f388d0>]], dtype=object)
```

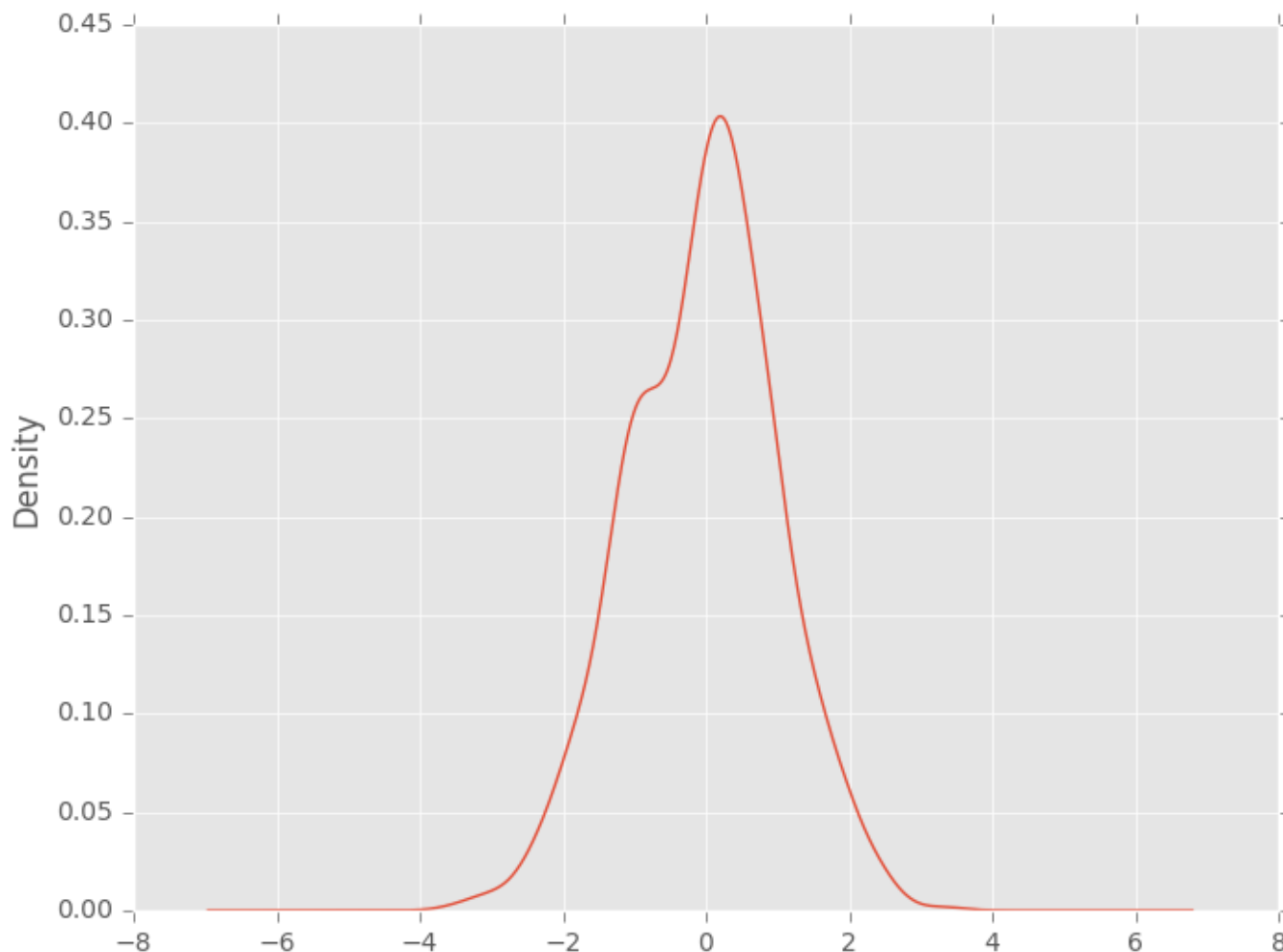


Density Plot

New in version 0.8.0.

You can create density plots using the `Series.plot.kde()` and `DataFrame.plot.kde()` methods.

```
In [83]: ser = pd.Series(np.random.randn(1000))  
In [84]: ser.plot.kde()  
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbf9623d0>
```

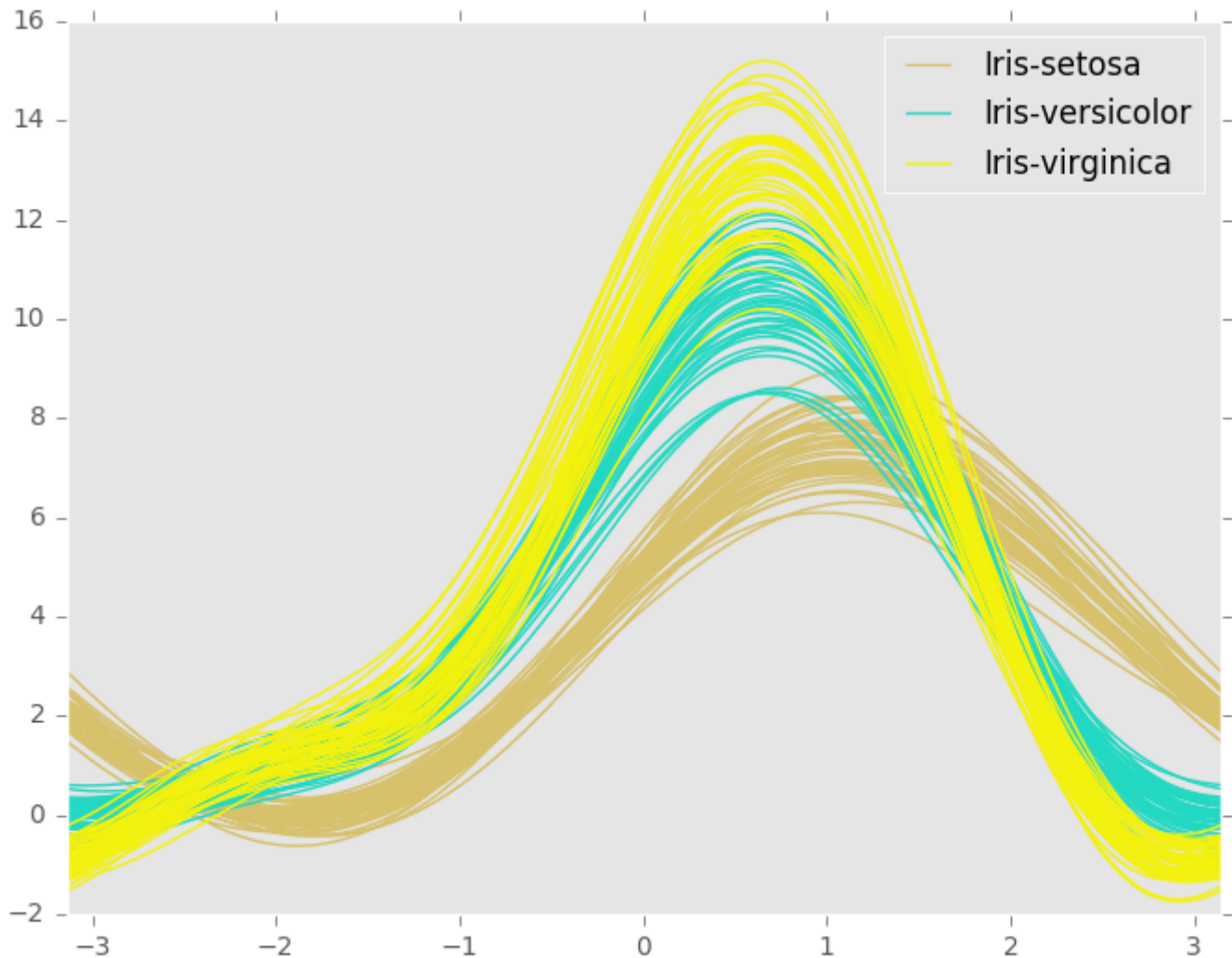


Andrews Curves

Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

Note: The “Iris” dataset is available [here](#).

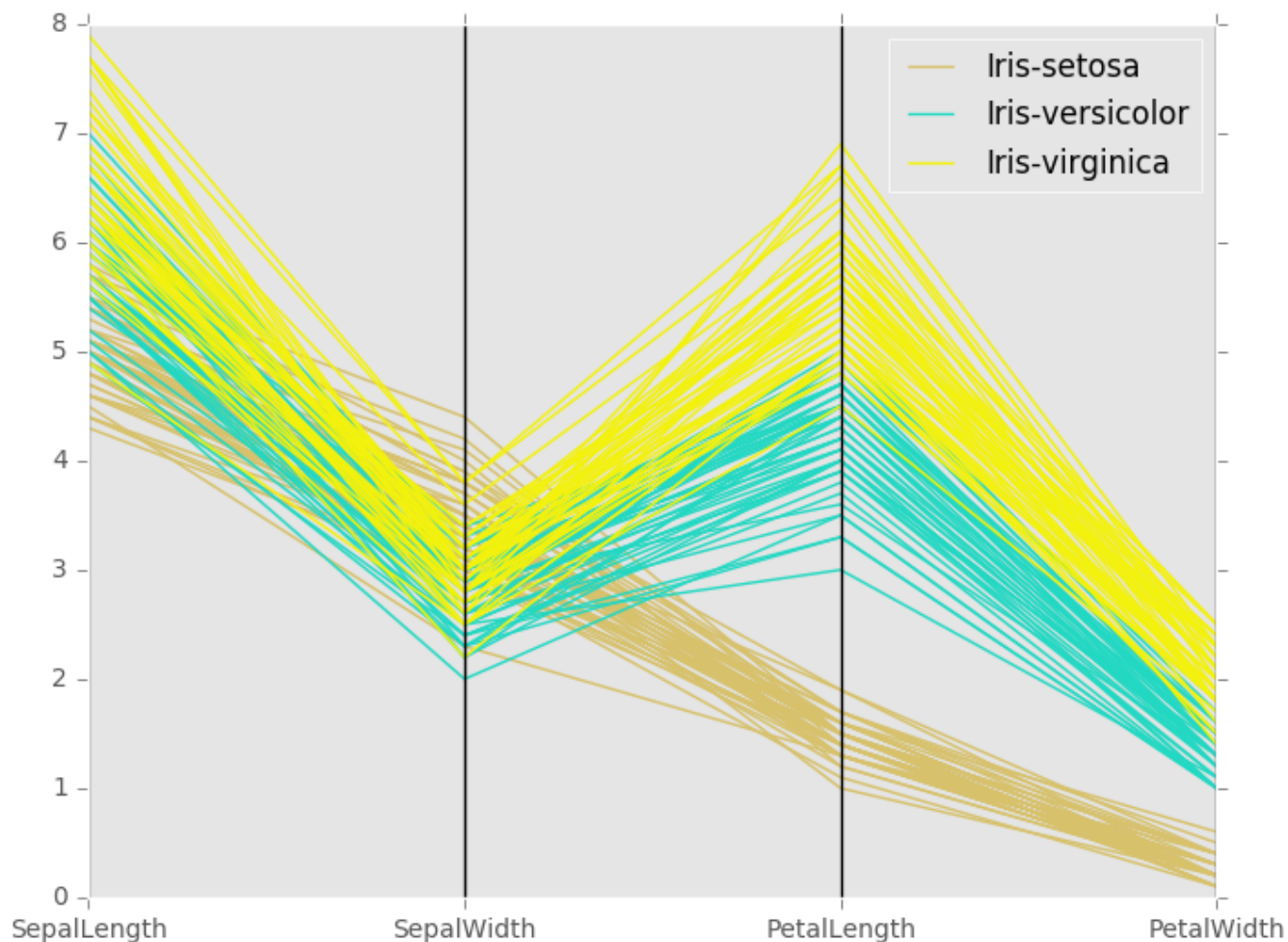
```
In [85]: from pandas.tools.plotting import andrews_curves
In [86]: data = pd.read_csv('data/iris.data')
In [87]: plt.figure()
Out[87]: <matplotlib.figure.Figure at 0x7f3dbc0e6d50>
In [88]: andrews_curves(data, 'Name')
Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbbf40e50>
```



Parallel Coordinates

Parallel coordinates is a plotting technique for plotting multivariate data. It allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

```
In [89]: from pandas.tools.plotting import parallel_coordinates
In [90]: data = pd.read_csv('data/iris.data')
In [91]: plt.figure()
Out[91]: <matplotlib.figure.Figure at 0x7f3db8d16710>
In [92]: parallel_coordinates(data, 'Name')
Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8d1c710>
```

Lag Plot

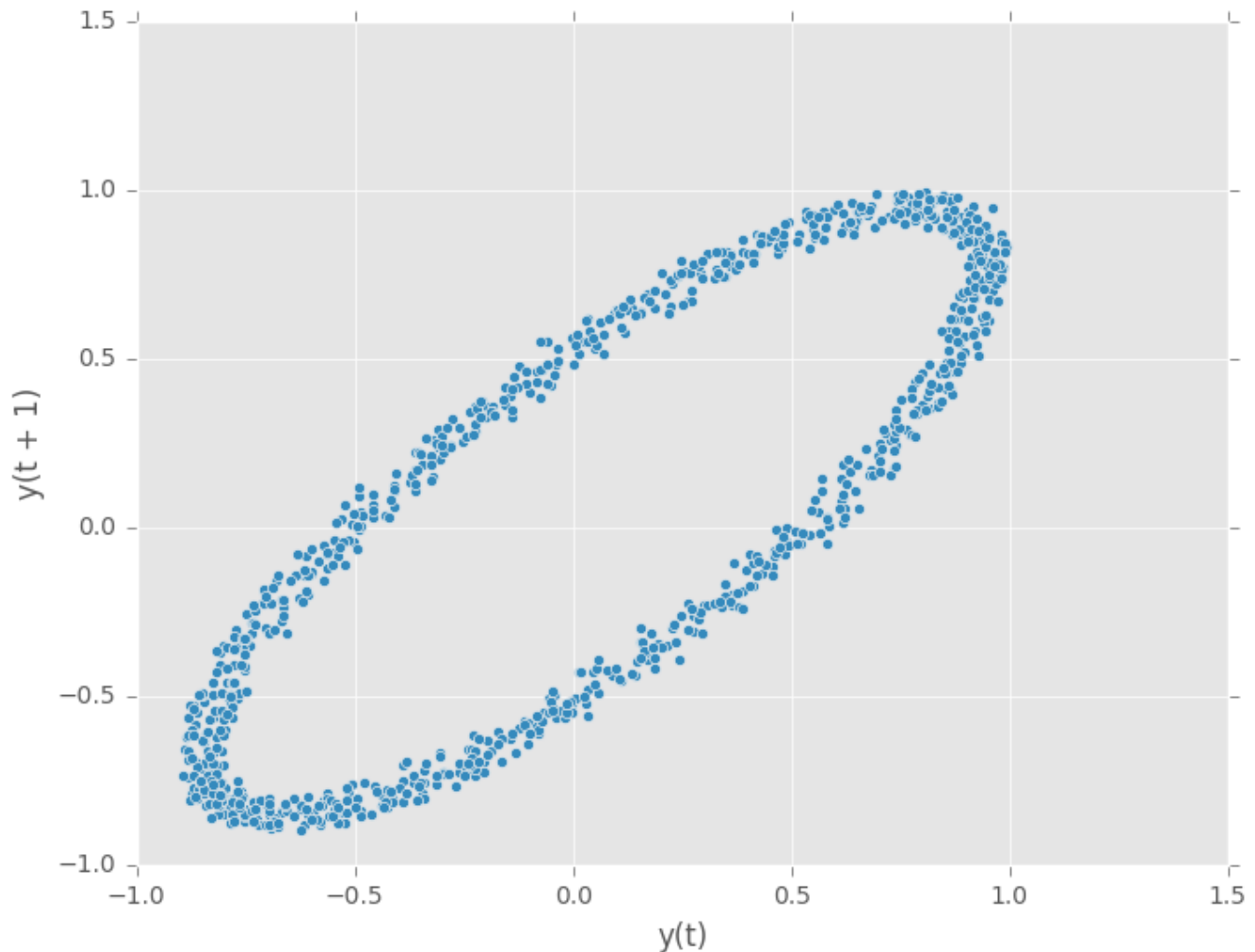
Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random.

