CHAPTER 4
Visualization with Matplotlib

We'll now take an in-depth look at the Matplotlib tool for visualization in Python. Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this as a cue to set out on his own, and the Matplotlib package was born, with version 0.1 released in 2003. It received an early boost when it was adopted as the plotting package of choice of the Space Telescope Science Institute (the folks behind the Hubble Telescope), which financially supported Matplotlib's development and greatly expanded its capabilities.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large userbase, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.

In recent years, however, the interface and style of Matplotlib have begun to show their age. Newer tools like ggplot and ggvis in the R language, along with web visualization toolkits based on D3js and HTML5 canvas, often make Matplotlib feel clunky and old-fashioned. Still, I'm of the opinion that we cannot ignore Matplotlib's strength as a well-tested, cross-platform graphics engine. Recent Matplotlib versions make it relatively easy to set new global plotting styles (see "Customizing Matplotlib: Configurations and Stylesheets" on page 282), and people have been developing new packages that build on its powerful internals to drive Matplotlib via cleaner, more

modern APIs—for example, Seaborn (discussed in "Visualization with Seaborn" on page 311), ggplot, HoloViews, Altair, and even Pandas itself can be used as wrappers around Matplotlib's API. Even with wrappers like these, it is still often useful to dive into Matplotlib's syntax to adjust the final plot output. For this reason, I believe that Matplotlib itself will remain a vital piece of the data visualization stack, even if new tools mean the community gradually moves away from using the Matplotlib API directly.

General Matplotlib Tips

Before we dive into the details of creating visualizations with Matplotlib, there are a few useful things you should know about using the package.

Importing matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

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

The `plt` interface is what we will use most often, as we'll see throughout this chapter.

Setting Styles

We will use the `plt.style` directive to choose appropriate aesthetic styles for our figures. Here we will set the `classic` style, which ensures that the plots we create use the classic Matplotlib style:

```
In[2]: plt.style.use('classic')
```

Throughout this section, we will adjust this style as needed. Note that the stylesheets used here are supported as of Matplotlib version 1.5; if you are using an earlier version of Matplotlib, only the default style is available. For more information on stylesheets, see "Customizing Matplotlib: Configurations and Stylesheets" on page 282.

show() or No show()? How to Display Your Plots

A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context. The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in an IPython notebook.

Plotting from a script

If you are using Matplotlib from within a script, the function `plt.show()` is your friend. `plt.show()` starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

So, for example, you may have a file called *myplot.py* containing the following:

```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

You can then run this script from the command-line prompt, which will result in a window opening with your figure displayed:

```
$ python myplot.py
```

The `plt.show()` command does a lot under the hood, as it must interact with your system's interactive graphical backend. The details of this operation can vary greatly from system to system and even installation to installation, but Matplotlib does its best to hide all these details from you.

One thing to be aware of: the `plt.show()` command should be used *only once* per Python session, and is most often seen at the very end of the script. Multiple `show()` commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Plotting from an IPython shell

It can be very convenient to use Matplotlib interactively within an IPython shell (see Chapter 1). IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the `%matplotlib` magic command after starting `ipython`:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

At this point, any `plt` plot command will cause a figure window to open, and further commands can be run to update the plot. Some changes (such as modifying properties of lines that are already drawn) will not draw automatically; to force an update, use `plt.draw()`. Using `plt.show()` in Matplotlib mode is not required.

Plotting from an IPython notebook

The IPython notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document (see Chapter 1).

Plotting interactively within an IPython notebook can be done with the `%matplotlib` command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

- `%matplotlib notebook` will lead to *interactive* plots embedded within the notebook
- `%matplotlib inline` will lead to *static* images of your plot embedded in the notebook

For this book, we will generally opt for `%matplotlib inline`:

```
In[3]: %matplotlib inline
```

After you run this command (it needs to be done only once per kernel/session), any cell within the notebook that creates a plot will embed a PNG image of the resulting graphic (Figure 4-1):

```
In[4]: import numpy as np
       x = np.linspace(0, 10, 100)

       fig = plt.figure()
       plt.plot(x, np.sin(x), '-')
       plt.plot(x, np.cos(x), '--');
```

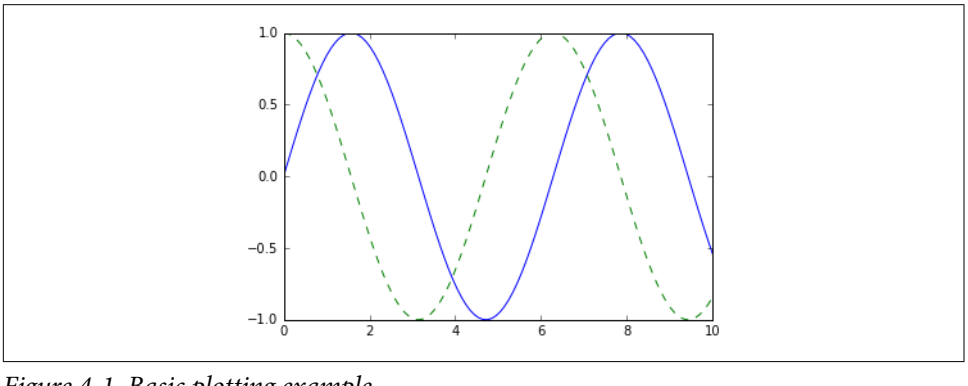

Figure 4-1. Basic plotting example

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. You can save a figure using the `savefig()` command. For example, to save the previous figure as a PNG file, you can run this:

```
In[5]: fig.savefig('my_figure.png')
```

We now have a file called *my_figure.png* in the current working directory:

```
In[6]: !ls -lh my_figure.png
```

```
-rw-r--r--  1 jakevdp  staff    16K Aug 11 10:59 my_figure.png
```

To confirm that it contains what we think it contains, let's use the IPython `Image` object to display the contents of this file (Figure 4-2):

```
In[7]: from IPython.display import Image
       Image('my_figure.png')
```

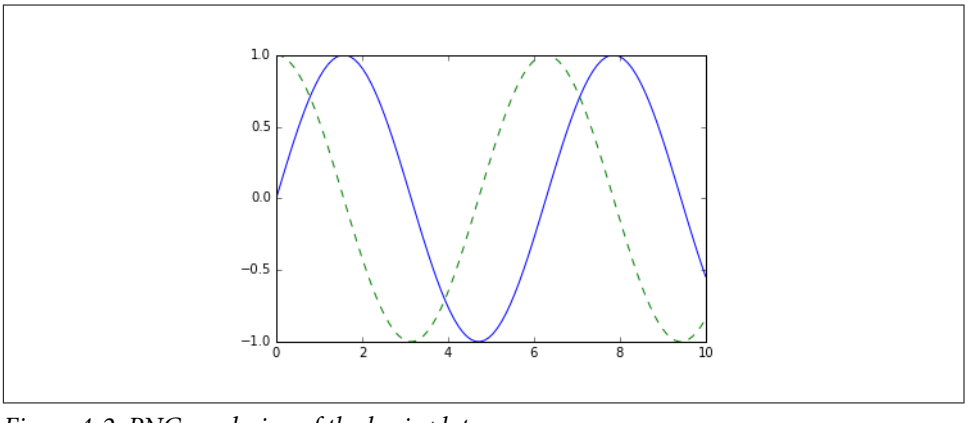

Figure 4-2. PNG rendering of the basic plot

In `savefig()`, the file format is inferred from the extension of the given filename. Depending on what backends you have installed, many different file formats are available. You can find the list of supported file types for your system by using the following method of the figure `canvas` object:

```
In[8]: fig.canvas.get_supported_filetypes()
```

```
Out[8]: {'eps': 'Encapsulated Postscript',
         'jpeg': 'Joint Photographic Experts Group',
         'jpg': 'Joint Photographic Experts Group',
         'pdf': 'Portable Document Format',
         'pgf': 'PGF code for LaTeX',
         'png': 'Portable Network Graphics',
         'ps': 'Postscript',
         'raw': 'Raw RGBA bitmap',
         'rgba': 'Raw RGBA bitmap',
```

```
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

Note that when saving your figure, it's not necessary to use `plt.show()` or related commands discussed earlier.

Two Interfaces for the Price of One

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface. We'll quickly highlight the differences between the two here.

MATLAB-style interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (`plt`) interface. For example, the following code will probably look quite familiar to MATLAB users (Figure 4-3):

```
In[9]: plt.figure()  # create a plot figure

       # create the first of two panels and set current axis
       plt.subplot(2, 1, 1) # (rows, columns, panel number)
       plt.plot(x, np.sin(x))

       # create the second panel and set current axis
       plt.subplot(2, 1, 2)
       plt.plot(x, np.cos(x));
```

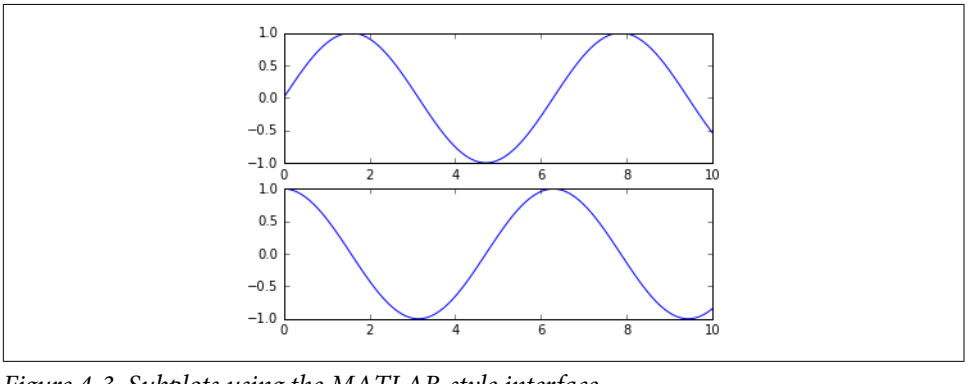

Figure 4-3. Subplots using the MATLAB-style interface

It's important to note that this interface is *stateful*: it keeps track of the "current" figure and axes, which are where all `plt` commands are applied. You can get a reference to

these using the `plt.gcf()` (get current figure) and `plt.gca()` (get current axes) routines.

While this stateful interface is fast and convenient for simple plots, it is easy to run into problems. For example, once the second panel is created, how can we go back and add something to the first? This is possible within the MATLAB-style interface, but a bit clunky. Fortunately, there is a better way.

Object-oriented interface

The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are *methods* of explicit `Figure` and `Axes` objects. To re-create the previous plot using this style of plotting, you might do the following (Figure 4-4):

```
In[10]: # First create a grid of plots
        # ax will be an array of two Axes objects
        fig, ax = plt.subplots(2)

        # Call plot() method on the appropriate object
        ax[0].plot(x, np.sin(x))
        ax[1].plot(x, np.cos(x));
```

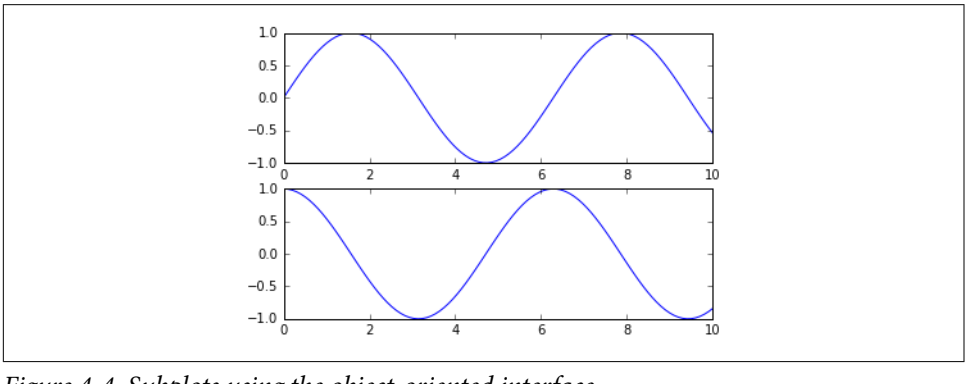

Figure 4-4. Subplots using the object-oriented interface

For more simple plots, the choice of which style to use is largely a matter of preference, but the object-oriented approach can become a necessity as plots become more complicated. Throughout this chapter, we will switch between the MATLAB-style and object-oriented interfaces, depending on what is most convenient. In most cases, the difference is as small as switching `plt.plot()` to `ax.plot()`, but there are a few gotchas that we will highlight as they come up in the following sections.

