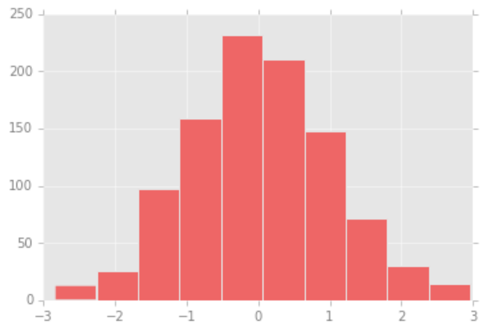

*Figure 4-83. A customized histogram using rc settings*

Let's see what simple line plots look like with these `rc` parameters (Figure 4-84):

```
In[7]: for i in range(4):
           plt.plot(np.random.rand(10))
```

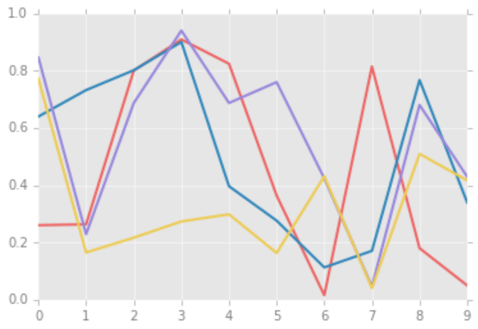

*Figure 4-84. A line plot with customized styles*

I find this much more aesthetically pleasing than the default styling. If you disagree with my aesthetic sense, the good news is that you can adjust the `rc` parameters to suit your own tastes! These settings can be saved in a *.matplotlibrc* file, which you can read about in the Matplotlib documentation. That said, I prefer to customize Matplotlib using its stylesheets instead.

## Stylesheets

The version 1.4 release of Matplotlib in August 2014 added a very convenient `style` module, which includes a number of new default stylesheets, as well as the ability to create and package your own styles. These stylesheets are formatted similarly to the *.matplotlibrc* files mentioned earlier, but must be named with a *.mplstyle* extension.

Even if you don't create your own style, the stylesheets included by default are extremely useful. The available styles are listed in `plt.style.available`—here I'll list only the first five for brevity:

```
In[8]: plt.style.available[:5]
```

```
Out[8]: ['fivethirtyeight',
         'seaborn-pastel',
         'seaborn-whitegrid',
         'ggplot',
         'grayscale']
```

The basic way to switch to a stylesheet is to call:

```
plt.style.use('stylename')
```

But keep in mind that this will change the style for the rest of the session! Alternatively, you can use the style context manager, which sets a style temporarily:

```
with plt.style.context('stylename'):
    make_a_plot()
```

Let's create a function that will make two basic types of plot:

```
In[9]: def hist_and_lines():
           np.random.seed(0)
           fig, ax = plt.subplots(1, 2, figsize=(11, 4))
           ax[0].hist(np.random.randn(1000))
           for i in range(3):
               ax[1].plot(np.random.rand(10))
           ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

We'll use this to explore how these plots look using the various built-in styles.

### Default style

The default style is what we've been seeing so far throughout the book; we'll start with that. First, let's reset our runtime configuration to the notebook default:

```
In[10]: # reset rcParams
        plt.rcParams.update(IPython_default);
```

Now let's see how it looks (Figure 4-85):

```
In[11]: hist_and_lines()
```

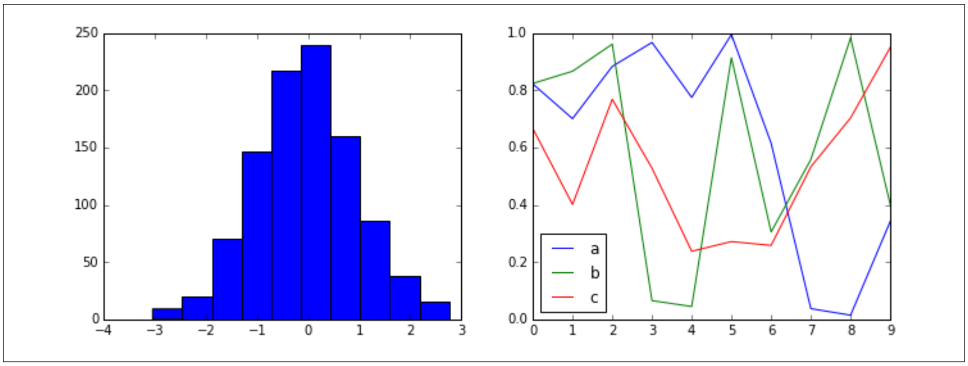

*Figure 4-85. Matplotlib's default style*

### FiveThirtyEight style

The FiveThirtyEight style mimics the graphics found on the popular FiveThirtyEight website. As you can see in Figure 4-86, it is typified by bold colors, thick lines, and transparent axes.