```
In [93]: from pandas.tools.plotting import lag_plot

In [94]: plt.figure()
Out[94]: <matplotlib.figure.Figure at 0x7f3db9bc9e10>

In [95]: data = pd.Series(0.1 * np.random.rand(1000) +
.....:     0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
.....:

In [96]: lag_plot(data)
Out[96]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db9bc9590>
```



Autocorrelation Plot

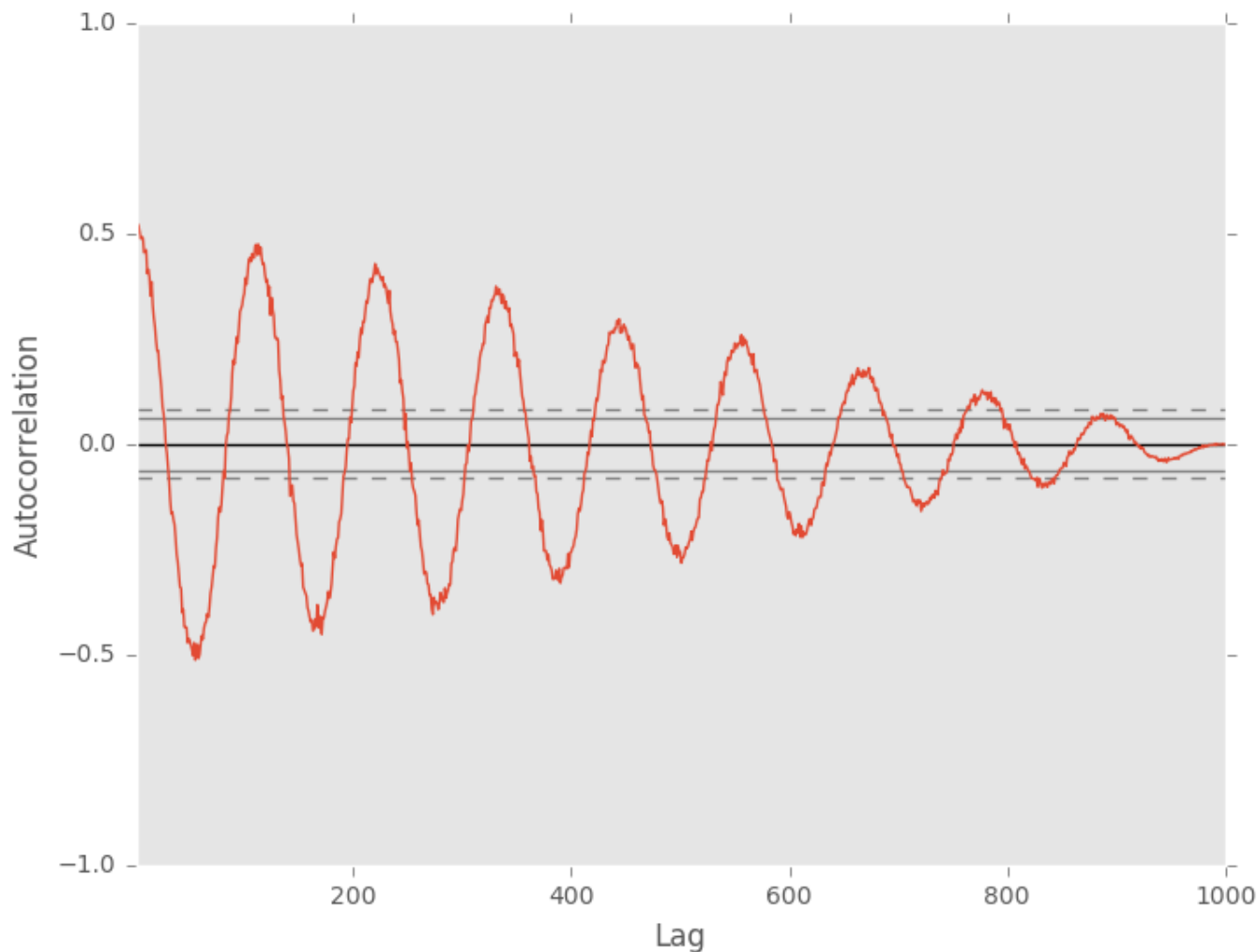
Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band.

```
In [97]: from pandas.tools.plotting import autocorrelation_plot

In [98]: plt.figure()
Out[98]: <matplotlib.figure.Figure at 0x7f3dbdab9590>

In [99]: data = pd.Series(0.7 * np.random.rand(1000) +
.....:     0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
.....:

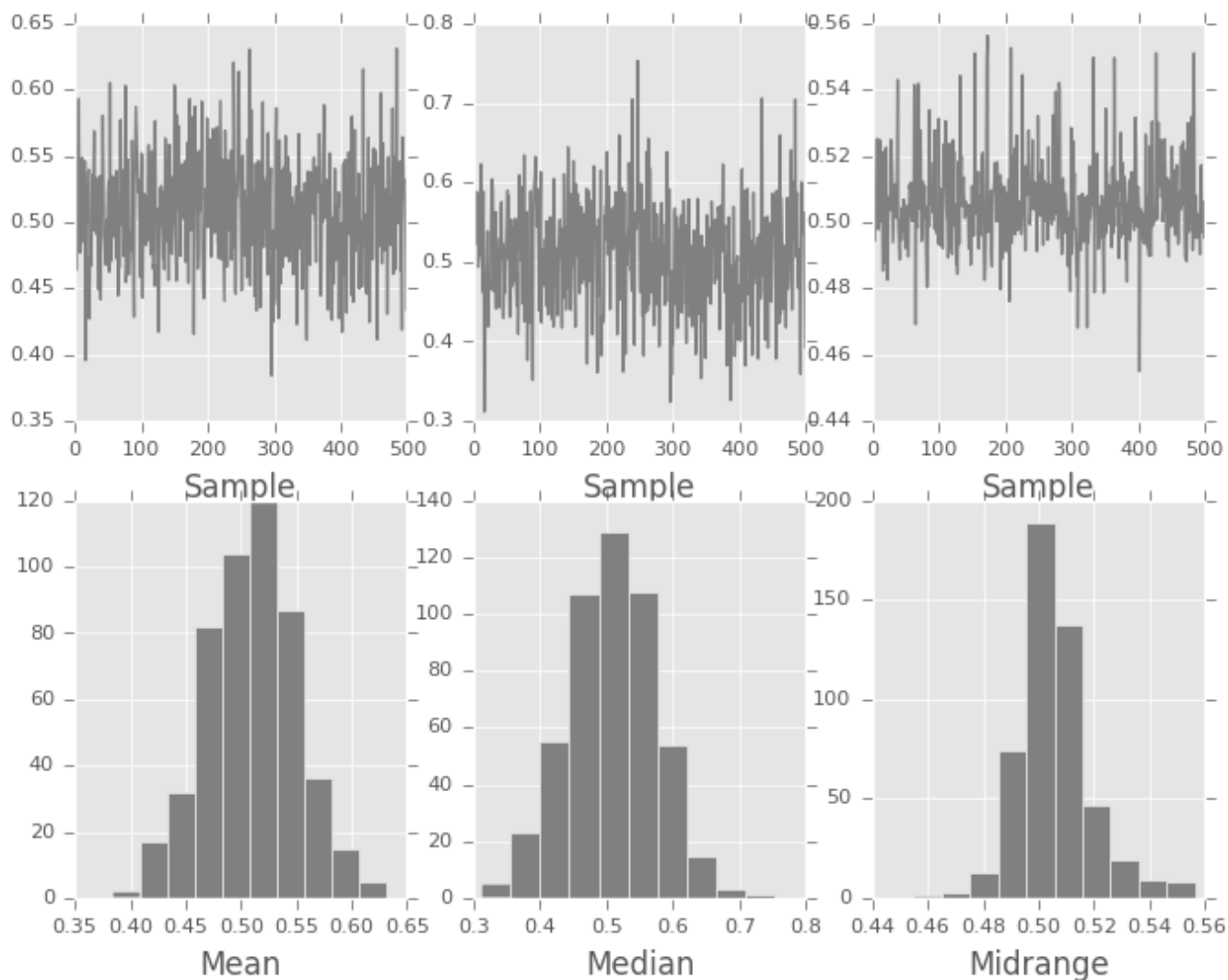
In [100]: autocorrelation_plot(data)
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbdaaec0>
```



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 [101]: from pandas.tools.plotting import bootstrap_plot
In [102]: data = pd.Series(np.random.rand(1000))
In [103]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[103]: <matplotlib.figure.Figure at 0x7f3daac13d50>
```

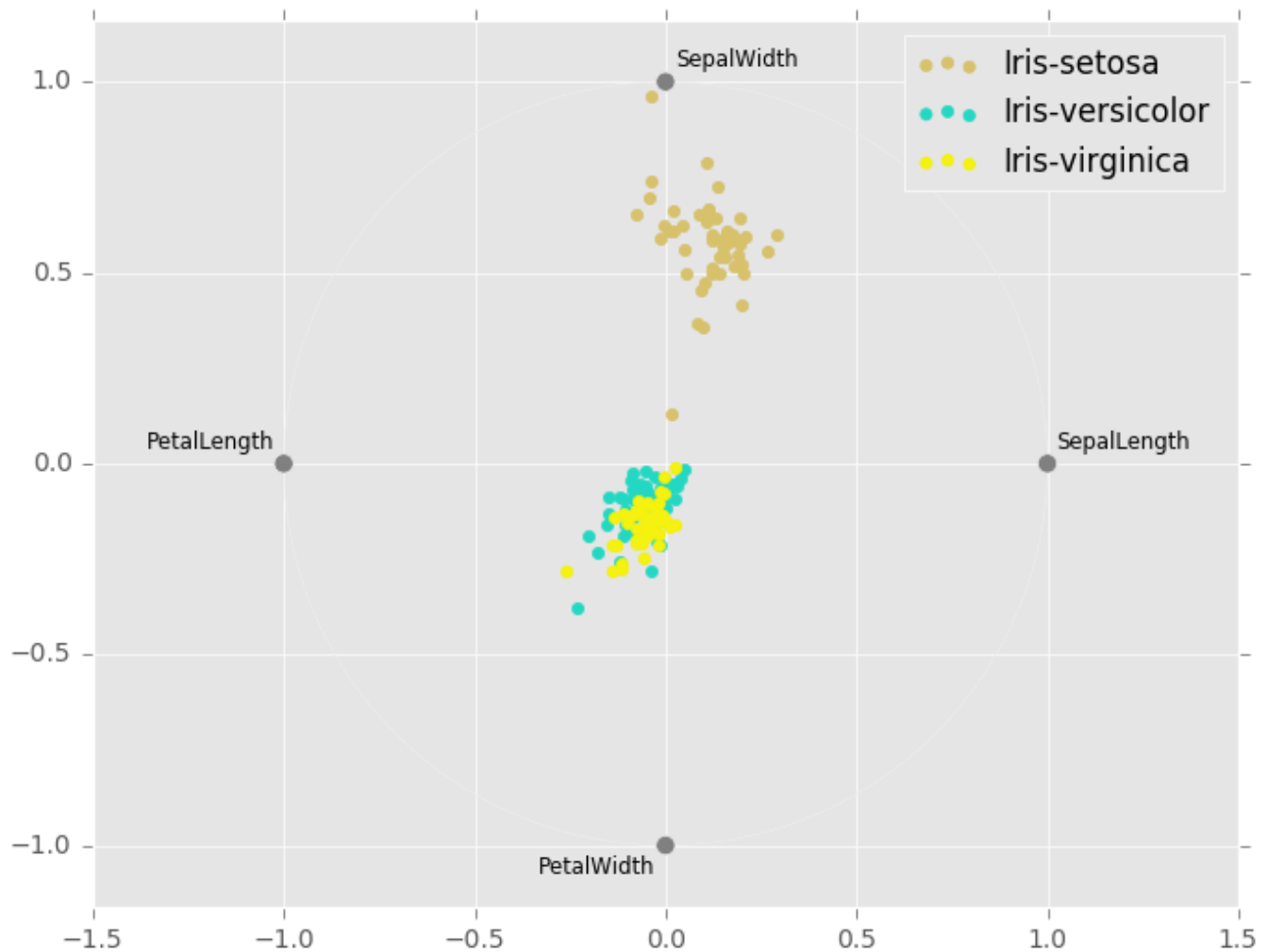


RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs it will be colored differently.