Simple Line Plots

Perhaps the simplest of all plots is the visualization of a single function $y = f(x)$. Here we will take a first look at creating a simple plot of this type. As with all the following sections, we'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np
```

For all Matplotlib plots, we start by creating a figure and an axes. In their simplest form, a figure and axes can be created as follows (Figure 4-5):

```
In[2]: fig = plt.figure()
       ax = plt.axes()
```

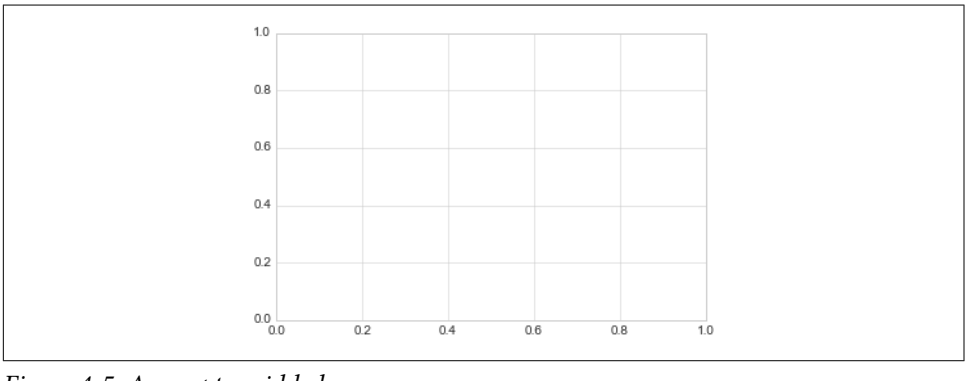

Figure 4-5. An empty gridded axes

In Matplotlib, the *figure* (an instance of the class `plt.Figure`) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The *axes* (an instance of the class `plt.Axes`) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization. Throughout this book, we'll commonly use the variable name `fig` to refer to a figure instance, and `ax` to refer to an axes instance or group of axes instances.

Once we have created an axes, we can use the `ax.plot` function to plot some data. Let's start with a simple sinusoid (Figure 4-6):

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

       x = np.linspace(0, 10, 1000)
       ax.plot(x, np.sin(x));
```

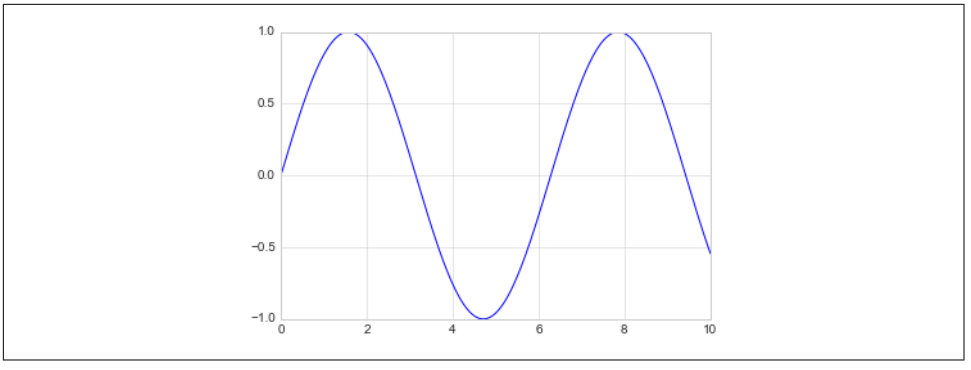

Figure 4-6. A simple sinusoid

Alternatively, we can use the pylab interface and let the figure and axes be created for us in the background (Figure 4-7; see "Two Interfaces for the Price of One" on page 222 for a discussion of these two interfaces):

```
In[4]: plt.plot(x, np.sin(x));
```

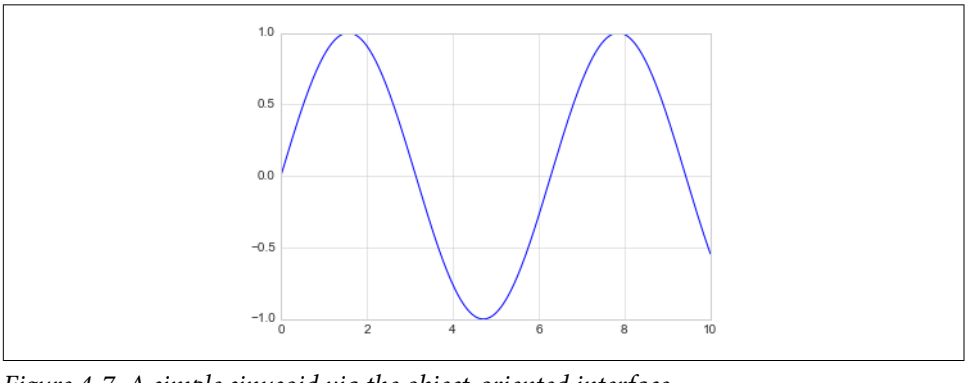

Figure 4-7. A simple sinusoid via the object-oriented interface

If we want to create a single figure with multiple lines, we can simply call the `plot` function multiple times (Figure 4-8):

```
In[5]: plt.plot(x, np.sin(x))
       plt.plot(x, np.cos(x));
```

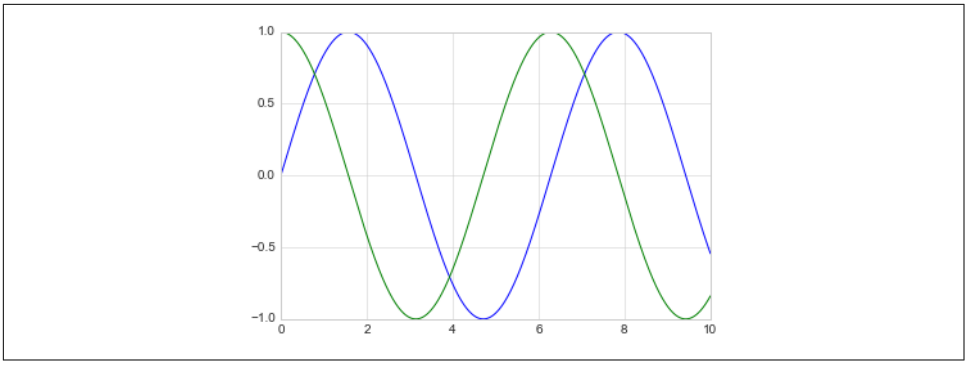

Figure 4-8. Over-plotting multiple lines

That's all there is to plotting simple functions in Matplotlib! We'll now dive into some more details about how to control the appearance of the axes and lines.

Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The `plt.plot()` function takes additional arguments that can be used to specify these. To adjust the color, you can use the `color` keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways (Figure 4-9):

```
In[6]:
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

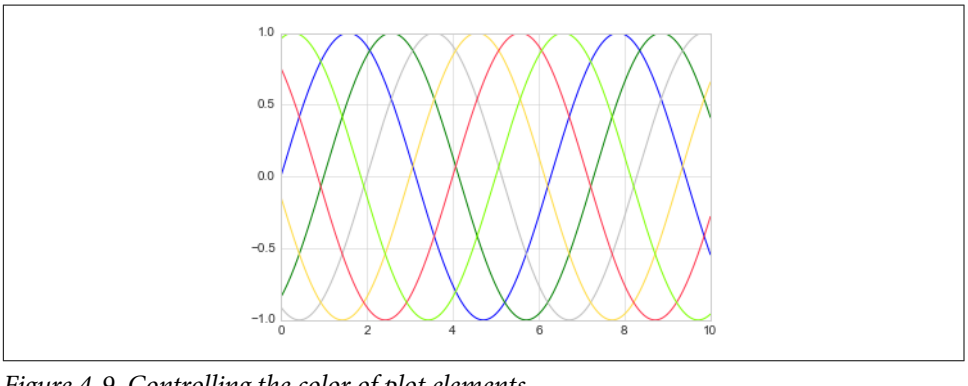

Figure 4-9. Controlling the color of plot elements

If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.

Similarly, you can adjust the line style using the `linestyle` keyword (Figure 4-10):

```
In[7]: plt.plot(x, x + 0, linestyle='solid')
       plt.plot(x, x + 1, linestyle='dashed')
       plt.plot(x, x + 2, linestyle='dashdot')
       plt.plot(x, x + 3, linestyle='dotted');

       # For short, you can use the following codes:
       plt.plot(x, x + 4, linestyle='-')  # solid
       plt.plot(x, x + 5, linestyle='--') # dashed
       plt.plot(x, x + 6, linestyle='-.') # dashdot
       plt.plot(x, x + 7, linestyle=':');  # dotted
```

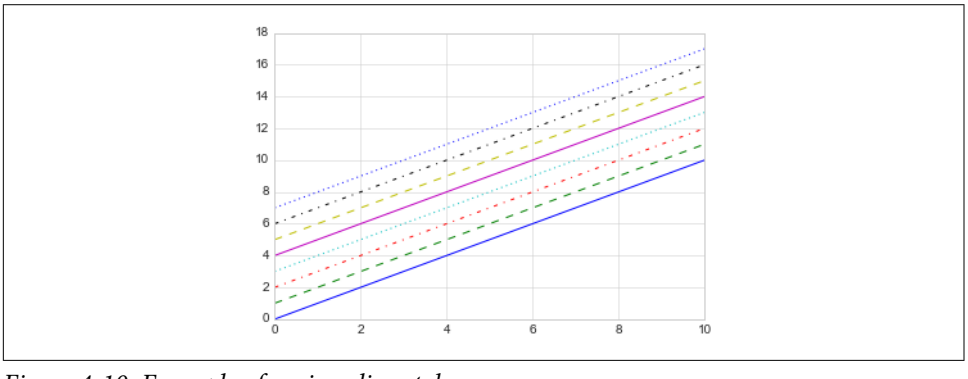

Figure 4-10. Example of various line styles

If you would like to be extremely terse, these `linestyle` and `color` codes can be combined into a single nonkeyword argument to the `plt.plot()` function (Figure 4-11):

```
In[8]: plt.plot(x, x + 0, '-g')  # solid green
       plt.plot(x, x + 1, '--c') # dashed cyan
       plt.plot(x, x + 2, '-.k') # dashdot black
       plt.plot(x, x + 3, ':r');  # dotted red
```

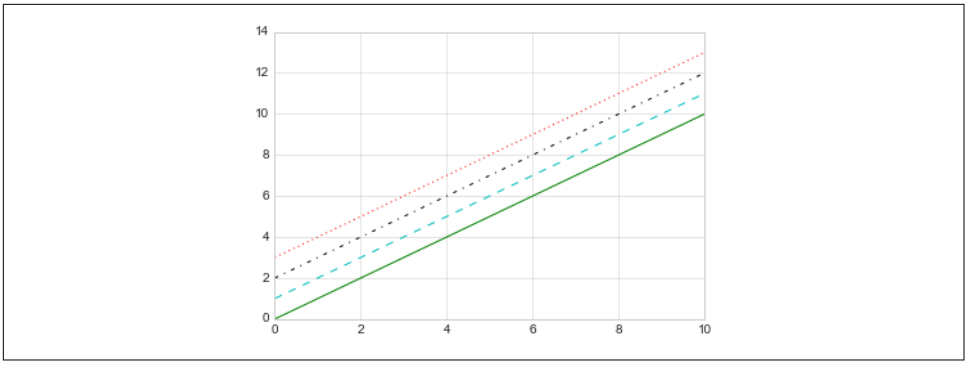

Figure 4-11. Controlling colors and styles with the shorthand syntax

These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/blacK) color systems, commonly used for digital color graphics.

There are many other keyword arguments that can be used to fine-tune the appearance of the plot; for more details, I'd suggest viewing the docstring of the `plt.plot()` function using IPython's help tools (see "Help and Documentation in IPython" on page 3).

Adjusting the Plot: Axes Limits

Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the `plt.xlim()` and `plt.ylim()` methods (Figure 4-12):