```
In[12]: with plt.style.context('fivethirtyeight'):
            hist_and_lines()
```

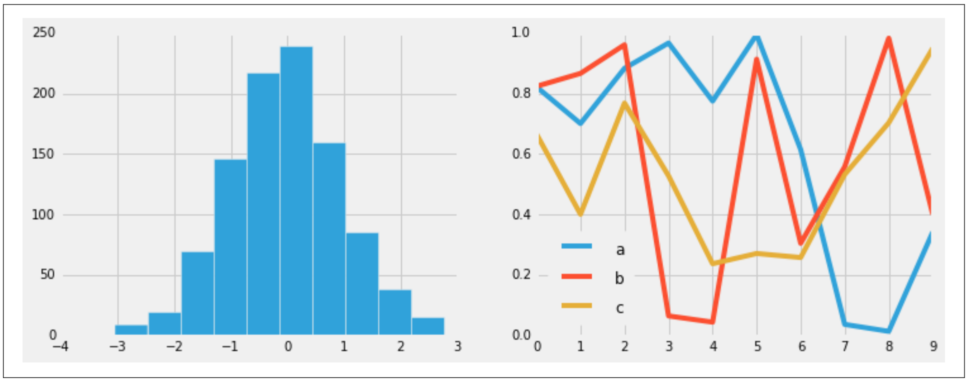

*Figure 4-86. The FiveThirtyEight style*

### ggplot

The `ggplot` package in the R language is a very popular visualization tool. Matplotlib's `ggplot` style mimics the default styles from that package (Figure 4-87):

```
In[13]: with plt.style.context('ggplot'):
            hist_and_lines()
```

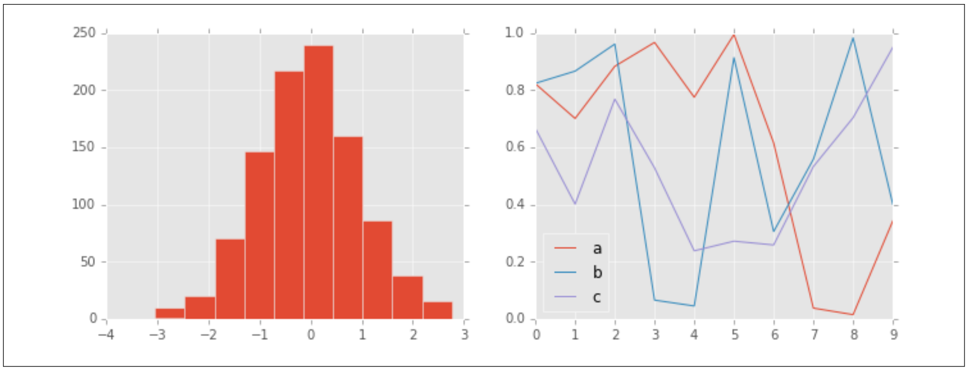

*Figure 4-87. The ggplot style*

### Bayesian Methods for Hackers style

There is a very nice short online book called *Probabilistic Programming and Bayesian Methods for Hackers*; it features figures created with Matplotlib, and uses a nice set of `rc` parameters to create a consistent and visually appealing style throughout the book. This style is reproduced in the `bmh` stylesheet (Figure 4-88):

```
In[14]: with plt.style.context('bmh'):
            hist_and_lines()
```

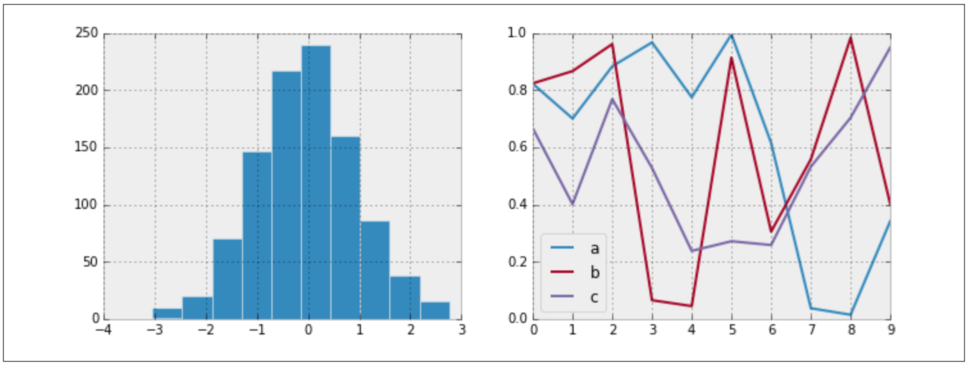

*Figure 4-88. The bmh style*

### Dark background

For figures used within presentations, it is often useful to have a dark rather than light background. The `dark_background` style provides this (Figure 4-89):