Note: The “Iris” dataset is available [here](#).

```
In [104]: from pandas.tools.plotting import radviz
In [105]: data = pd.read_csv('data/iris.data')
In [106]: plt.figure()
Out[106]: <matplotlib.figure.Figure at 0x7f3dbaa17510>
In [107]: radviz(data, 'Name')
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8b14750>
```



Plot Formatting

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [108]: plt.figure(); ts.plot(style='k--', label='Series');
```

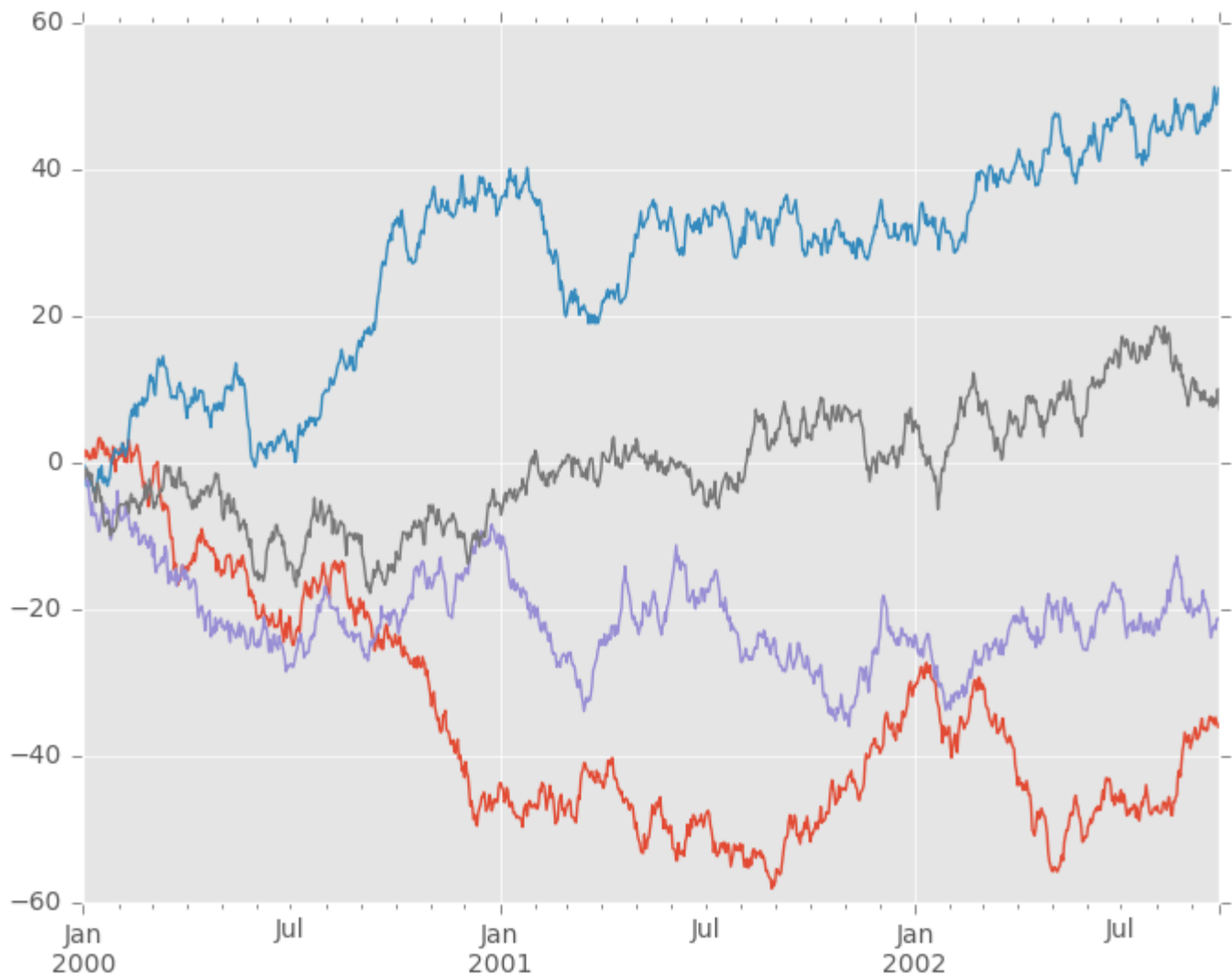


For each kind of plot (e.g. *line*, *bar*, *scatter*) any additional arguments keywords are passed along to the corresponding matplotlib function (`ax.plot()`, `ax.bar()`, `ax.scatter()`). These can be used to control additional styling, beyond what pandas provides.

Controlling the Legend

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

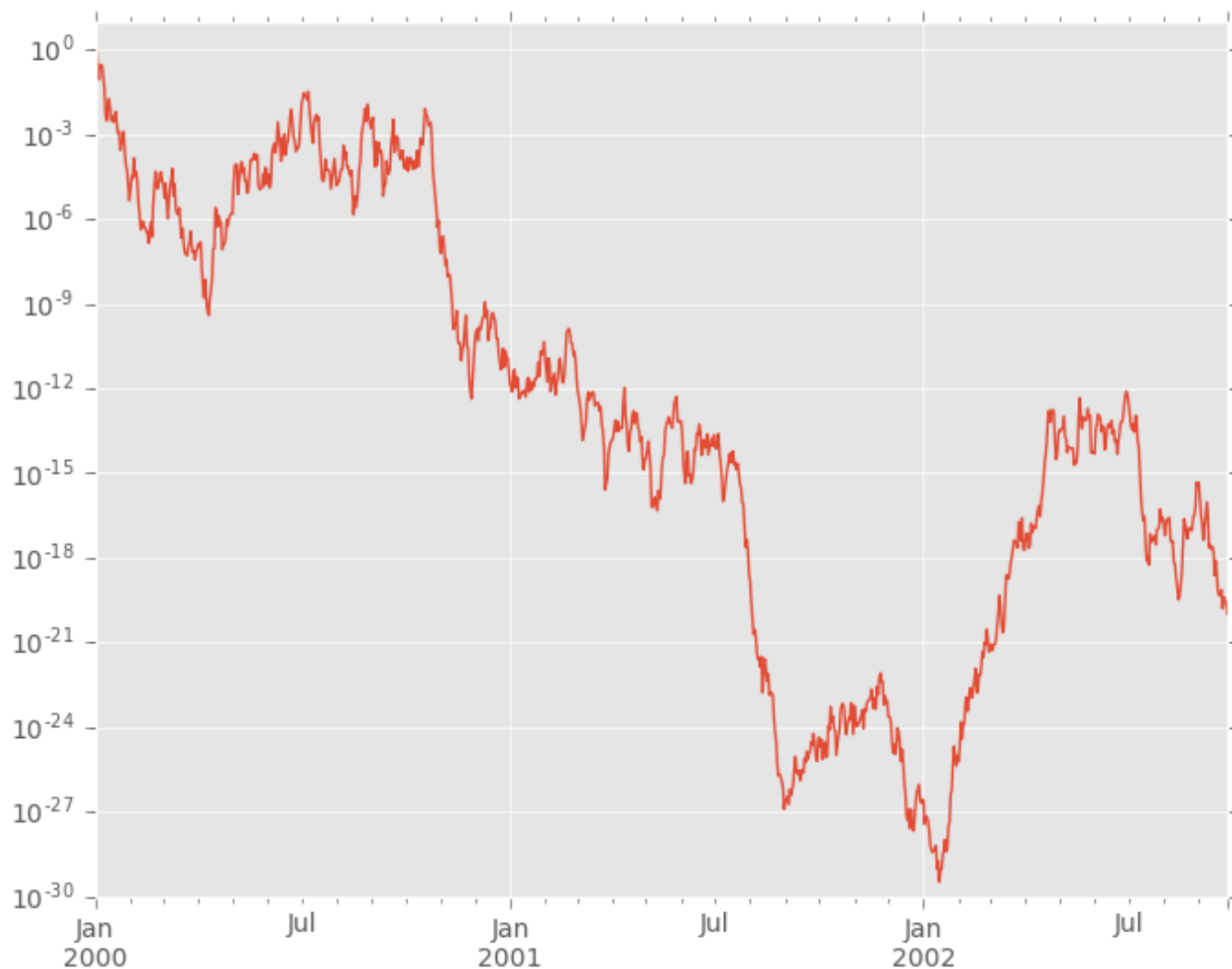
```
In [109]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD'))
In [110]: df = df.cumsum()
In [111]: df.plot(legend=False)
Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8ab7610>
```



Scales

You may pass `logy` to get a log-scale Y axis.

```
In [112]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
In [113]: ts = np.exp(ts.cumsum())
In [114]: ts.plot(logy=True)
Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dba523f10>
```



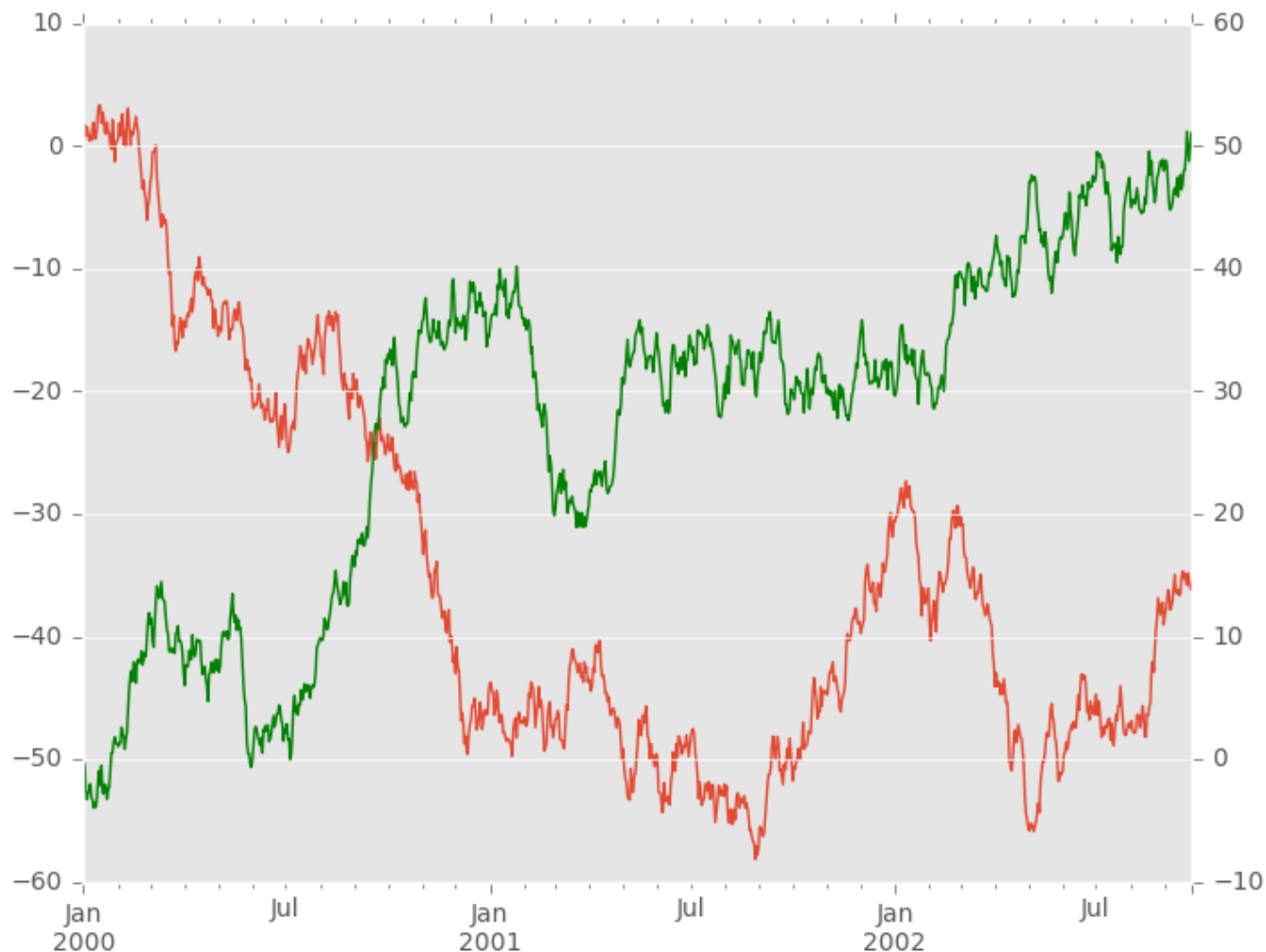
See also the `logx` and `loglog` keyword arguments.

Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```
In [115]: df.A.plot()
Out[115]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbeeb8090>

In [116]: df.B.plot(secondary_y=True, style='g')
Out[116]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbe650710>
```

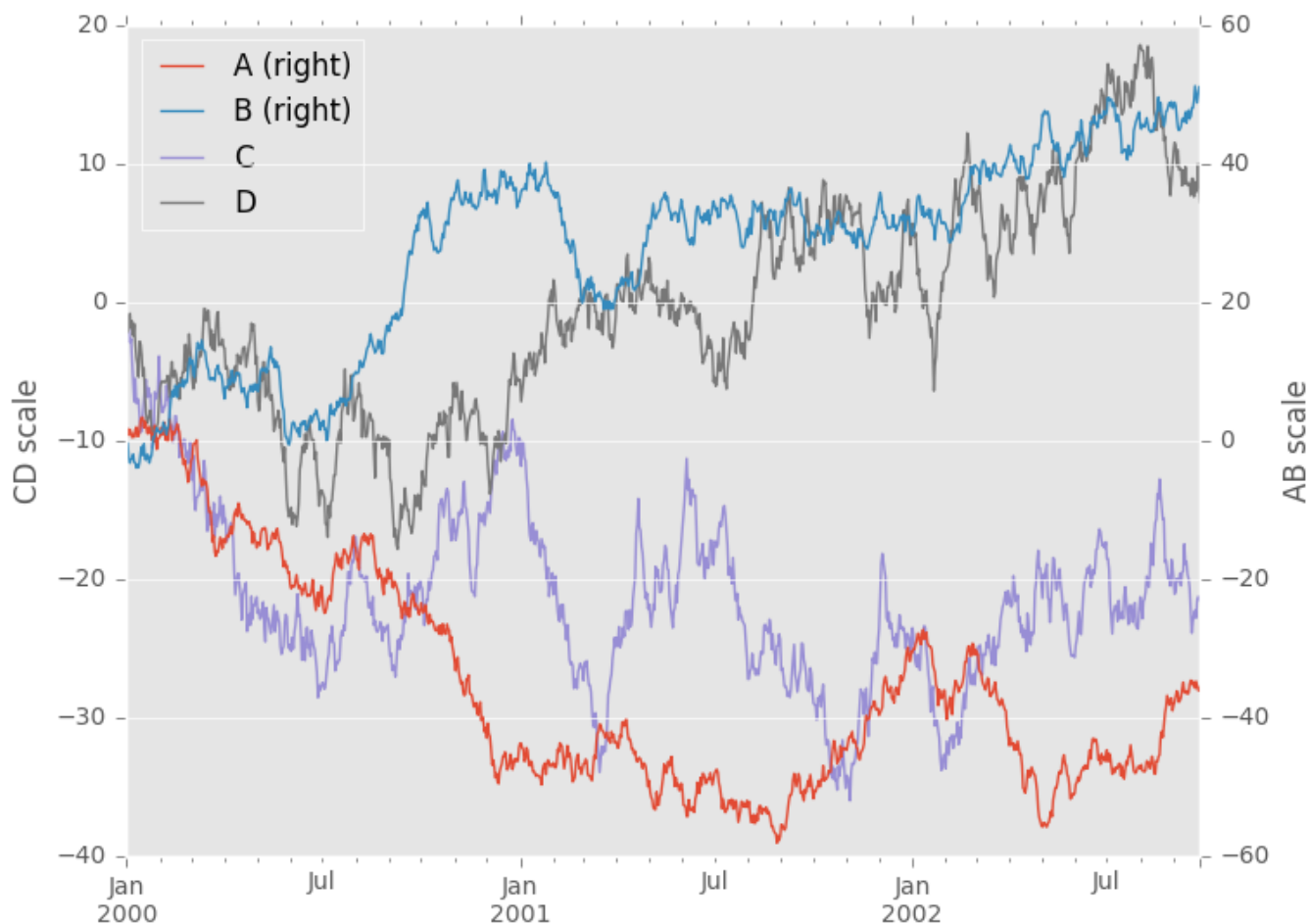
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```
In [117]: plt.figure()
Out[117]: <matplotlib.figure.Figure at 0x7f3db8f63a50>