```
In[9]: plt.plot(x, np.sin(x))

       plt.xlim(-1, 11)
       plt.ylim(-1.5, 1.5);
```

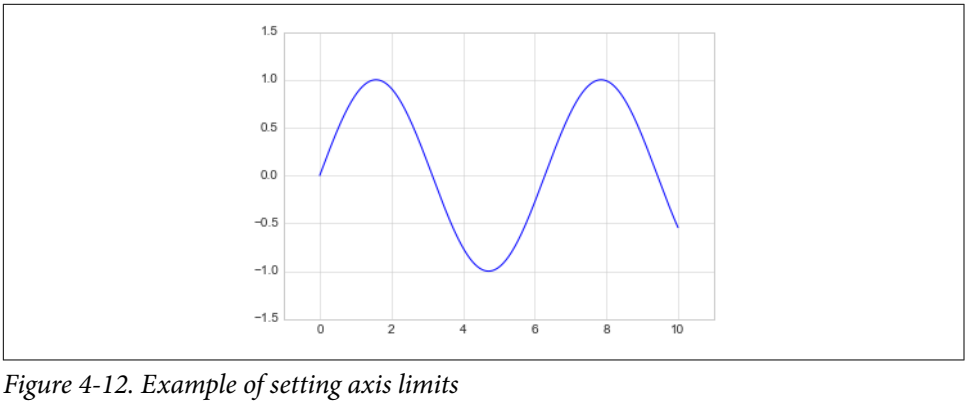

Figure 4-12. Example of setting axis limits

If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments (Figure 4-13):

```
In[10]: plt.plot(x, np.sin(x))

        plt.xlim(10, 0)
        plt.ylim(1.2, -1.2);
```

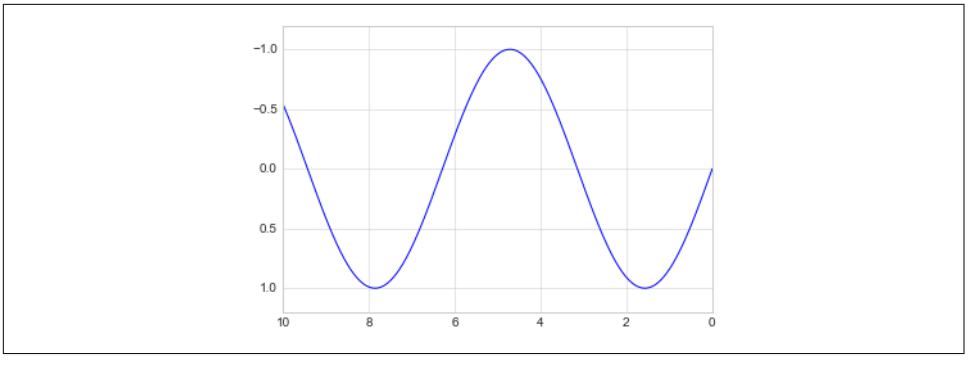

Figure 4-13. Example of reversing the y-axis

A useful related method is `plt.axis()` (note here the potential confusion between *axes* with an *e*, and *axis* with an *i*). The `plt.axis()` method allows you to set the `x` and `y` limits with a single call, by passing a list that specifies `[xmin, xmax, ymin, ymax]` (Figure 4-14):

```
In[11]: plt.plot(x, np.sin(x))
        plt.axis([-1, 11, -1.5, 1.5]);
```

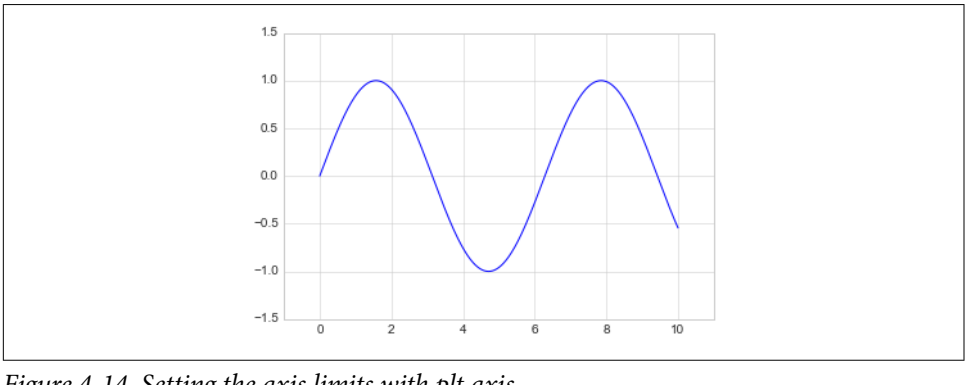

Figure 4-14. Setting the axis limits with plt.axis

The `plt.axis()` method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot (Figure 4-15):

```
In[12]: plt.plot(x, np.sin(x))
        plt.axis('tight');
```

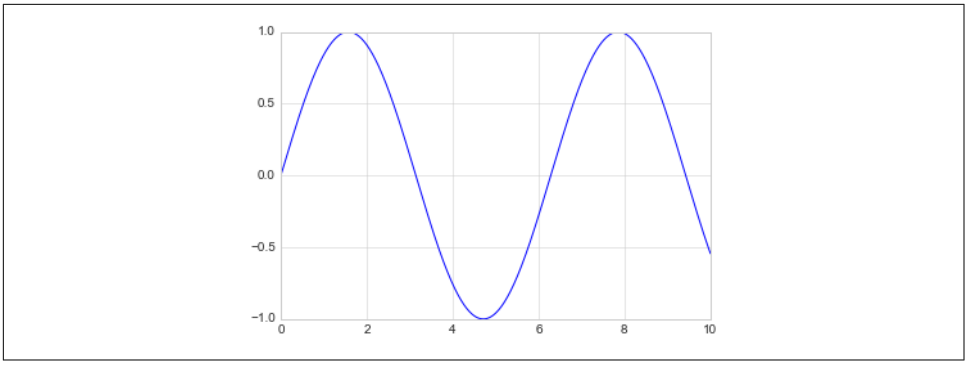

Figure 4-15. Example of a "tight" layout

It allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in `x` is equal to one unit in `y` (Figure 4-16):

```
In[13]: plt.plot(x, np.sin(x))
        plt.axis('equal');
```

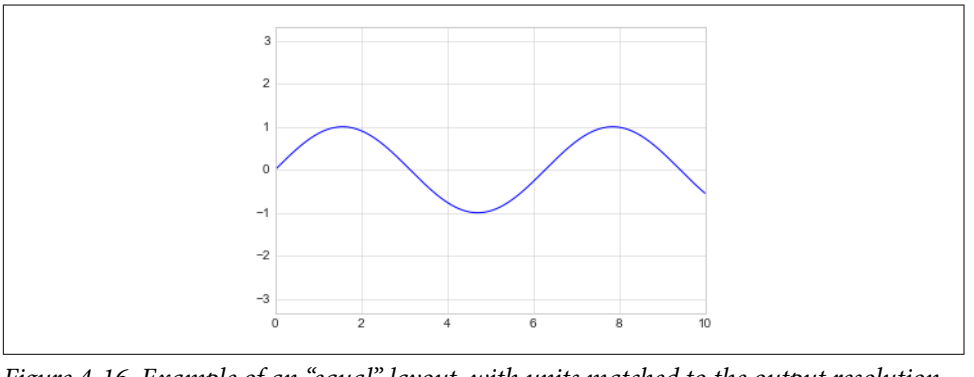

Figure 4-16. Example of an "equal" layout, with units matched to the output resolution

For more information on axis limits and the other capabilities of the `plt.axis()` method, refer to the `plt.axis()` docstring.

Labeling Plots

As the last piece of this section, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them (Figure 4-17):

```
In[14]: plt.plot(x, np.sin(x))
        plt.title("A Sine Curve")
```

```
plt.xlabel("x")
plt.ylabel("sin(x)");
```

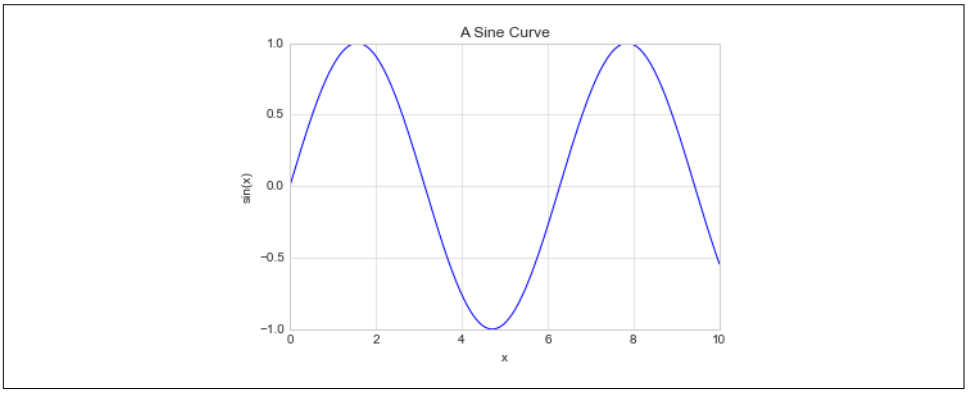

Figure 4-17. Examples of axis labels and title

You can adjust the position, size, and style of these labels using optional arguments to the function. For more information, see the Matplotlib documentation and the docstrings of each of these functions.

When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via the (you guessed it) `plt.legend()` method. Though there are several valid ways of using this, I find it easiest to specify the label of each line using the `label` keyword of the plot function (Figure 4-18):

```
In[15]: plt.plot(x, np.sin(x), '-g', label='sin(x)')
        plt.plot(x, np.cos(x), ':b', label='cos(x)')
        plt.axis('equal')

        plt.legend();
```

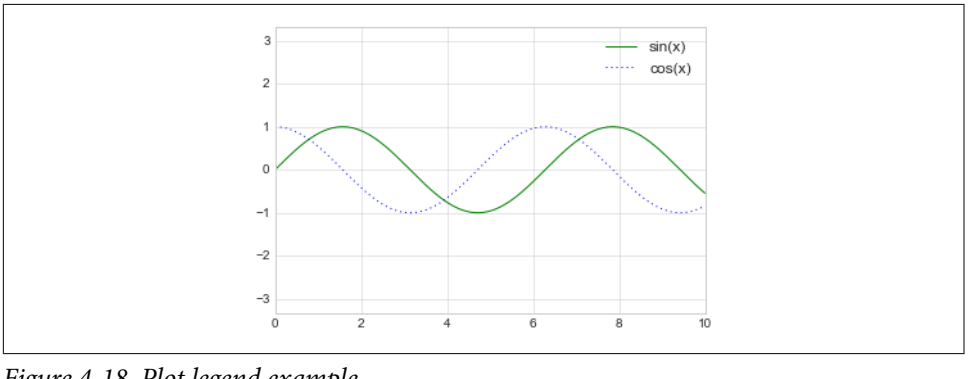

Figure 4-18. Plot legend example

As you can see, the `plt.legend()` function keeps track of the line style and color, and matches these with the correct label. More information on specifying and formatting plot legends can be found in the `plt.legend()` docstring; additionally, we will cover some more advanced legend options in "Customizing Plot Legends" on page 249.

Matplotlib Gotchas

While most `plt` functions translate directly to `ax` methods (such as `plt.plot()` → `ax.plot()`, `plt.legend()` → `ax.legend()`, etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

- `plt.xlabel()` → `ax.set_xlabel()`
- `plt.ylabel()` → `ax.set_ylabel()`
- `plt.xlim()` → `ax.set_xlim()`
- `plt.ylim()` → `ax.set_ylim()`
- `plt.title()` → `ax.set_title()`

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the `ax.set()` method to set all these properties at once (Figure 4-19):