```
In[15]: with plt.style.context('dark_background'):
            hist_and_lines()
```

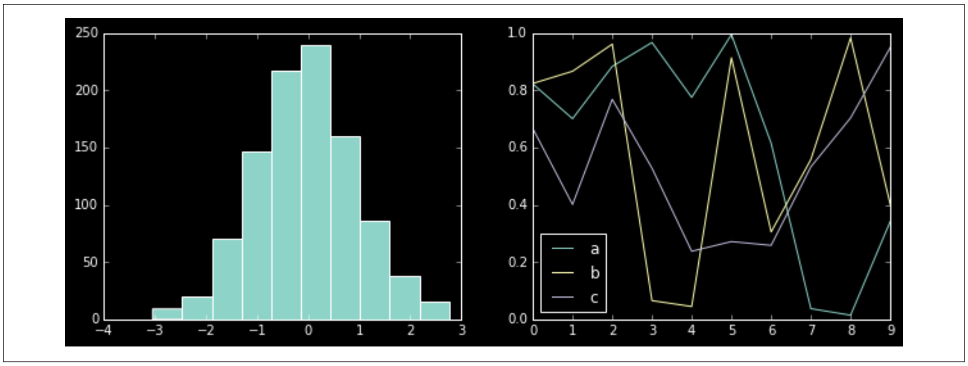

*Figure 4-89. The dark_background style*

### Grayscale

Sometimes you might find yourself preparing figures for a print publication that does not accept color figures. For this, the `grayscale` style, shown in Figure 4-90, can be very useful:

```
In[16]: with plt.style.context('grayscale'):
            hist_and_lines()
```

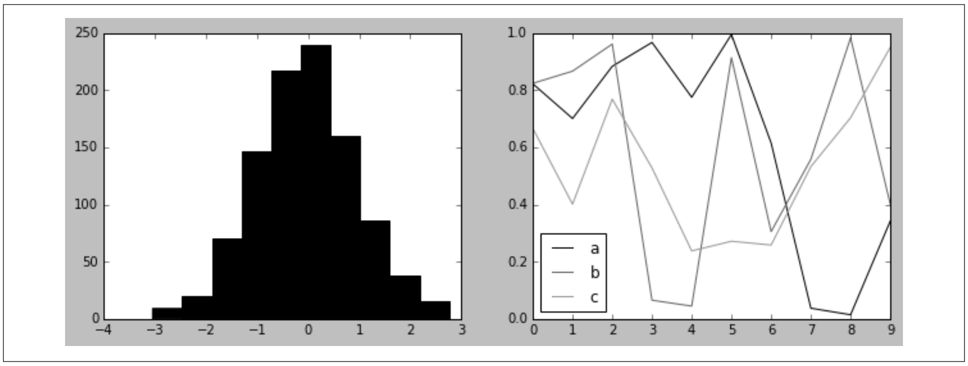

*Figure 4-90. The grayscale style*

### Seaborn style

Matplotlib also has stylesheets inspired by the Seaborn library (discussed more fully in "Visualization with Seaborn" on page 311). As we will see, these styles are loaded automatically when Seaborn is imported into a notebook. I've found these settings to be very nice, and tend to use them as defaults in my own data exploration (see Figure 4-91):

```
In[17]: import seaborn
        hist_and_lines()
```

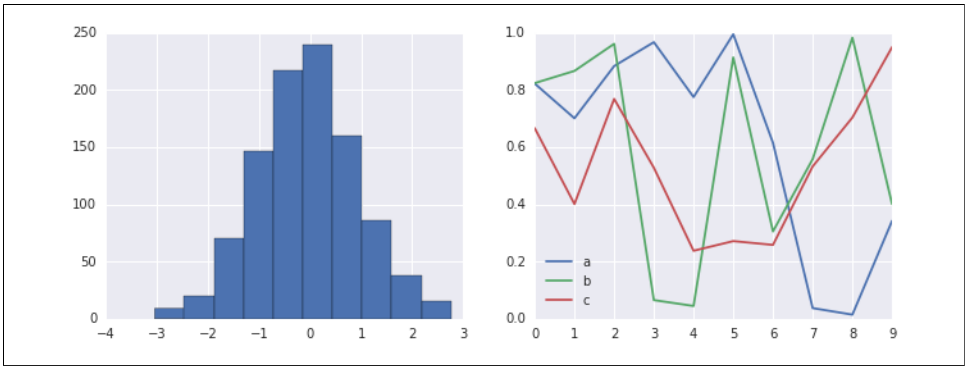

*Figure 4-91. Seaborn's plotting style*

With all of these built-in options for various plot styles, Matplotlib becomes much more useful for both interactive visualization and creation of figures for publication. Throughout this book, I will generally use one or more of these style conventions when creating plots.

## Three-Dimensional Plotting in Matplotlib

Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. We enable three-dimensional plots by importing the `mplot3d` toolkit, included with the main Matplotlib installation (Figure 4-92):

```
In[1]: from mpl_toolkits import mplot3d
```

Once this submodule is imported, we can create a three-dimensional axes by passing the keyword `projection='3d'` to any of the normal axes creation routines:

```
In[2]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
```

```
In[3]: fig = plt.figure()
       ax = plt.axes(projection='3d')
```