In [118]: ax = df.plot(secondary_y=['A', 'B'])

In [119]: ax.set_ylabel('CD scale')
Out[119]: <matplotlib.text.Text at 0x7f3da9544a10>

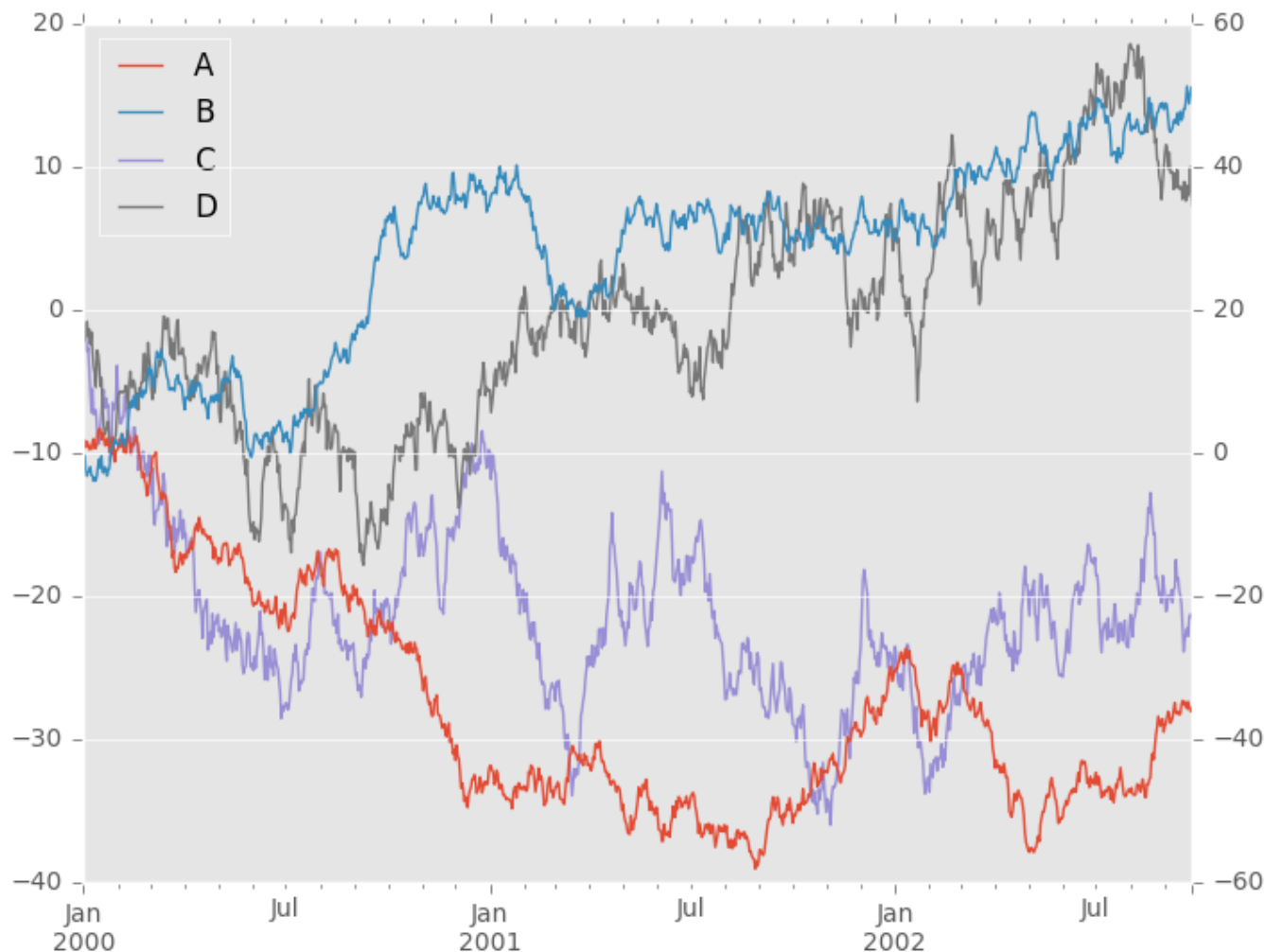
In [120]: ax.right_ax.set_ylabel('AB scale')
Out[120]: <matplotlib.text.Text at 0x7f3daadcec50>
```



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 [121]: plt.figure()
Out[121]: <matplotlib.figure.Figure at 0x7f3db8066f50>

In [122]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[122]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8060b10>
```



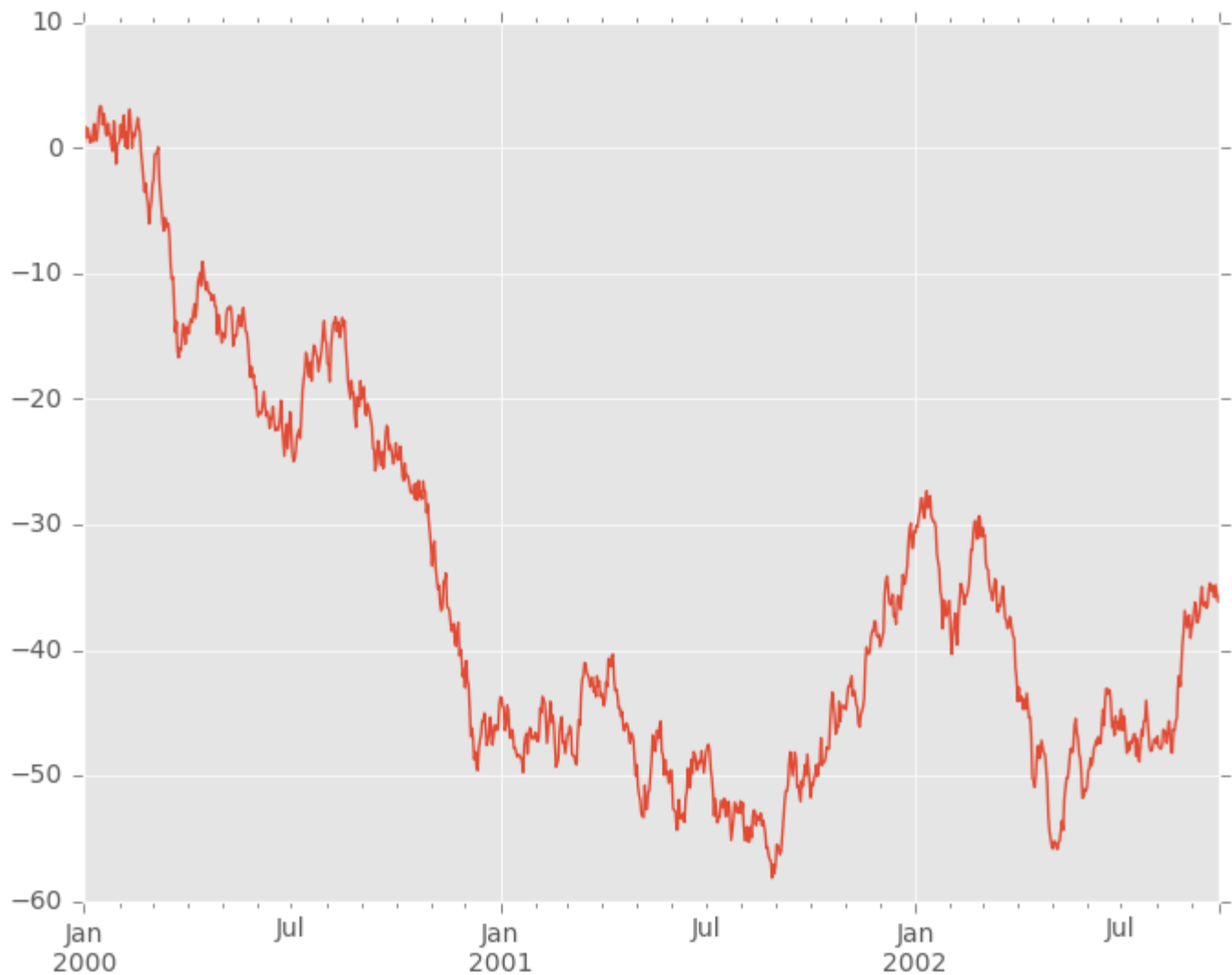
Suppressing Tick Resolution Adjustment

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 labelling is performed:

```
In [123]: plt.figure()
Out[123]: <matplotlib.figure.Figure at 0x7f3dbe14dd90>

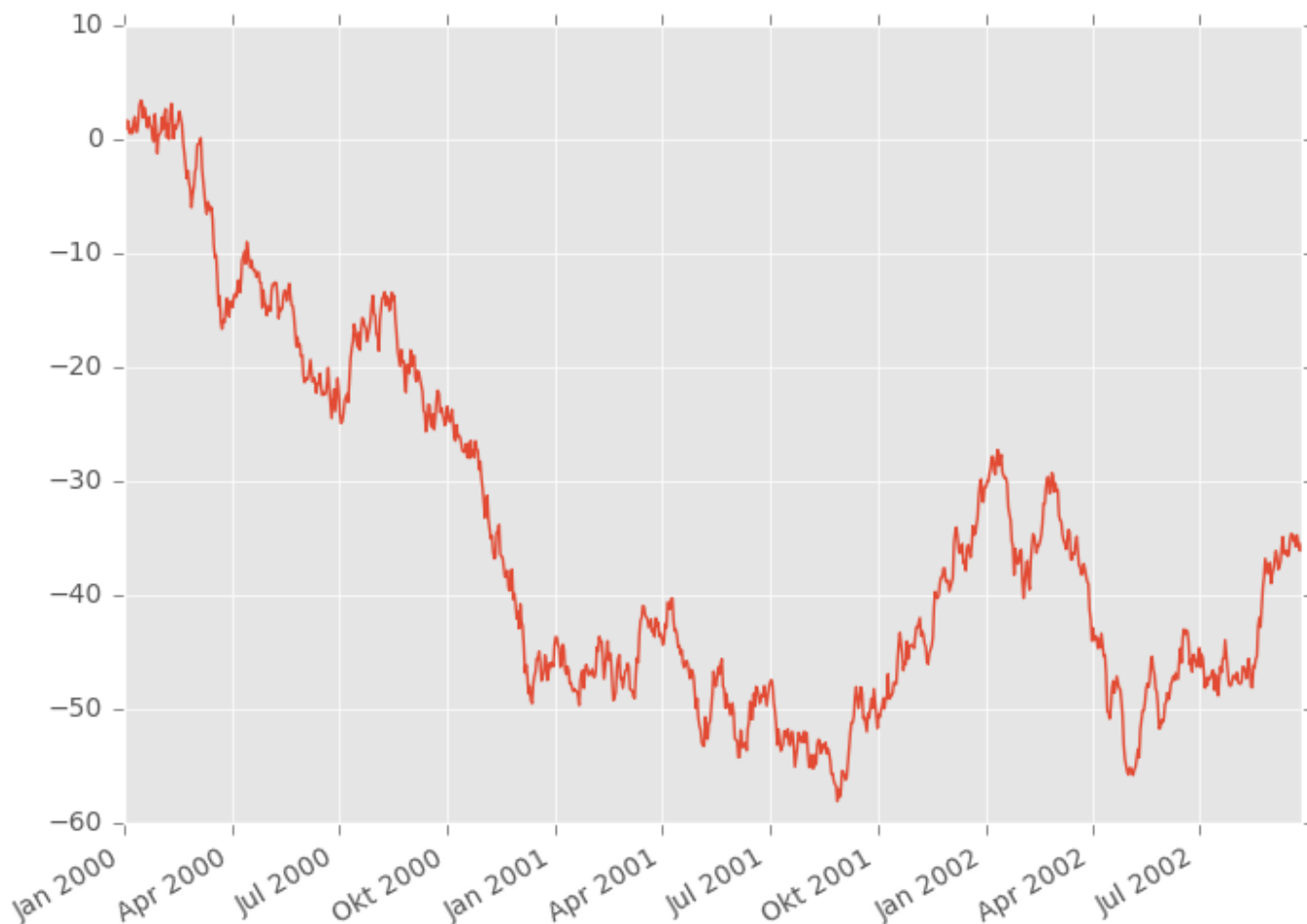
In [124]: df.A.plot()
Out[124]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3db8ff6710>
```



Using the `x_compat` parameter, you can suppress this behavior:

```
In [125]: plt.figure()
Out[125]: <matplotlib.figure.Figure at 0x7f3da9e38b10>

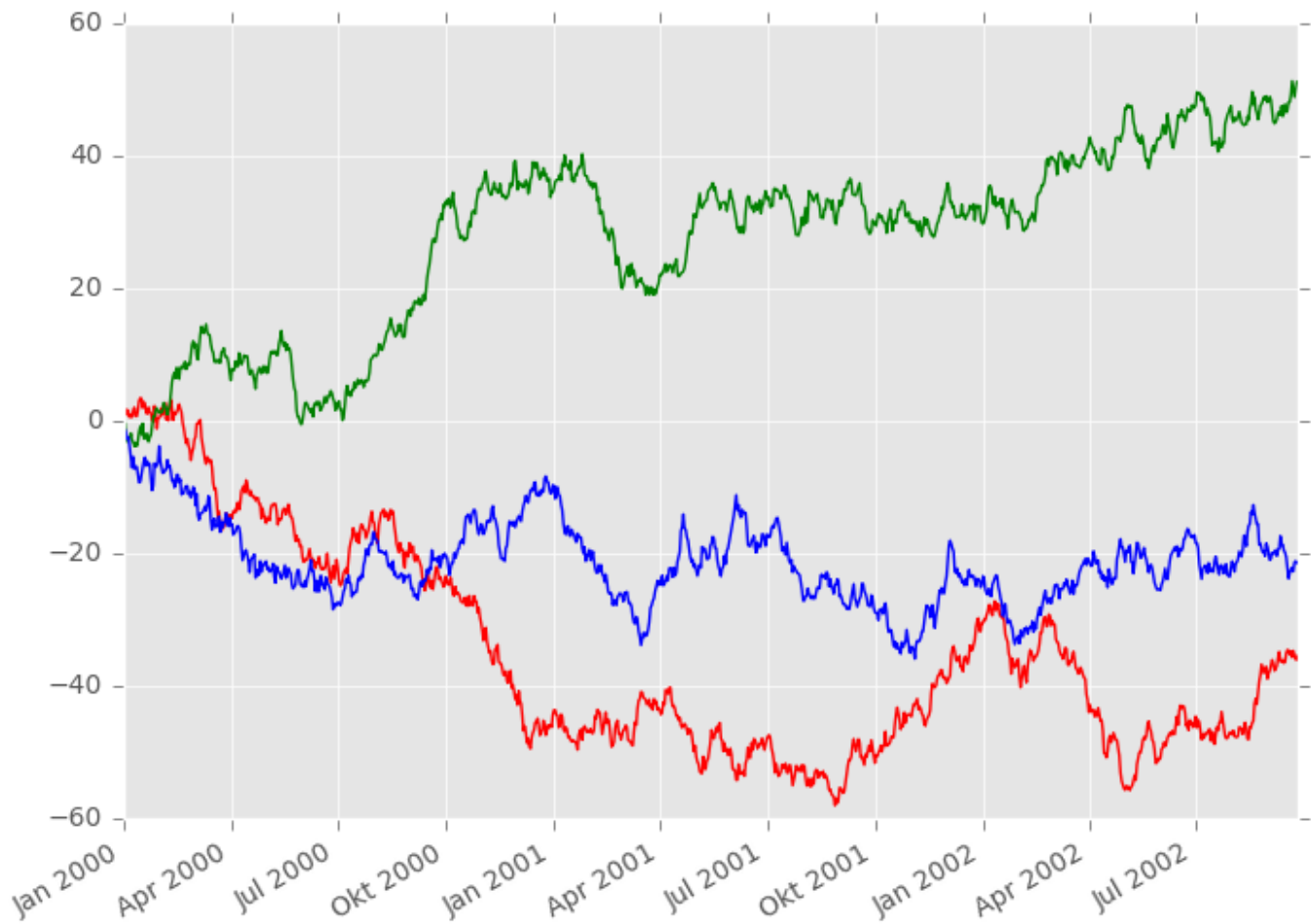
In [126]: df.A.plot(x_compat=True)
Out[126]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3da9e4a6d0>
```