```
In[16]: ax = plt.axes()
        ax.plot(x, np.sin(x))
        ax.set(xlim=(0, 10), ylim=(-2, 2),
               xlabel='x', ylabel='sin(x)',
               title='A Simple Plot');
```

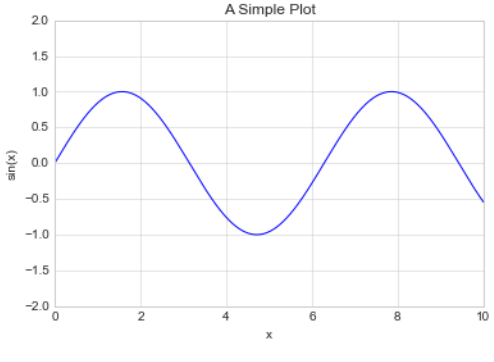

Figure 4-19. Example of using ax.set to set multiple properties at once

Simple Scatter Plots

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape. We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np
```

Scatter Plots with plt.plot

In the previous section, we looked at `plt.plot`/`ax.plot` to produce line plots. It turns out that this same function can produce scatter plots as well (Figure 4-20):

```
In[2]: x = np.linspace(0, 10, 30)
       y = np.sin(x)

       plt.plot(x, y, 'o', color='black');
```

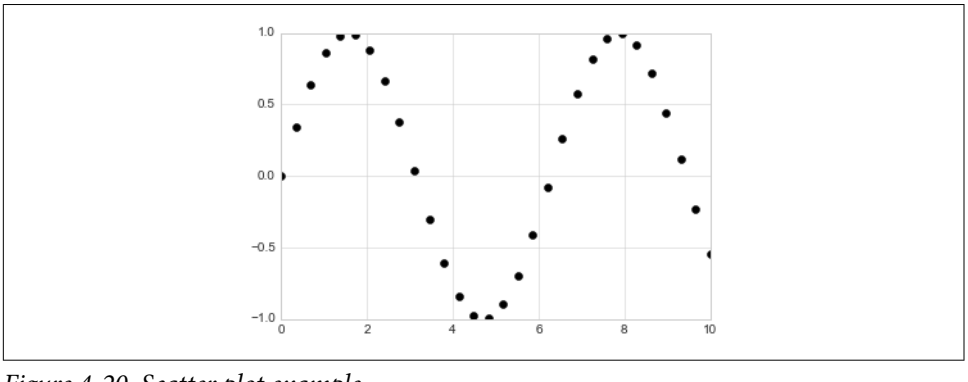

Figure 4-20. Scatter plot example

The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as `'-'` and `'--'` to control the line style, the marker style has its own set of short string codes. The full list of available symbols can be seen in the documentation of `plt.plot`, or in Matplotlib's online documentation. Most of the possibilities are fairly intuitive, and we'll show a number of the more common ones here (Figure 4-21):

```
In[3]: rng = np.random.RandomState(0)
       for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
           plt.plot(rng.rand(5), rng.rand(5), marker,
                    label="marker='{0}'".format(marker))
```

```
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

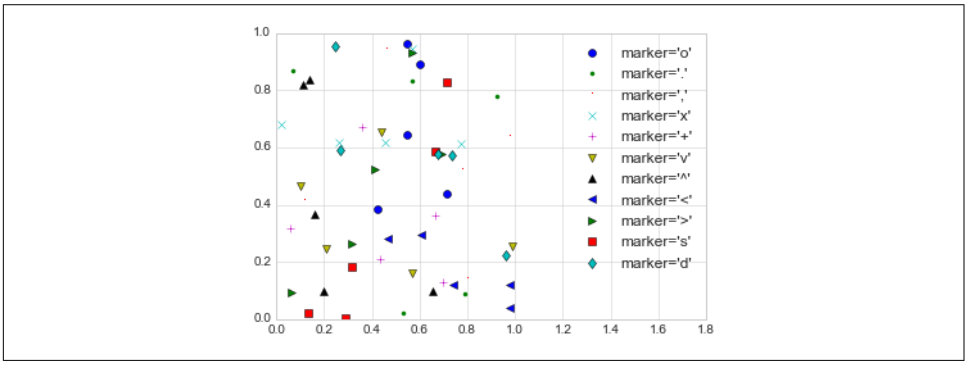

Figure 4-21. Demonstration of point numbers

For even more possibilities, these character codes can be used together with line and color codes to plot points along with a line connecting them (Figure 4-22):

```
In[4]: plt.plot(x, y, '-ok');   # line (-), circle marker (o), black (k)
```

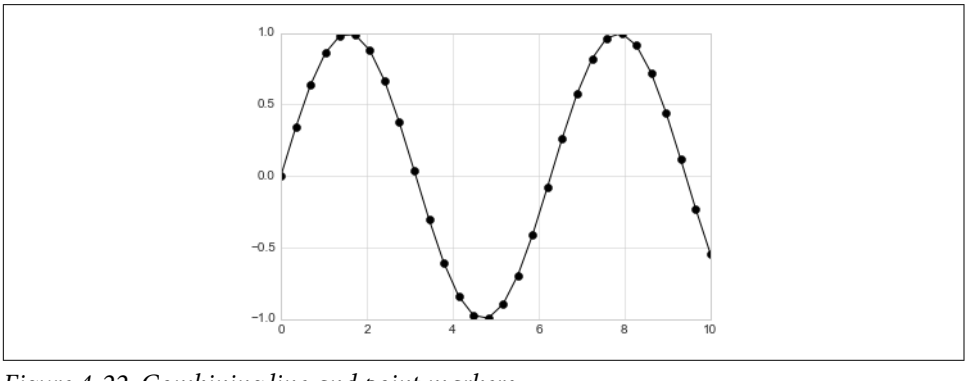

Figure 4-22. Combining line and point markers

Additional keyword arguments to `plt.plot` specify a wide range of properties of the lines and markers (Figure 4-23):

```
In[5]: plt.plot(x, y, '-p', color='gray',
                markersize=15, linewidth=4,
                markerfacecolor='white',
                markeredgecolor='gray',
                markeredgewidth=2)
       plt.ylim(-1.2, 1.2);
```

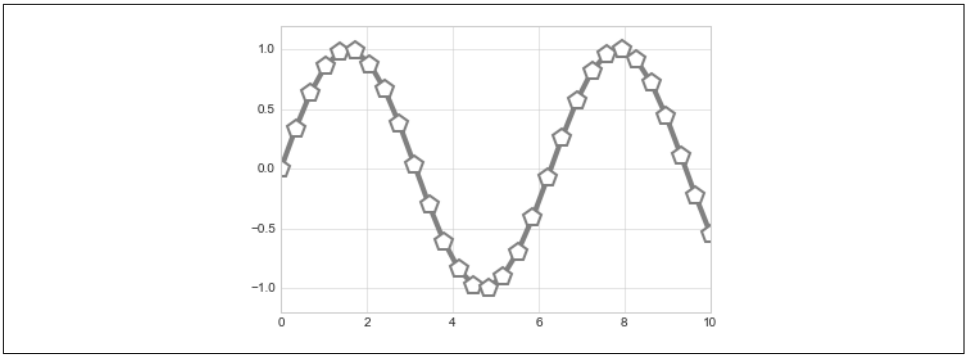

Figure 4-23. Customizing line and point numbers

This type of flexibility in the `plt.plot` function allows for a wide variety of possible visualization options. For a full description of the options available, refer to the `plt.plot` documentation.

Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the `plt.scatter` function, which can be used very similarly to the `plt.plot` function (Figure 4-24):

```
In[6]: plt.scatter(x, y, marker='o');
```

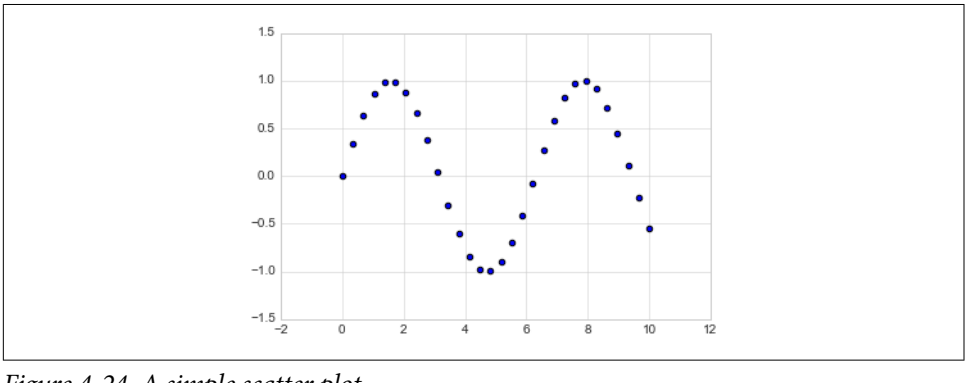

Figure 4-24. A simple scatter plot

The primary difference of `plt.scatter` from `plt.plot` is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the `alpha` keyword to adjust the transparency level (Figure 4-25):

```
In[7]: rng = np.random.RandomState(0)
       x = rng.randn(100)
       y = rng.randn(100)
       colors = rng.rand(100)
       sizes = 1000 * rng.rand(100)

       plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
                   cmap='viridis')
       plt.colorbar();  # show color scale
```

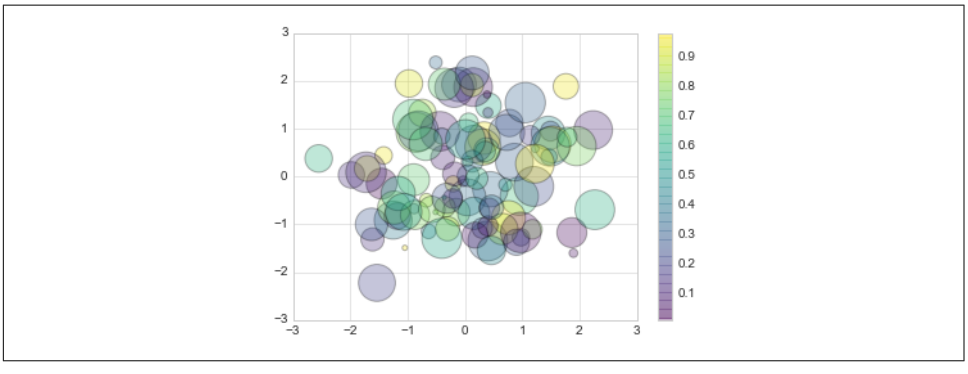

Figure 4-25. Changing size, color, and transparency in scatter points

Notice that the color argument is automatically mapped to a color scale (shown here by the `colorbar()` command), and the size argument is given in pixels. In this way, the color and size of points can be used to convey information in the visualization, in order to illustrate multidimensional data.

For example, we might use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully measured (Figure 4-26):