If you have more than one plot that needs to be suppressed, the use method in `pandas.plot_params` can be used in a *with statement*:

```
In [127]: plt.figure()
Out[127]: <matplotlib.figure.Figure at 0x7f3db9e69250>

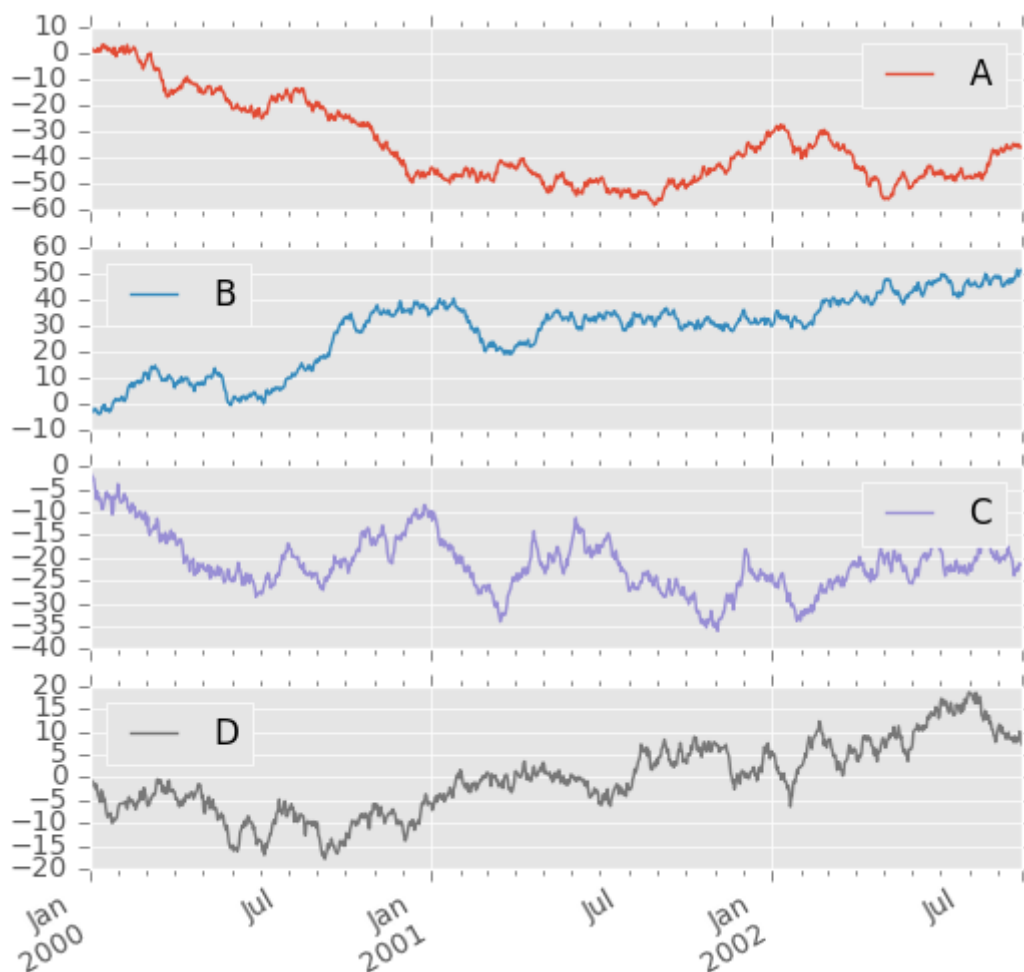
In [128]: with pd.plot_params.use('x_compat', True):
.....:     df.A.plot(color='r')
.....:     df.B.plot(color='g')
.....:     df.C.plot(color='b')
.....:
```



Subplots

Each Series in a DataFrame can be plotted on a different axis with the `subplots` keyword:

```
In [129]: df.plot(subplots=True, figsize=(6, 6));
```

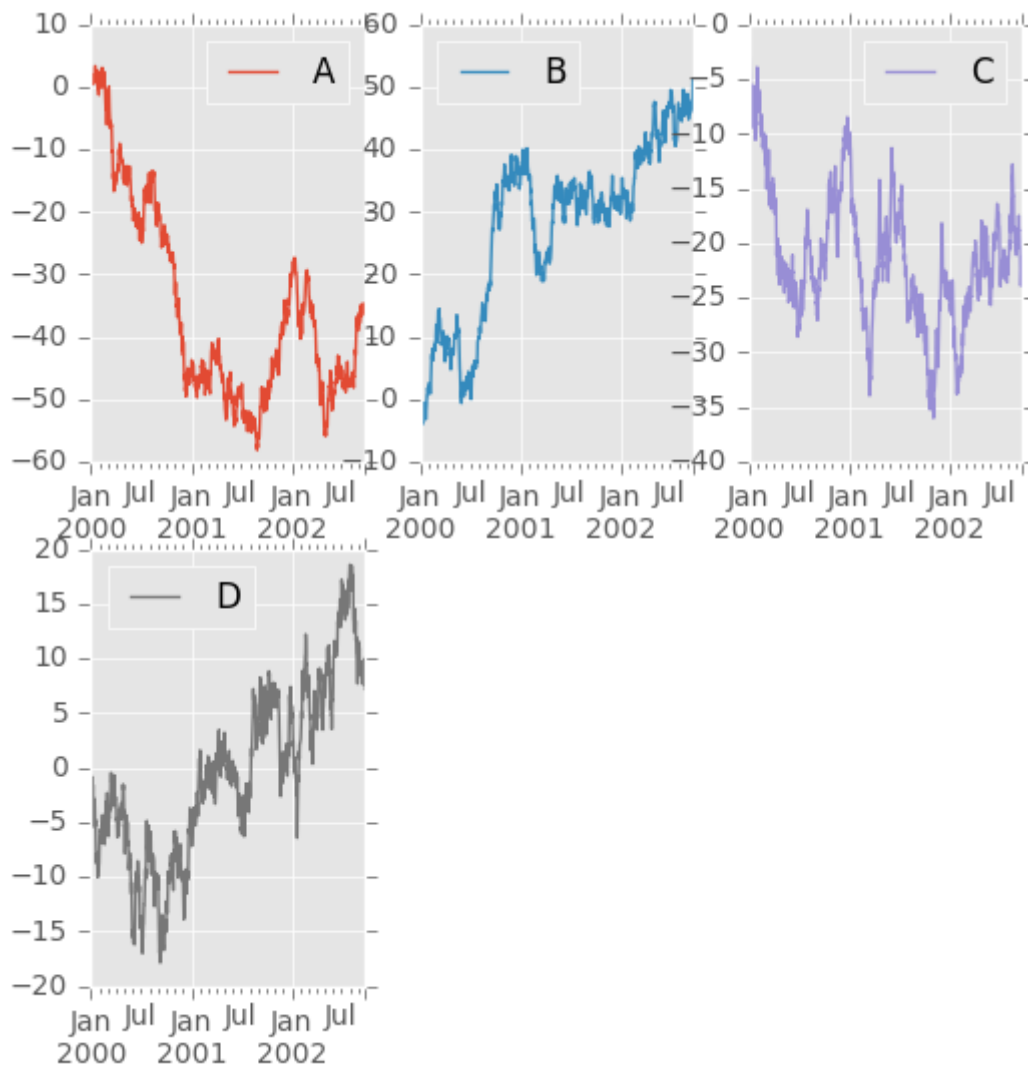


Using Layout and Targeting Multiple Axes

The layout of subplots can be specified by `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If input is invalid, `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 [130]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```



The above example is identical to using

```
In [131]: 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).

Also, you can pass multiple axes created beforehand as list-like via `ax` keyword. This allows to use more complicated layout. The passed axes must be the same number as the subplots being drawn.

When multiple axes are passed via `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 [132]: fig, axes = plt.subplots(4, 4, figsize=(6, 6));
```

```
In [133]: plt.subplots_adjust(wspace=0.5, hspace=0.5);
```

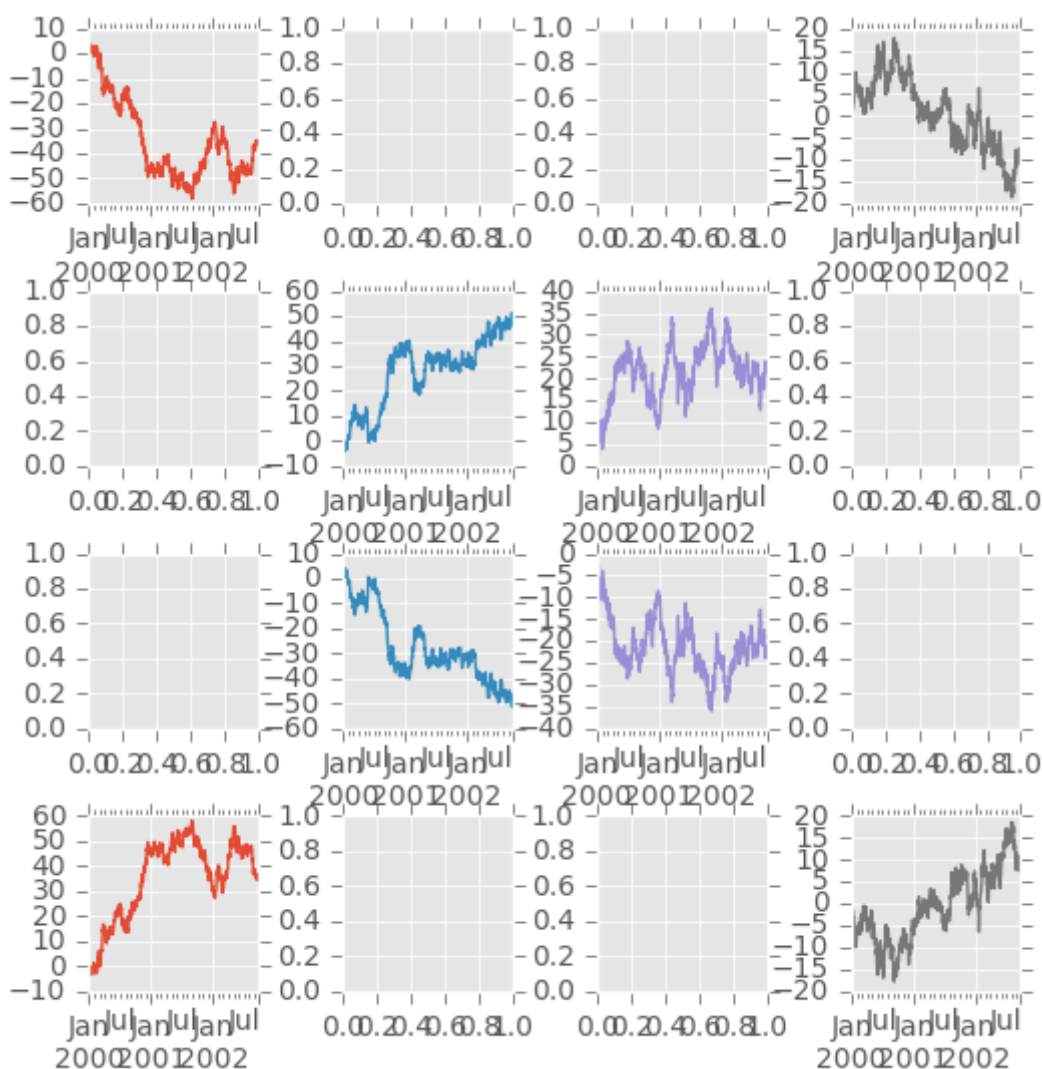
```
In [134]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
```

```
In [135]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]
```



```
In [136]: df.plot(subplots=True, ax=target1, legend=False, sharex=False, sharey=False);
```

```
In [137]: (-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:

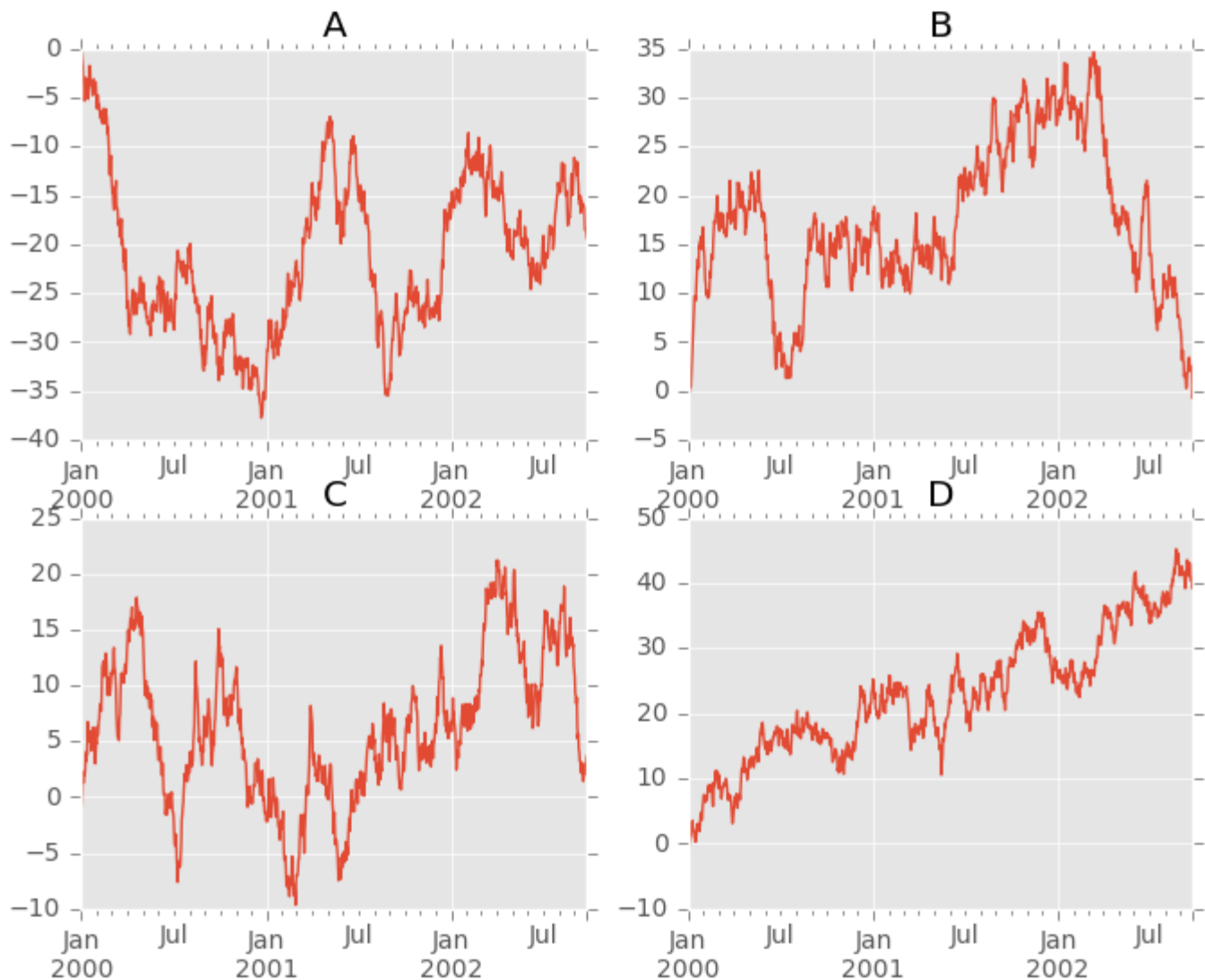
```
In [138]: fig, axes = plt.subplots(nrows=2, ncols=2)
```