```
In[8]: from sklearn.datasets import load_iris
       iris = load_iris()
       features = iris.data.T

       plt.scatter(features[0], features[1], alpha=0.2,
                   s=100*features[3], c=iris.target, cmap='viridis')
       plt.xlabel(iris.feature_names[0])
       plt.ylabel(iris.feature_names[1]);
```

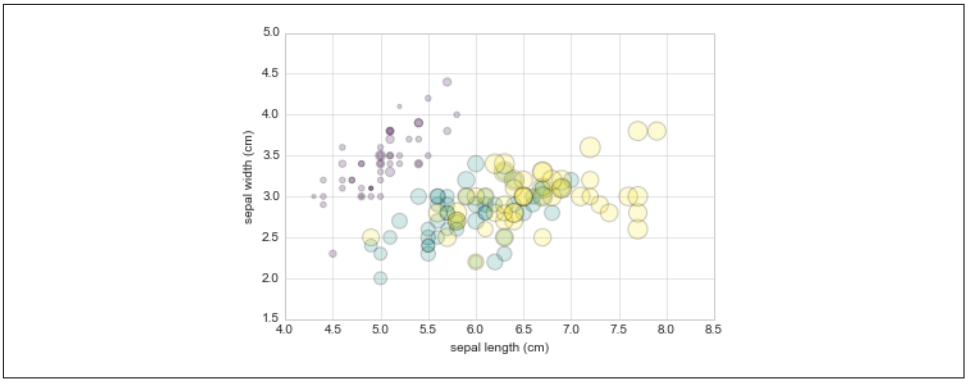

Figure 4-26. Using point properties to encode features of the Iris data

We can see that this scatter plot has given us the ability to simultaneously explore four different dimensions of the data: the (x, y) location of each point corresponds to the sepal length and width, the size of the point is related to the petal width, and the color is related to the particular species of flower. Multicolor and multifeature scatter plots like this can be useful for both exploration and presentation of data.

plot Versus scatter: A Note on Efficiency

Aside from the different features available in `plt.plot` and `plt.scatter`, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, `plt.plot` can be noticeably more efficient than `plt.scatter`. The reason is that `plt.scatter` has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In `plt.plot`, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, `plt.plot` should be preferred over `plt.scatter` for large datasets.

Visualizing Errors

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself. For example, imagine that I am using some astrophysical observations to estimate the Hubble Constant, the local measurement of the expansion rate of the universe. I know that the current literature suggests a value of around 71 (km/s)/Mpc, and I measure a value of 74 (km/s)/Mpc with my method. Are the values consistent? The only correct answer, given this information, is this: there is no way to know.

Suppose I augment this information with reported uncertainties: the current literature suggests a value of around 71 ± 2.5 (km/s)/Mpc, and my method has measured a value of 74 ± 5 (km/s)/Mpc. Now are the values consistent? That is a question that can be quantitatively answered.

In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

Basic Errorbars

A basic errorbar can be created with a single Matplotlib function call (Figure 4-27):

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np

In[2]: x = np.linspace(0, 10, 50)
       dy = 0.8
       y = np.sin(x) + dy * np.random.randn(50)

       plt.errorbar(x, y, yerr=dy, fmt='.k');
```

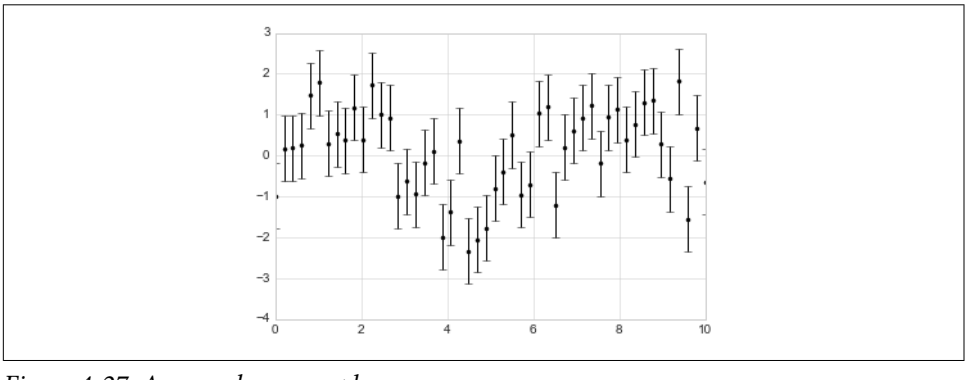

Figure 4-27. An errorbar example

Here the `fmt` is a format code controlling the appearance of lines and points, and has the same syntax as the shorthand used in `plt.plot`, outlined in "Simple Line Plots" on page 224 and "Simple Scatter Plots" on page 233.

In addition to these basic options, the `errorbar` function has many options to fine-tune the outputs. Using these additional options you can easily customize the aesthetics of your errorbar plot. I often find it helpful, especially in crowded plots, to make the errorbars lighter than the points themselves (Figure 4-28):

```
In[3]: plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
                    ecolor='lightgray', elinewidth=3, capsize=0);
```

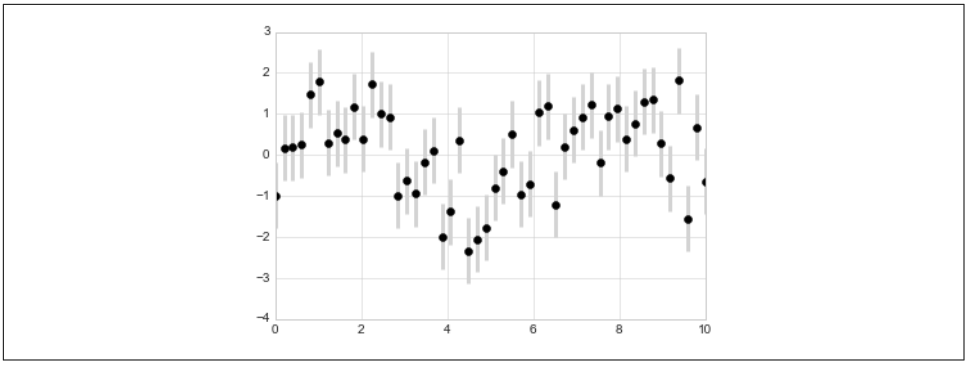

Figure 4-28. Customizing errorbars

In addition to these options, you can also specify horizontal errorbars (`xerr`), one-sided errorbars, and many other variants. For more information on the options available, refer to the docstring of `plt.errorbar`.

Continuous Errors

In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like `plt.plot` and `plt.fill_between` for a useful result.

Here we'll perform a simple *Gaussian process regression* (GPR), using the Scikit-Learn API (see "Introducing Scikit-Learn" on page 343 for details). This is a method of fitting a very flexible nonparametric function to data with a continuous measure of the uncertainty. We won't delve into the details of Gaussian process regression at this point, but will focus instead on how you might visualize such a continuous error measurement:

```
In[4]: from sklearn.gaussian_process import GaussianProcess

       # define the model and draw some data
       model = lambda x: x * np.sin(x)
       xdata = np.array([1, 3, 5, 6, 8])
       ydata = model(xdata)

       # Compute the Gaussian process fit
       gp = GaussianProcess(corr='cubic', theta0=1e-2, thetaL=1e-4, thetaU=1E-1,
                            random_start=100)
       gp.fit(xdata[:, np.newaxis], ydata)

       xfit = np.linspace(0, 10, 1000)
       yfit, MSE = gp.predict(xfit[:, np.newaxis], eval_MSE=True)
       dyfit = 2 * np.sqrt(MSE)  # 2*sigma ~ 95% confidence region
```

We now have `xfit`, `yfit`, and `dyfit`, which sample the continuous fit to our data. We could pass these to the `plt.errorbar` function as above, but we don't really want to plot 1,000 points with 1,000 errorbars. Instead, we can use the `plt.fill_between` function with a light color to visualize this continuous error (Figure 4-29):

```
In[5]: # Visualize the result
       plt.plot(xdata, ydata, 'or')
       plt.plot(xfit, yfit, '-', color='gray')

       plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                        color='gray', alpha=0.2)
       plt.xlim(0, 10);
```

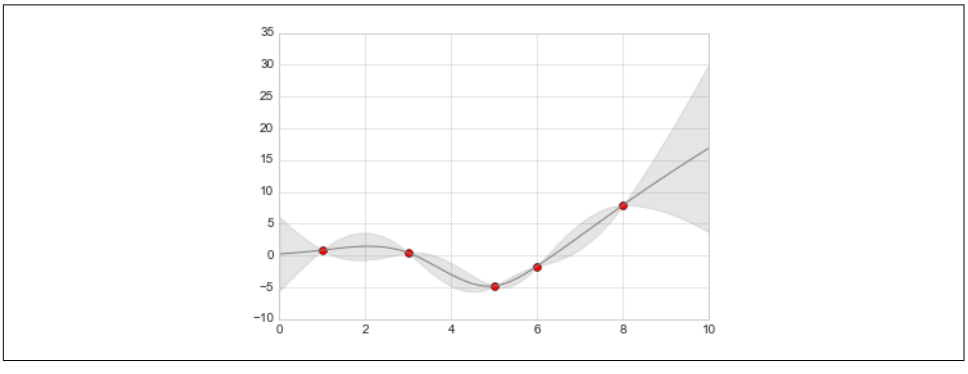

Figure 4-29. Representing continuous uncertainty with filled regions

Note what we've done here with the `fill_between` function: we pass an x value, then the lower y-bound, then the upper y-bound, and the result is that the area between these regions is filled.

The resulting figure gives a very intuitive view into what the Gaussian process regression algorithm is doing: in regions near a measured data point, the model is strongly constrained and this is reflected in the small model errors. In regions far from a measured data point, the model is not strongly constrained, and the model errors increase.

For more information on the options available in `plt.fill_between()` (and the closely related `plt.fill()` function), see the function docstring or the Matplotlib documentation.

Finally, if this seems a bit too low level for your taste, refer to "Visualization with Seaborn" on page 311, where we discuss the Seaborn package, which has a more streamlined API for visualizing this type of continuous errorbar.

Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: `plt.contour` for contour plots, `plt.contourf` for filled contour plots, and `plt.imshow` for showing images. This section looks at several examples of using these. We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')
       import numpy as np
```

Visualizing a Three-Dimensional Function

We'll start by demonstrating a contour plot using a function $z = f(x, y)$, using the following particular choice for f (we've seen this before in "Computation on Arrays: Broadcasting" on page 63, when we used it as a motivating example for array broadcasting):

```
In[2]: def f(x, y):
           return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

A contour plot can be created with the `plt.contour` function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the `np.meshgrid` function, which builds two-dimensional grids from one-dimensional arrays:

```
In[3]: x = np.linspace(0, 5, 50)
       y = np.linspace(0, 5, 40)

       X, Y = np.meshgrid(x, y)
       Z = f(X, Y)
```

Now let's look at this with a standard line-only contour plot (Figure 4-30):

```
In[4]: plt.contour(X, Y, Z, colors='black');
```

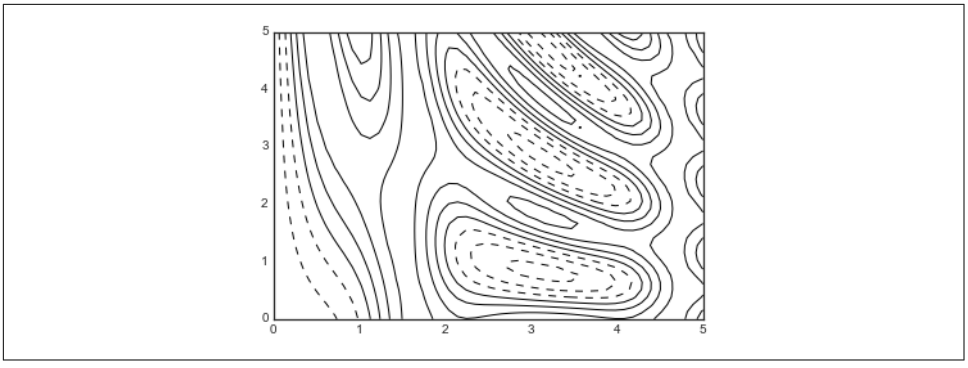

Figure 4-30. Visualizing three-dimensional data with contours

Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines. Alternatively, you can color-code the lines by specifying a colormap with the `cmap` argument. Here, we'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range (Figure 4-31):

```
In[5]: plt.contour(X, Y, Z, 20, cmap='RdGy');
```

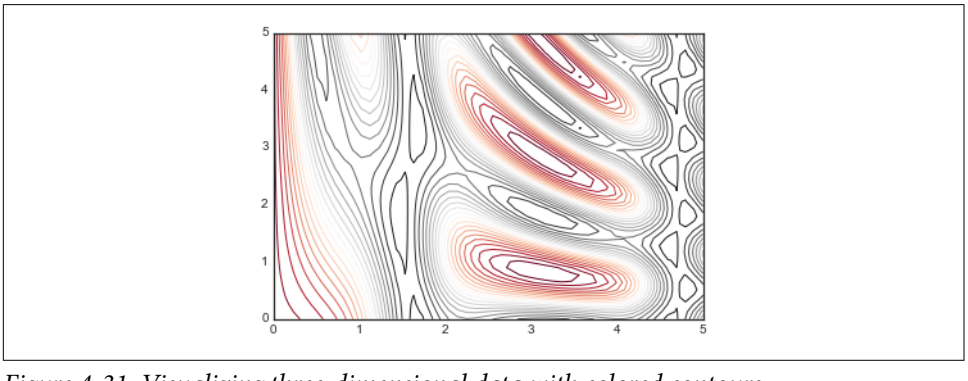

Figure 4-31. Visualizing three-dimensional data with colored contours

Here we chose the `RdGy` (short for *Red-Gray*) colormap, which is a good choice for centered data. Matplotlib has a wide range of colormaps available, which you can easily browse in IPython by doing a tab completion on the `plt.cm` module:

```
plt.cm.<TAB>
```

Our plot is looking nicer, but the spaces between the lines may be a bit distracting. We can change this by switching to a filled contour plot using the `plt.contourf()` function (notice the `f` at the end), which uses largely the same syntax as `plt.contour()`.

Additionally, we'll add a `plt.colorbar()` command, which automatically creates an additional axis with labeled color information for the plot (Figure 4-32):

```
In[6]: plt.contourf(X, Y, Z, 20, cmap='RdGy')
       plt.colorbar();
```

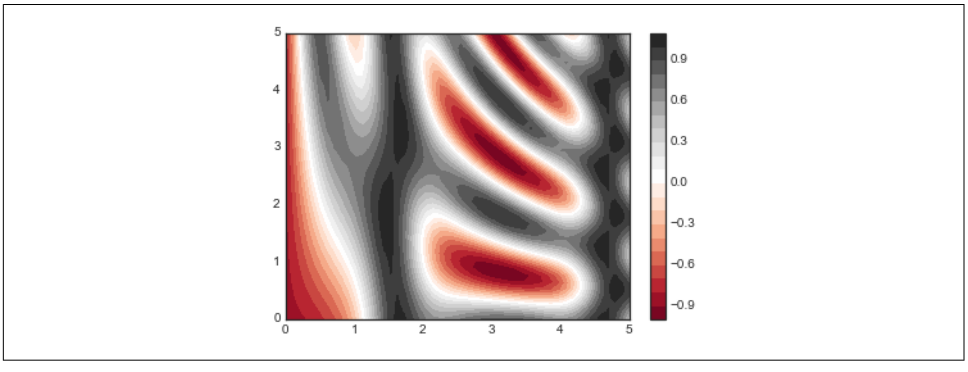

Figure 4-32. Visualizing three-dimensional data with filled contours

The colorbar makes it clear that the black regions are "peaks," while the red regions are "valleys."

One potential issue with this plot is that it is a bit "splotchy." That is, the color steps are discrete rather than continuous, which is not always what is desired. You could remedy this by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level. A better way to handle this is to use the `plt.imshow()` function, which interprets a two-dimensional grid of data as an image.

Figure 4-33 shows the result of the following code:

```
In[7]: plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
                  cmap='RdGy')
       plt.colorbar()
       plt.axis(aspect='image');
```

There are a few potential gotchas with `imshow()`, however:

- `plt.imshow()` doesn't accept an *x* and *y* grid, so you must manually specify the *extent* [*xmin*, *xmax*, *ymin*, *ymax*] of the image on the plot.
- `plt.imshow()` by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.

- `plt.imshow()` will automatically adjust the axis aspect ratio to match the input data; you can change this by setting, for example, `plt.axis(aspect='image')` to make *x* and *y* units match.

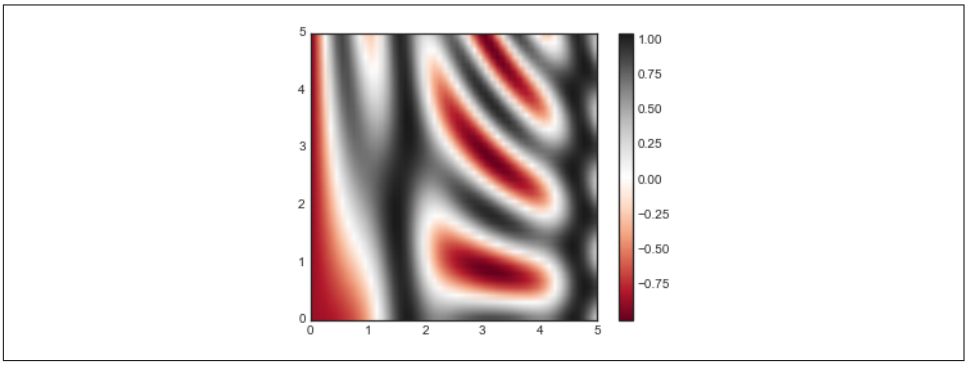

Figure 4-33. Representing three-dimensional data as an image

Finally, it can sometimes be useful to combine contour plots and image plots. For example, to create the effect shown in Figure 4-34, we'll use a partially transparent background image (with transparency set via the `alpha` parameter) and over-plot contours with labels on the contours themselves (using the `plt.clabel()` function):

```
In[8]: contours = plt.contour(X, Y, Z, 3, colors='black')
       plt.clabel(contours, inline=True, fontsize=8)

       plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
                  cmap='RdGy', alpha=0.5)
       plt.colorbar();
```

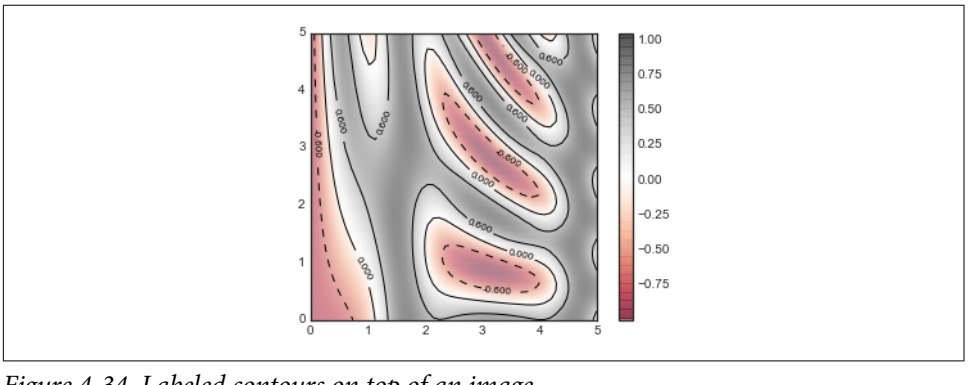

Figure 4-34. Labeled contours on top of an image

The combination of these three functions—`plt.contour`, `plt.contourf`, and `plt.imshow`—gives nearly limitless possibilities for displaying this sort of three-dimensional data within a two-dimensional plot. For more information on the

options available in these functions, refer to their docstrings. If you are interested in three-dimensional visualizations of this type of data, see "Three-Dimensional Plotting in Matplotlib" on page 290.

Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function (see "Comparisons, Masks, and Boolean Logic" on page 70), which creates a basic histogram in one line, once the normal boilerplate imports are done (Figure 4-35):

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')

       data = np.random.randn(1000)

In[2]: plt.hist(data);
```

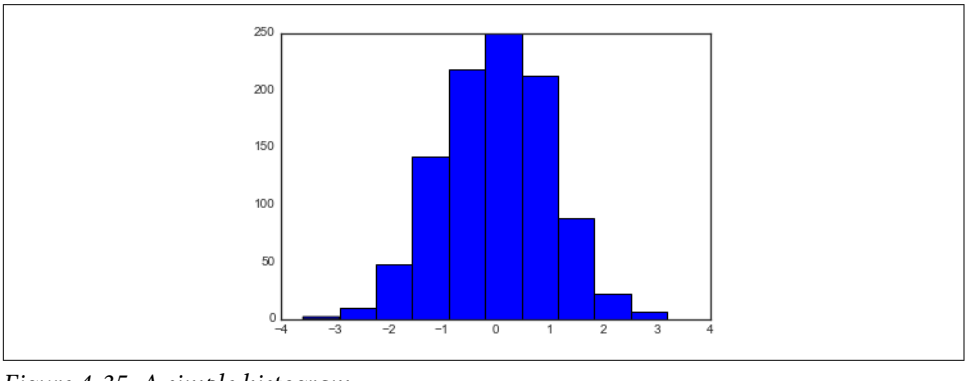

Figure 4-35. A simple histogram

The `hist()` function has many options to tune both the calculation and the display; here's an example of a more customized histogram (Figure 4-36):

```
In[3]: plt.hist(data, bins=30, normed=True, alpha=0.5,
                histtype='stepfilled', color='steelblue',
                edgecolor='none');
```

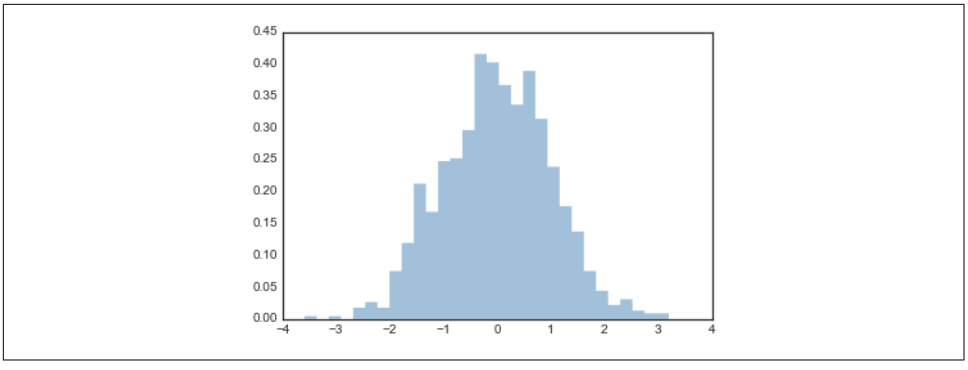

Figure 4-36. A customized histogram

The `plt.hist` docstring has more information on other customization options available. I find this combination of `histtype='stepfilled'` along with some transparency `alpha` to be very useful when comparing histograms of several distributions (Figure 4-37):