```
In [139]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A');
```

```
In [140]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B');
```

```
In [141]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C');
```

```
In [142]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D');
```



Plotting With Error Bars

New in version 0.14.

Plotting with error bars is now supported in the `DataFrame.plot()` and `Series.plot()`

Horizontal and vertical errorbars 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 [143]: ix3 = pd.MultiIndex.from_arrays([[ 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'], [ 'foo', 'foo', 'bar', 'bar', 'foo', 'bar', 'bar', 'bar']])

In [144]: df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2], 'data2': [6, 5, 7, 5, 4, 5, 6, 4]})

# Group by index labels and take the means and standard deviations for each group
In [145]: gp3 = df3.groupby(level=('letter', 'word'))

In [146]: means = gp3.mean()

In [147]: errors = gp3.std()

In [148]: means
Out[148]:
```

		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

```

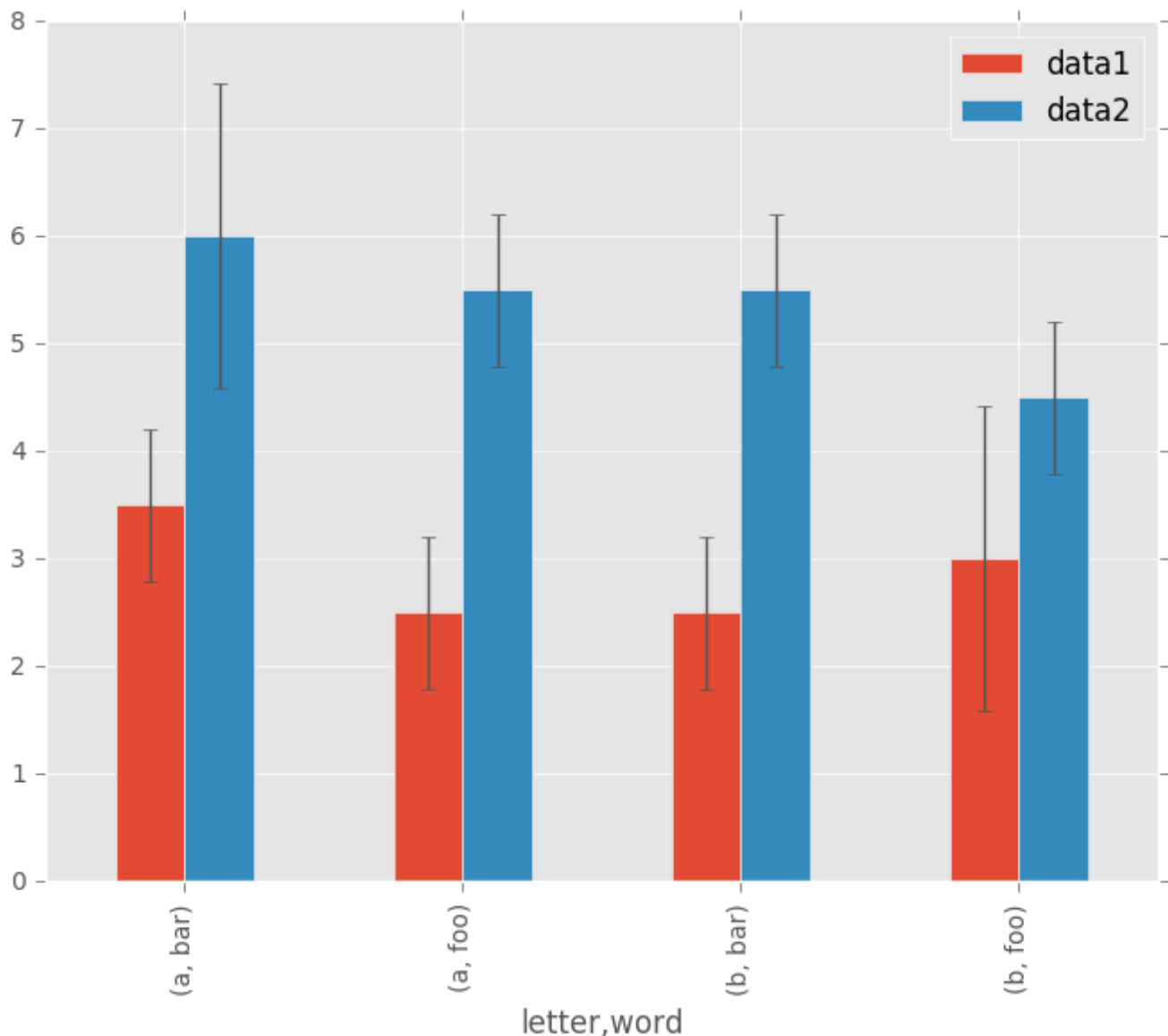

In [149]: errors
Out[149]:
```

		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 [150]: fig, ax = plt.subplots()

In [151]: means.plot.bar(yerr=errors, ax=ax)
Out[151]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dba3cc9d0>
```

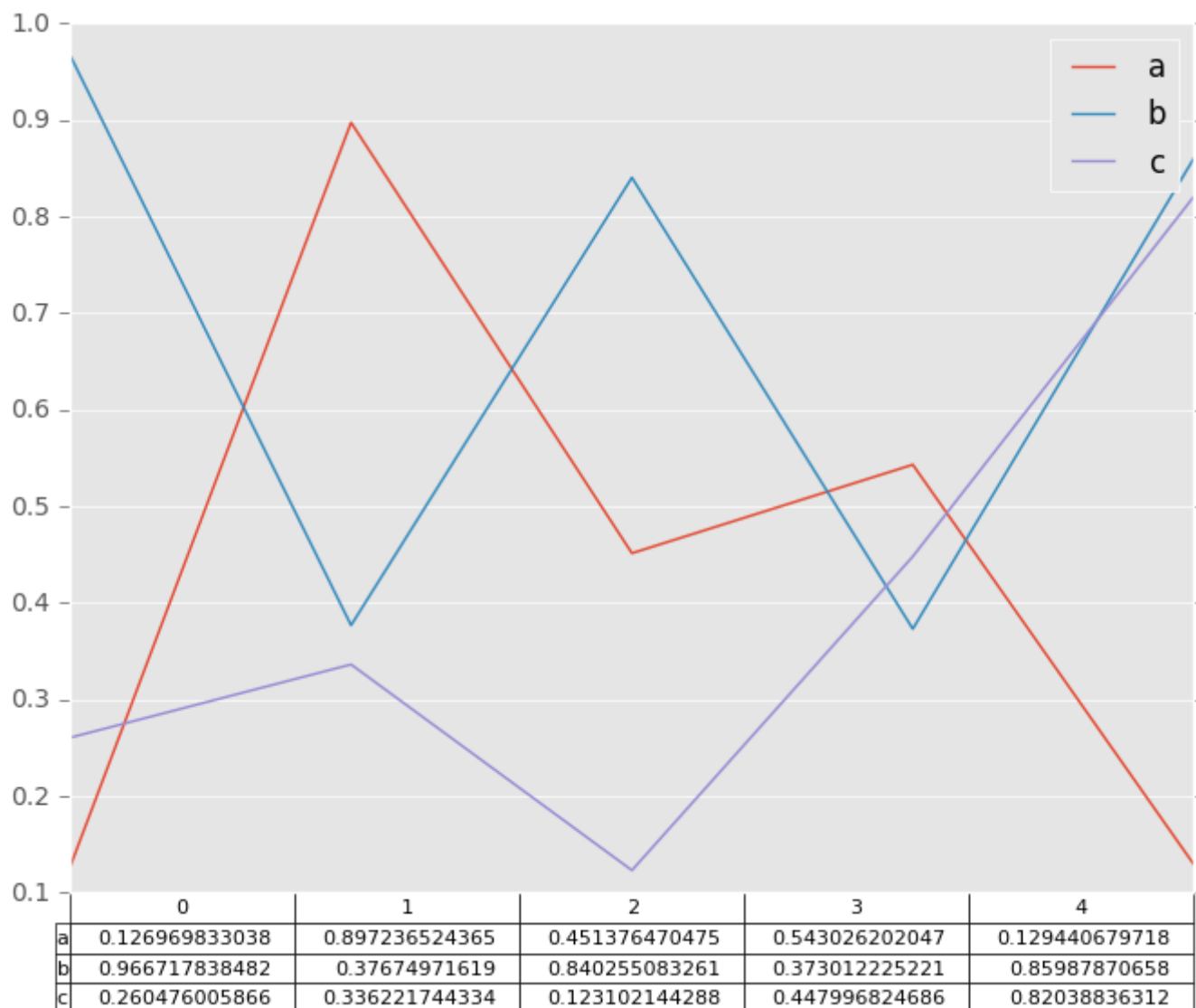


Plotting Tables

New in version 0.14.

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 [152]: fig, ax = plt.subplots(1, 1)
In [153]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])
In [154]: ax.get_xaxis().set_visible(False)    # Hide Ticks
In [155]: df.plot(table=True, ax=ax)
Out[155]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbeee80d0>
```

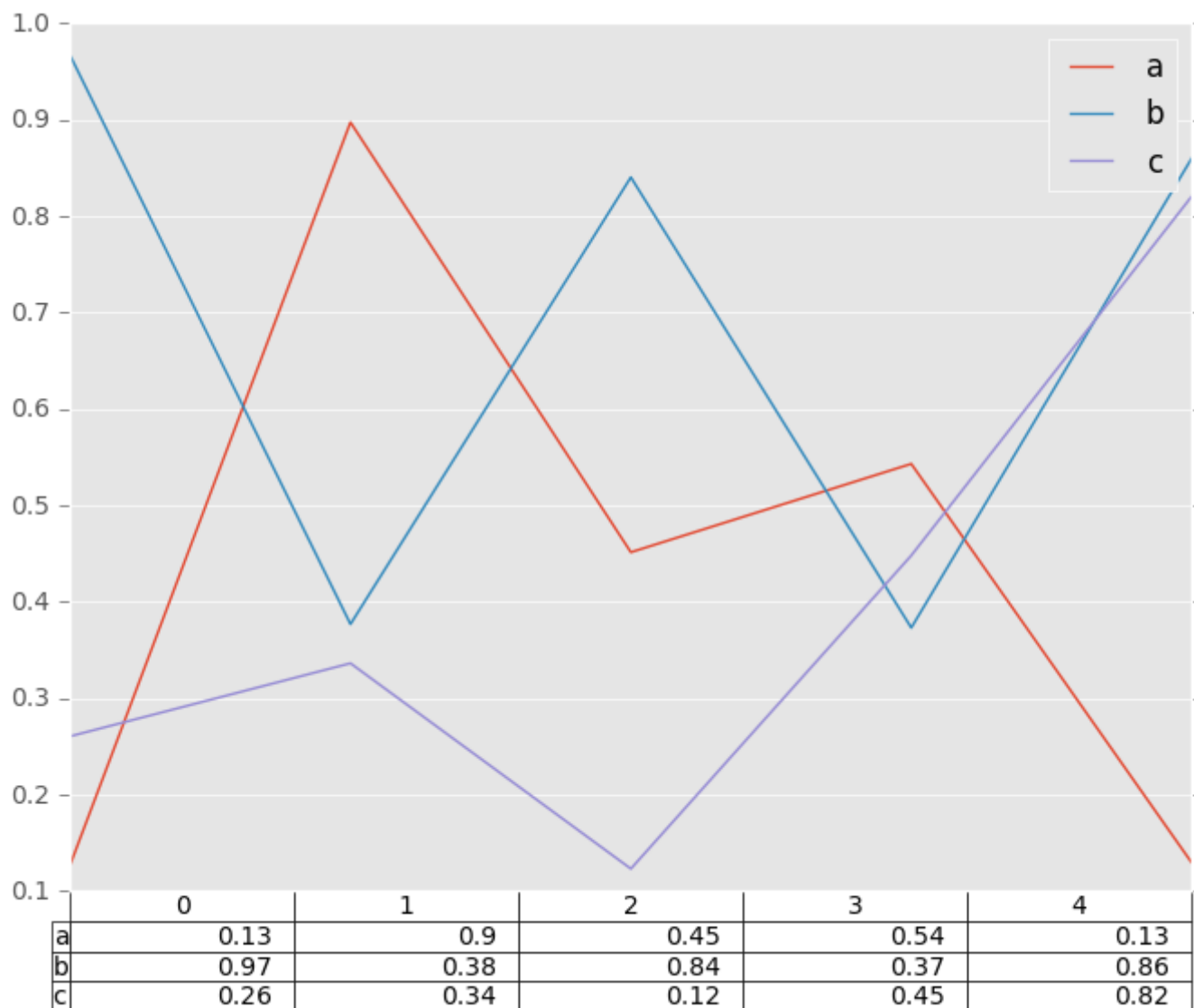


Also, you can pass different `DataFrame` or `Series` for `table` keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as below example.

```
In [156]: fig, ax = plt.subplots(1, 1)

In [157]: ax.get_xaxis().set_visible(False) # Hide Ticks

In [158]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbbc8b310>
```