```
In[4]: x1 = np.random.normal(0, 0.8, 1000)
       x2 = np.random.normal(-2, 1, 1000)
       x3 = np.random.normal(3, 2, 1000)

       kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)

       plt.hist(x1, **kwargs)
       plt.hist(x2, **kwargs)
       plt.hist(x3, **kwargs);
```

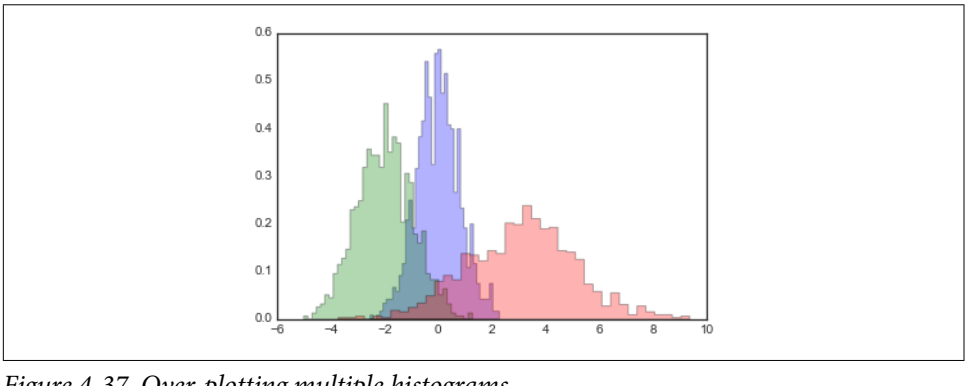

Figure 4-37. Over-plotting multiple histograms

If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the `np.histogram()` function is available:

```
In[5]: counts, bin_edges = np.histogram(data, bins=5)
       print(counts)
```

```
[ 12 190 468 301  29]
```

Two-Dimensional Histograms and Binnings

Just as we create histograms in one dimension by dividing the number line into bins, we can also create histograms in two dimensions by dividing points among two-dimensional bins. We'll take a brief look at several ways to do this here. We'll start by defining some data—an `x` and `y` array drawn from a multivariate Gaussian distribution:

```
In[6]: mean = [0, 0]
       cov = [[1, 1], [1, 2]]
       x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

plt.hist2d: Two-dimensional histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's `plt.hist2d` function (Figure 4-38):

```
In[12]: plt.hist2d(x, y, bins=30, cmap='Blues')
        cb = plt.colorbar()
        cb.set_label('counts in bin')
```

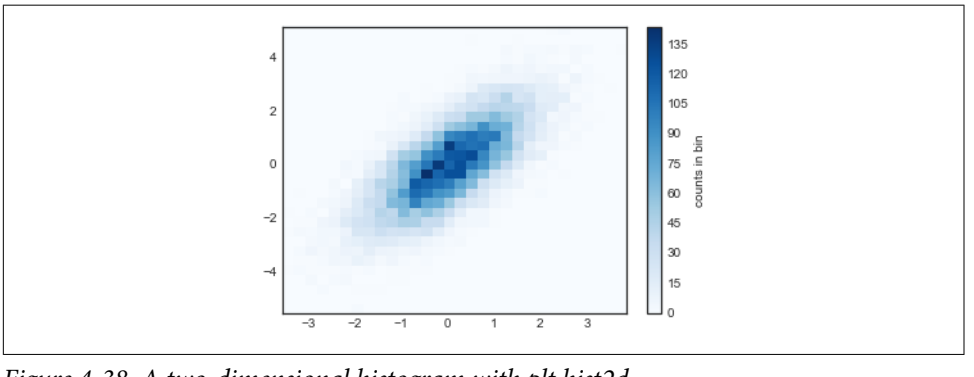

Figure 4-38. A two-dimensional histogram with plt.hist2d

Just as with `plt.hist`, `plt.hist2d` has a number of extra options to fine-tune the plot and the binning, which are nicely outlined in the function docstring. Further, just as `plt.hist` has a counterpart in `np.histogram`, `plt.hist2d` has a counterpart in `np.histogram2d`, which can be used as follows:

```
In[8]: counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

For the generalization of this histogram binning in dimensions higher than two, see the `np.histogramdd` function.

plt.hexbin: Hexagonal binnings

The two-dimensional histogram creates a tessellation of squares across the axes. Another natural shape for such a tessellation is the regular hexagon. For this purpose, Matplotlib provides the `plt.hexbin` routine, which represents a two-dimensional dataset binned within a grid of hexagons (Figure 4-39):

```
In[9]: plt.hexbin(x, y, gridsize=30, cmap='Blues')
       cb = plt.colorbar(label='count in bin')
```

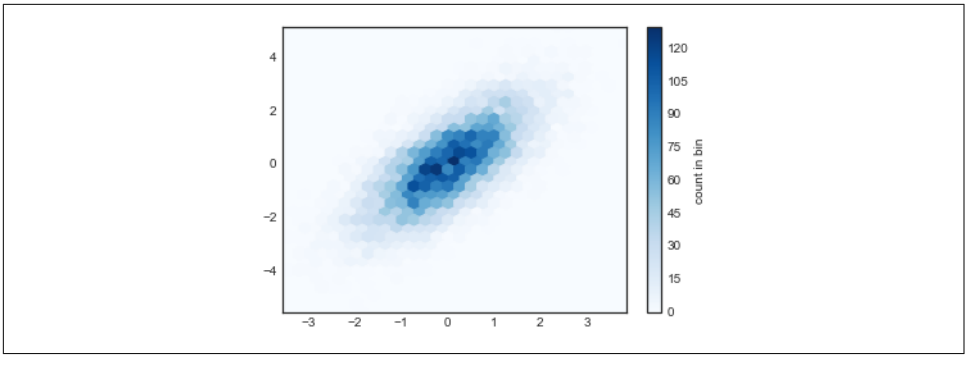

Figure 4-39. A two-dimensional histogram with plt.hexbin

`plt.hexbin` has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Kernel density estimation

Another common method of evaluating densities in multiple dimensions is *kernel density estimation* (KDE). This will be discussed more fully in "In-Depth: Kernel Density Estimation" on page 491, but for now we'll simply mention that KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function. One extremely quick and simple KDE implementation exists in the `scipy.stats` package. Here is a quick example of using the KDE on this data (Figure 4-40):

```
In[10]: from scipy.stats import gaussian_kde

        # fit an array of size [Ndim, Nsamples]
        data = np.vstack([x, y])
        kde = gaussian_kde(data)

        # evaluate on a regular grid
        xgrid = np.linspace(-3.5, 3.5, 40)
        ygrid = np.linspace(-6, 6, 40)
        Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
        Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))
```

```
# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),
           origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6],
           cmap='Blues')
cb = plt.colorbar()
cb.set_label("density")
```

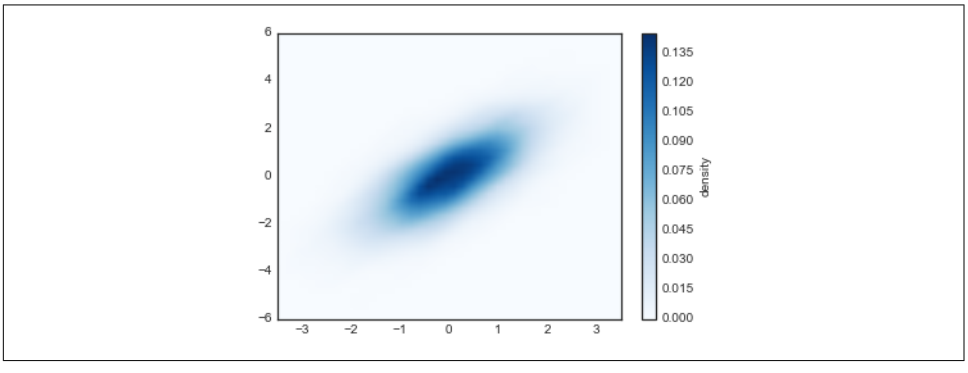

Figure 4-40. A kernel density representation of a distribution

KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off). The literature on choosing an appropriate smoothing length is vast: `gaussian_kde` uses a rule of thumb to attempt to find a nearly optimal smoothing length for the input data.

Other KDE implementations are available within the SciPy ecosystem, each with its own various strengths and weaknesses; see, for example, `sklearn.neighbors.KernelDensity` and `statsmodels.nonparametric.kernel_density.KDEMultivariate`. For visualizations based on KDE, using Matplotlib tends to be overly verbose. The Seaborn library, discussed in "Visualization with Seaborn" on page 311, provides a much more terse API for creating KDE-based visualizations.

Customizing Plot Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

The simplest legend can be created with the `plt.legend()` command, which automatically creates a legend for any labeled plot elements (Figure 4-41):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
```

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

In[3]: x = np.linspace(0, 10, 1000)
       fig, ax = plt.subplots()
       ax.plot(x, np.sin(x), '-b', label='Sine')
       ax.plot(x, np.cos(x), '--r', label='Cosine')
       ax.axis('equal')
       leg = ax.legend();
```

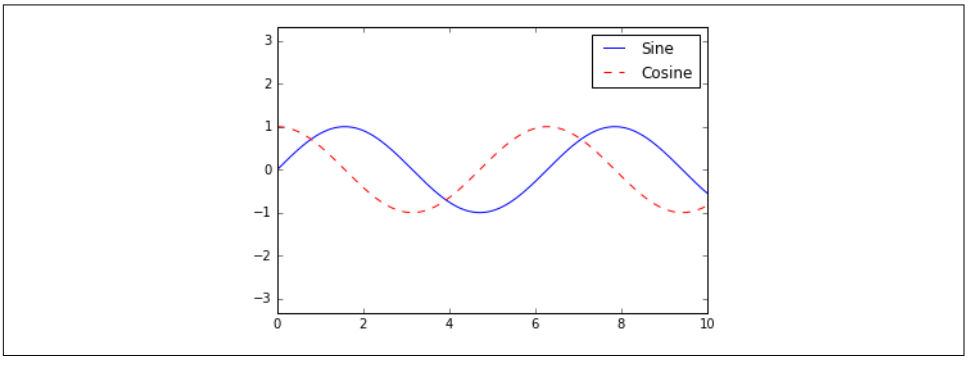

Figure 4-41. A default plot legend

But there are many ways we might want to customize such a legend. For example, we can specify the location and turn off the frame (Figure 4-42):

```
In[4]: ax.legend(loc='upper left', frameon=False)
       fig
```

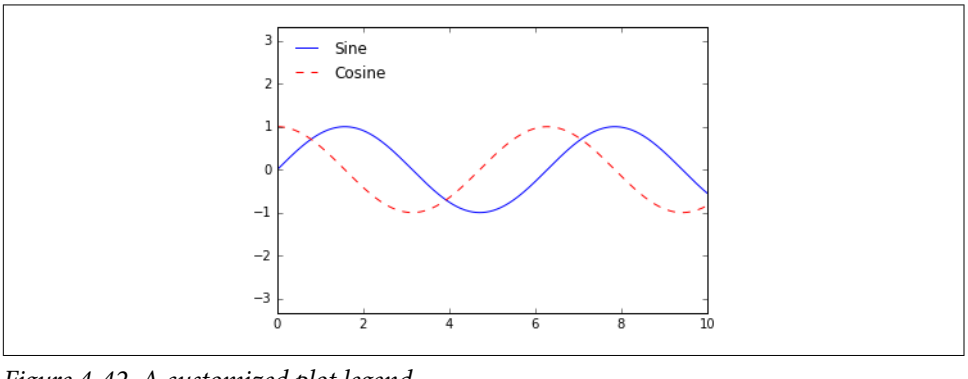

Figure 4-42. A customized plot legend

We can use the `ncol` command to specify the number of columns in the legend (Figure 4-43):

```
In[5]: ax.legend(frameon=False, loc='lower center', ncol=2)
       fig
```

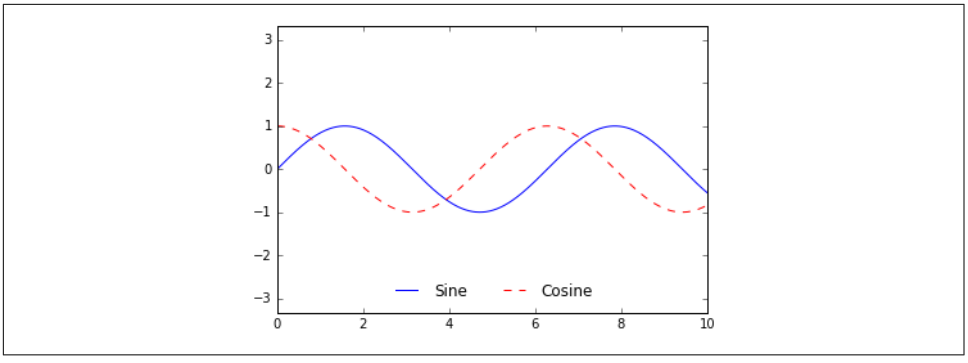

Figure 4-43. A two-column plot legend

We can use a rounded box (`fancybox`) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text (Figure 4-44):