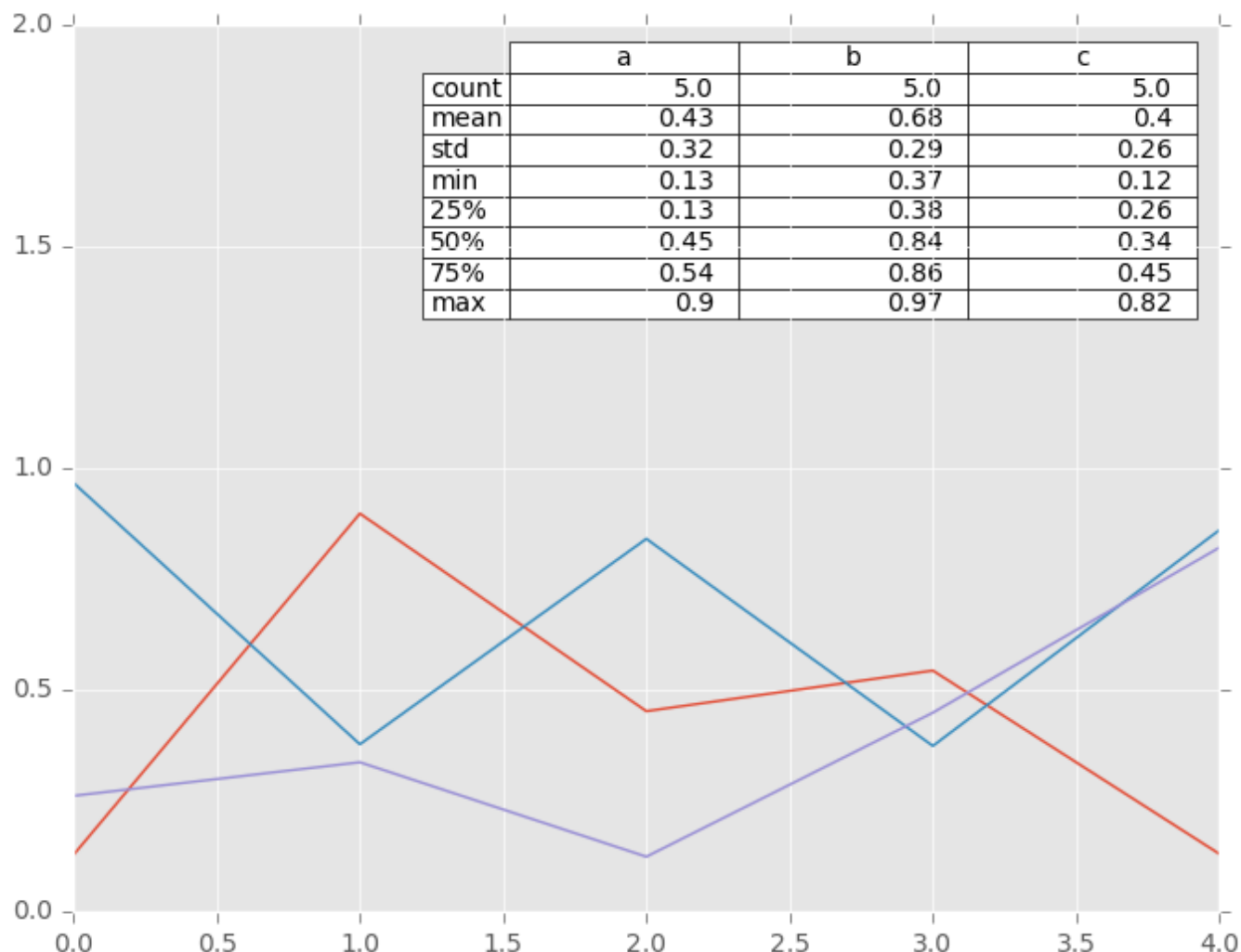
Finally, there is a helper function `pandas.tools.plotting.table` to create a table from `DataFrame` and `Series`, and add it to an `matplotlib.Axes`. This function can accept keywords which `matplotlib` table has.

```
In [159]: from pandas.tools.plotting import table

In [160]: fig, ax = plt.subplots(1, 1)

In [161]: table(ax, np.round(df.describe(), 2),
.....:         loc='upper right', colWidths=[0.2, 0.2, 0.2])
.....:
Out[161]: <matplotlib.table.Table at 0x7f3da9b03690>

In [162]: df.plot(ax=ax, ylim=(0, 2), legend=None)
Out[162]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbef08750>
```



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](#).

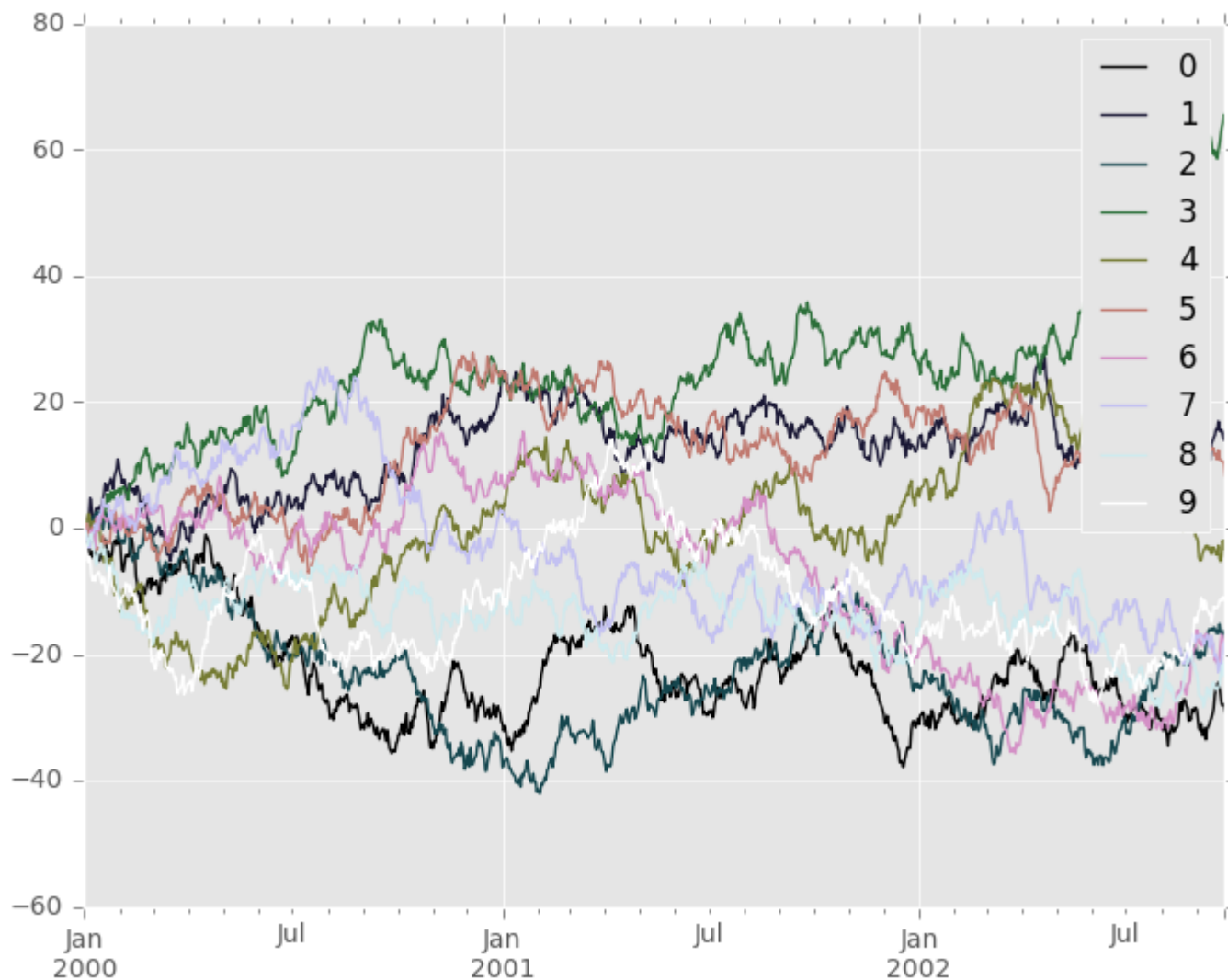
As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the DataFrame. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

To use the cubehelix colormap, we can simply pass 'cubehelix' to `colormap=`

```
In [163]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
In [164]: df = df.cumsum()
In [165]: plt.figure()
```

```
Out[165]: <matplotlib.figure.Figure at 0x7f3dbbab06d0>

In [166]: df.plot(colormap='cubehelix')
Out[166]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbf325410>
```

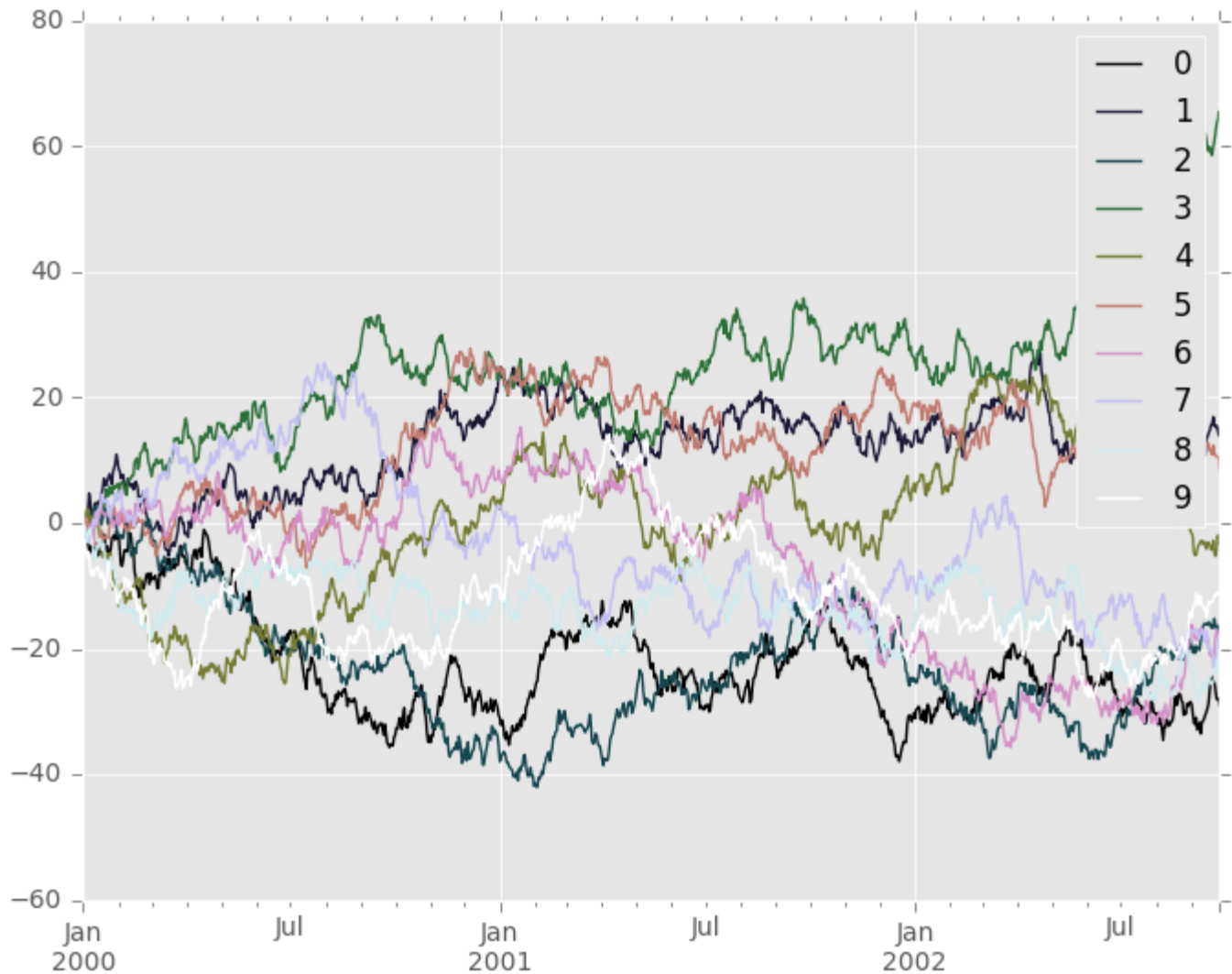


or we can pass the colormap itself

```
In [167]: from matplotlib import cm

In [168]: plt.figure()
Out[168]: <matplotlib.figure.Figure at 0x7f3dbd6d1f90>

In [169]: df.plot(colormap=cm.cubehelix)
Out[169]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbad2ba10>
```

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

```
In [170]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
```

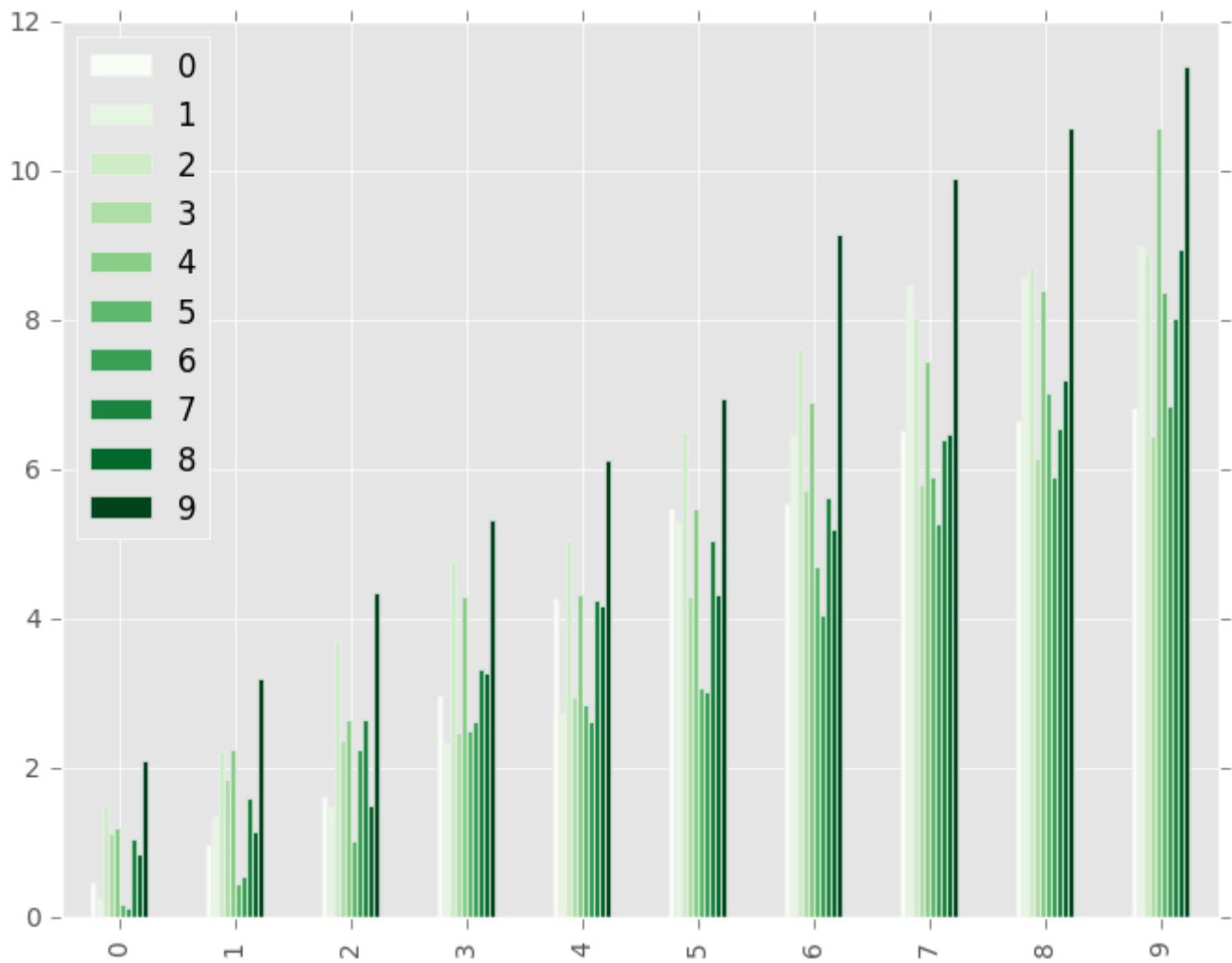
```
In [171]: dd = dd.cumsum()
```

```
In [172]: plt.figure()
```

```
Out[172]: <matplotlib.figure.Figure at 0x7f3dbbd97c50>
```

```
In [173]: dd.plot.bar(colormap='Greens')
```

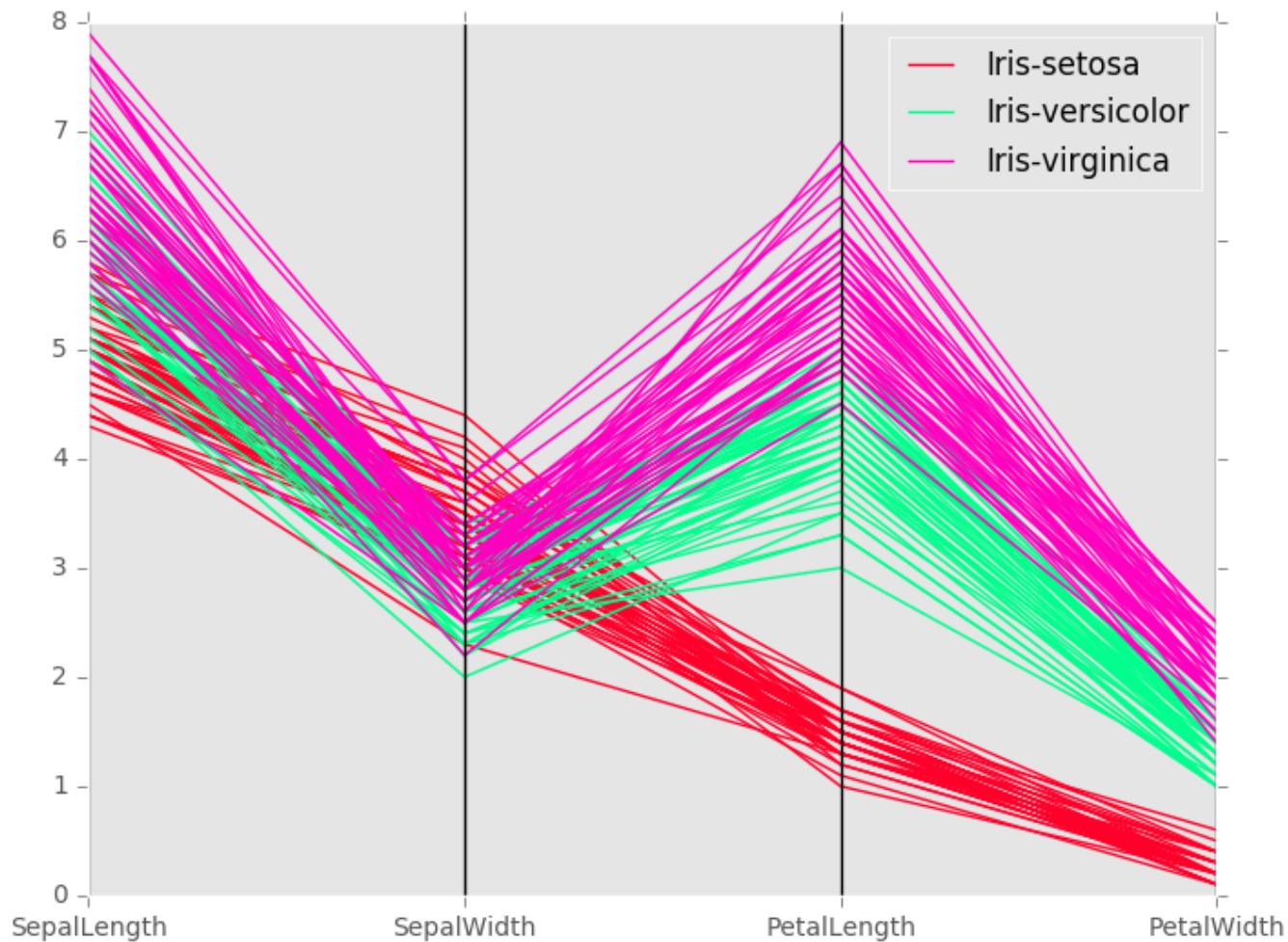
```
Out[173]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbb691f50>
```



Parallel coordinates charts:

```
In [174]: plt.figure()
Out[174]: <matplotlib.figure.Figure at 0x7f3dbea13890>

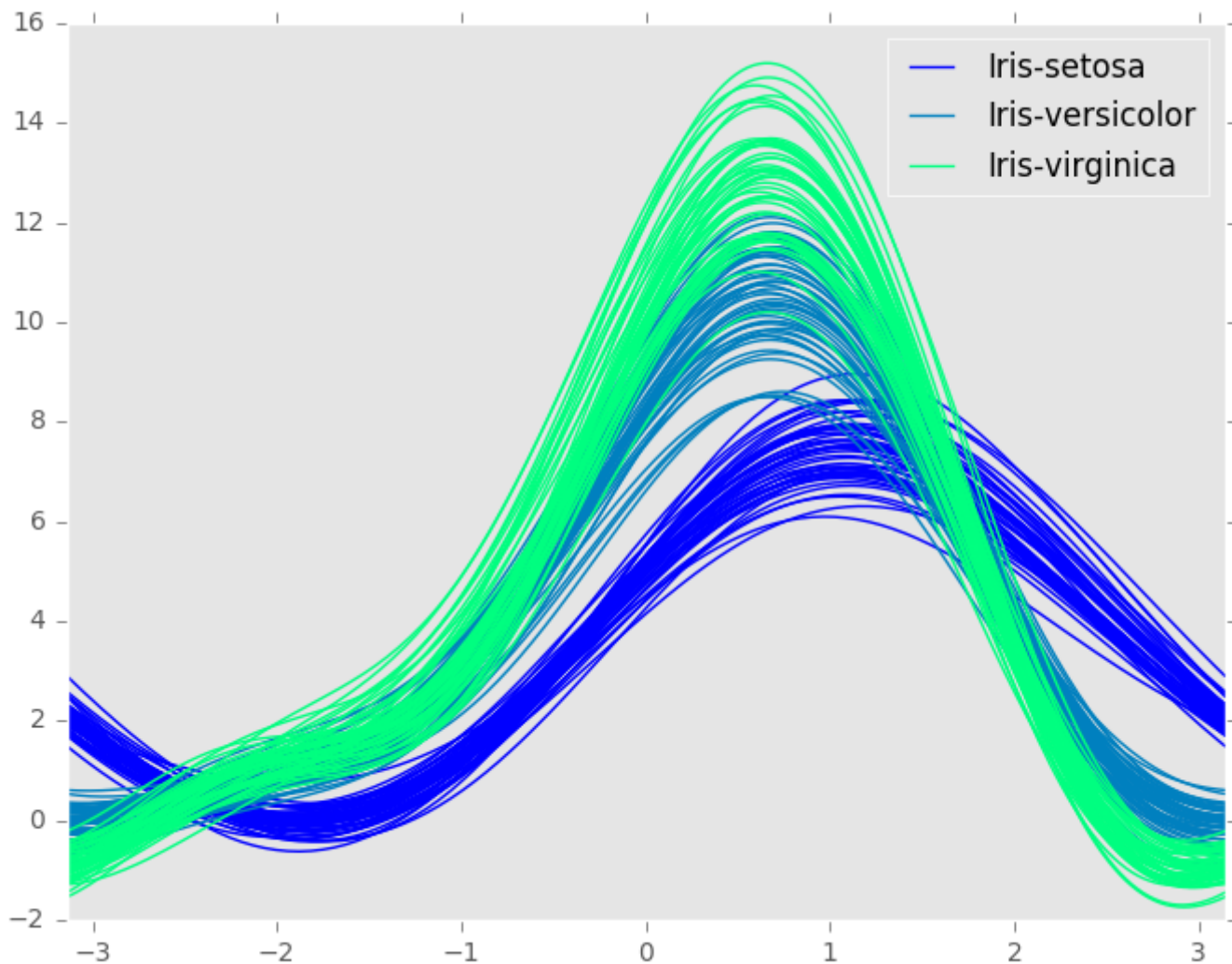
In [175]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
Out[175]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dbe9fa950>
```



Andrews curves charts:

```
In [176]: plt.figure()
Out[176]: <matplotlib.figure.Figure at 0x7f3dba665a90>

In [177]: andrews_curves(data, 'Name', colormap='winter')
Out[177]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3dba655d50>
```



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.

Note: The speed up for large data sets only applies to pandas 0.14.0 and later.

```
In [178]: price = pd.Series(np.random.randn(150).cumsum(),  
.....:                    index=pd.date_range('2000-1-1', periods=150, freq='B'))  
.....:  
In [179]: ma = price.rolling(20).mean()
```

```

In [180]: mstd = price.rolling(20).std()

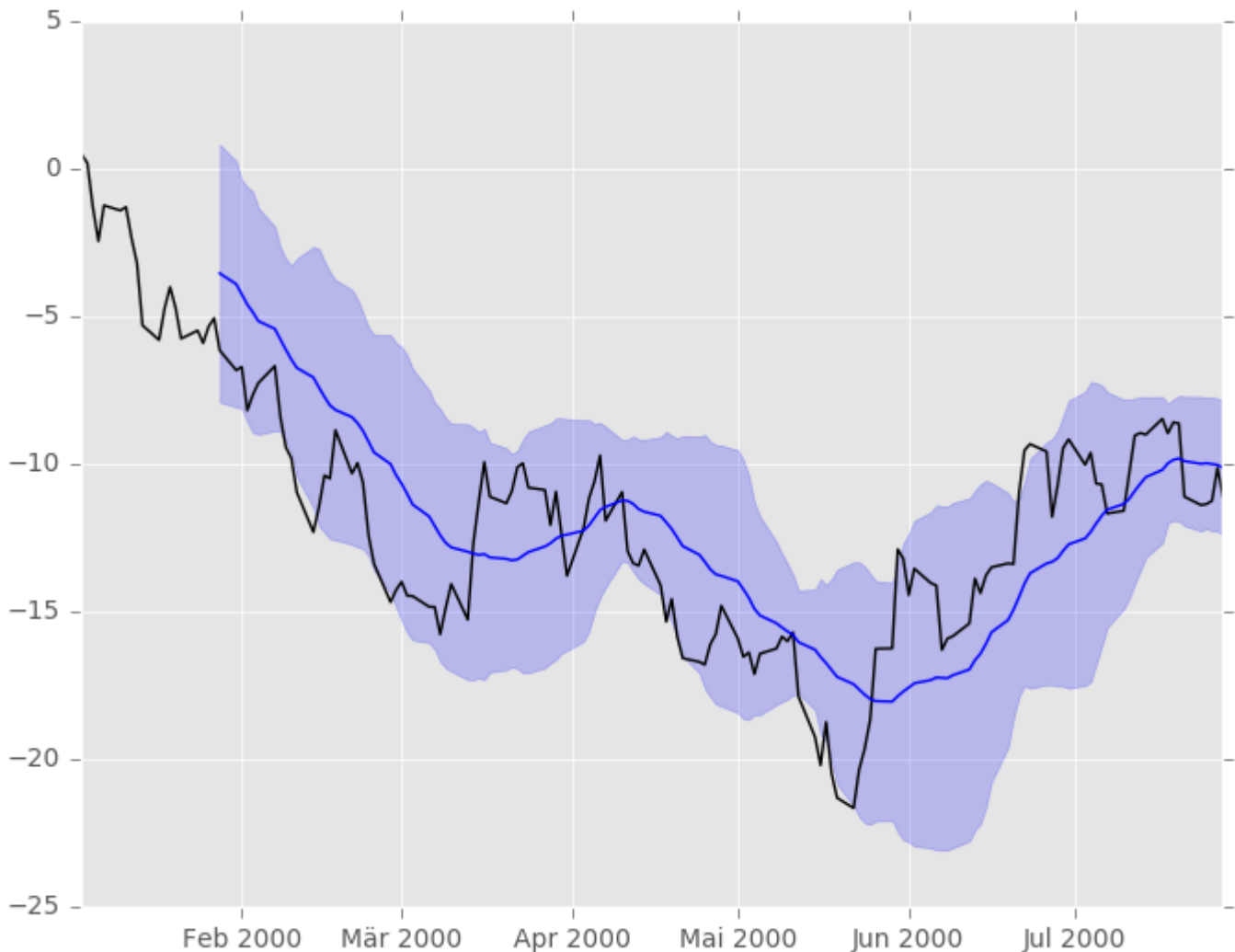
In [181]: plt.figure()
Out[181]: <matplotlib.figure.Figure at 0x7f3daaa31d10>

In [182]: plt.plot(price.index, price, 'k')
Out[182]: [<matplotlib.lines.Line2D at 0x7f3daaa17490>]

In [183]: plt.plot(ma.index, ma, 'b')
Out[183]: [<matplotlib.lines.Line2D at 0x7f3daaa19c50>]

In [184]: plt.fill_between(mstd.index, ma-2*mstd, ma+2*mstd, color='b', alpha=0.2)
Out[184]: <matplotlib.collections.PolyCollection at 0x7f3dbd31ab50>

```



Trellis plotting interface

Warning: The `rpplot` 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.