```
In[6]: ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
       fig
```

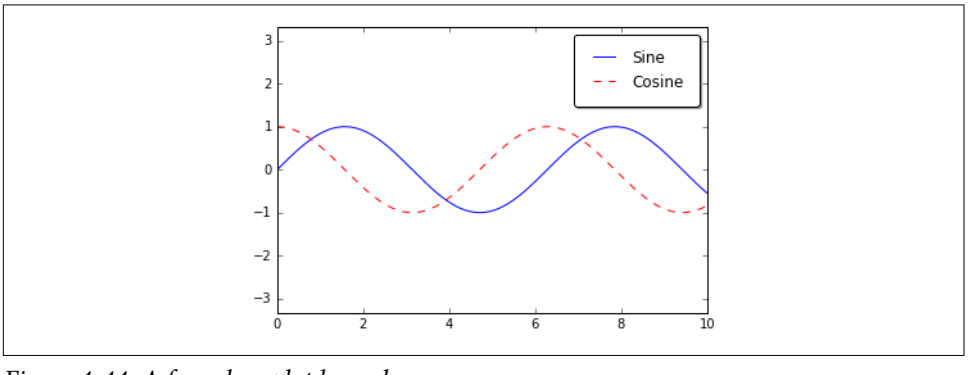

Figure 4-44. A fancybox plot legend

For more information on available legend options, see the `plt.legend` docstring.

Choosing Elements for the Legend

As we've already seen, the legend includes all labeled elements by default. If this is not what is desired, we can fine-tune which elements and labels appear in the legend by using the objects returned by plot commands. The `plt.plot()` command is able to create multiple lines at once, and returns a list of created line instances. Passing any of these to `plt.legend()` will tell it which to identify, along with the labels we'd like to specify (Figure 4-45):

```
In[7]: y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
       lines = plt.plot(x, y)
```

```
# lines is a list of plt.Line2D instances
plt.legend(lines[:2], ['first', 'second']);
```

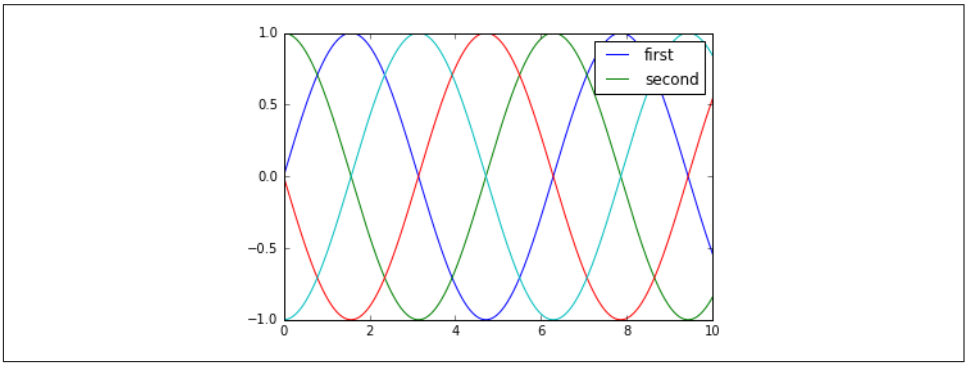

Figure 4-45. Customization of legend elements

I generally find in practice that it is clearer to use the first method, applying labels to the plot elements you'd like to show on the legend (Figure 4-46):

```
In[8]: plt.plot(x, y[:, 0], label='first')
       plt.plot(x, y[:, 1], label='second')
       plt.plot(x, y[:, 2:])
       plt.legend(framealpha=1, frameon=True);
```

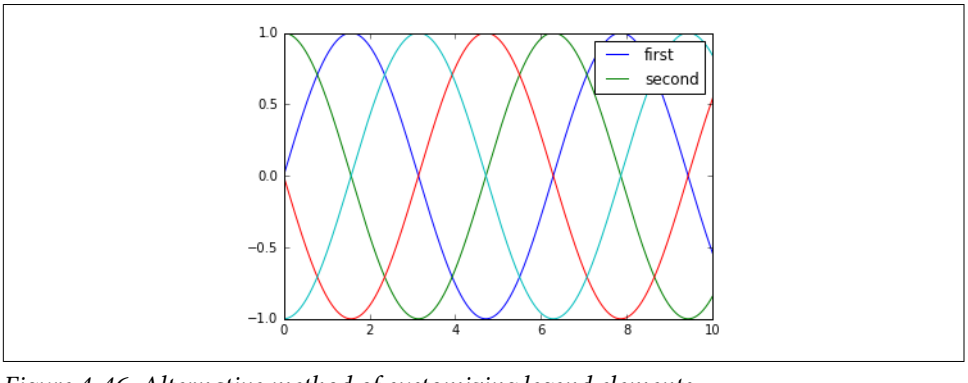

Figure 4-46. Alternative method of customizing legend elements

Notice that by default, the legend ignores all elements without a `label` attribute set.

Legend for Size of Points

Sometimes the legend defaults are not sufficient for the given visualization. For example, perhaps you're using the size of points to mark certain features of the data, and want to create a legend reflecting this. Here is an example where we'll use the size of points to indicate populations of California cities. We'd like a legend that specifies the

scale of the sizes of the points, and we'll accomplish this by plotting some labeled data with no entries (Figure 4-47):

```
In[9]: import pandas as pd
       cities = pd.read_csv('data/california_cities.csv')

       # Extract the data we're interested in
       lat, lon = cities['latd'], cities['longd']
       population, area = cities['population_total'], cities['area_total_km2']

       # Scatter the points, using size and color but no label
       plt.scatter(lon, lat, label=None,
                   c=np.log10(population), cmap='viridis',
                   s=area, linewidth=0, alpha=0.5)
       plt.axis(aspect='equal')
       plt.xlabel('longitude')
       plt.ylabel('latitude')
       plt.colorbar(label='log$_{10}$(population)')
       plt.clim(3, 7)

       # Here we create a legend:
       # we'll plot empty lists with the desired size and label
       for area in [100, 300, 500]:
           plt.scatter([], [], c='k', alpha=0.3, s=area,
                       label=str(area) + ' km$^2$')
       plt.legend(scatterpoints=1, frameon=False,
                  labelspacing=1, title='City Area')

       plt.title('California Cities: Area and Population');
```

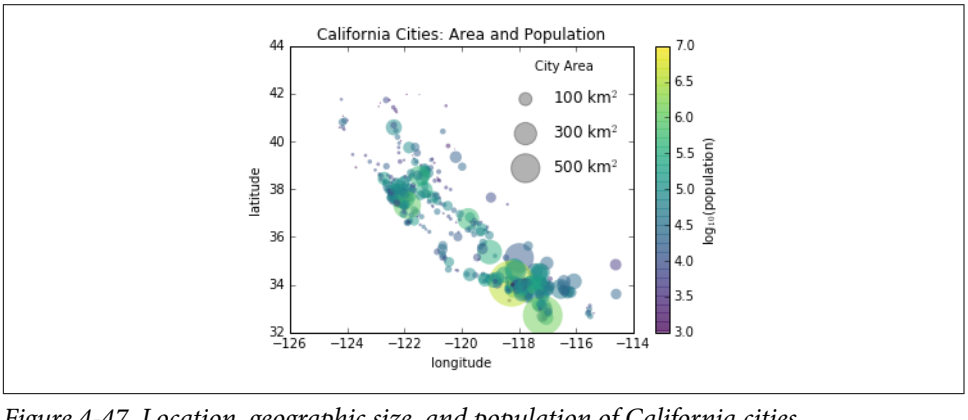

Figure 4-47. Location, geographic size, and population of California cities

The legend will always reference some object that is on the plot, so if we'd like to display a particular shape we need to plot it. In this case, the objects we want (gray circles) are not on the plot, so we fake them by plotting empty lists. Notice too that the legend only lists plot elements that have a label specified.

By plotting empty lists, we create labeled plot objects that are picked up by the legend, and now our legend tells us some useful information. This strategy can be useful for creating more sophisticated visualizations.

Finally, note that for geographic data like this, it would be clearer if we could show state boundaries or other map-specific elements. For this, an excellent choice of tool is Matplotlib's Basemap add-on toolkit, which we'll explore in "Geographic Data with Basemap" on page 298.

Multiple Legends

Sometimes when designing a plot you'd like to add multiple legends to the same axes. Unfortunately, Matplotlib does not make this easy: via the standard `legend` interface, it is only possible to create a single legend for the entire plot. If you try to create a second legend using `plt.legend()` or `ax.legend()`, it will simply override the first one. We can work around this by creating a new legend artist from scratch, and then using the lower-level `ax.add_artist()` method to manually add the second artist to the plot (Figure 4-48):

```
In[10]: fig, ax = plt.subplots()

        lines = []
        styles = ['-', '--', '-.', ':']
        x = np.linspace(0, 10, 1000)

        for i in range(4):
            lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                             styles[i], color='black')
        ax.axis('equal')

        # specify the lines and labels of the first legend
        ax.legend(lines[:2], ['line A', 'line B'],
                  loc='upper right', frameon=False)

        # Create the second legend and add the artist manually.
        from matplotlib.legend import Legend
        leg = Legend(ax, lines[2:], ['line C', 'line D'],
                     loc='lower right', frameon=False)
        ax.add_artist(leg);
```

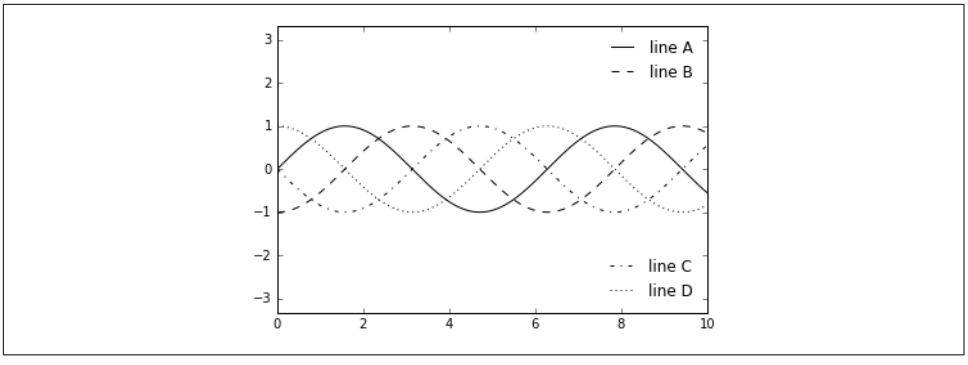

Figure 4-48. A split plot legend

This is a peek into the low-level artist objects that compose any Matplotlib plot. If you examine the source code of `ax.legend()` (recall that you can do this within the IPython notebook using `ax.legend??`) you'll see that the function simply consists of some logic to create a suitable `Legend` artist, which is then saved in the `legend_` attribute and added to the figure when the plot is drawn.

Customizing Colorbars

Plot legends identify discrete labels of discrete points. For continuous labels based on the color of points, lines, or regions, a labeled colorbar can be a great tool. In Matplotlib, a colorbar is a separate axes that can provide a key for the meaning of colors in a plot. Because the book is printed in black and white, this section has an accompanying online appendix where you can view the figures in full color (*https://github.com/jakevdp/PythonDataScienceHandbook*). We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')

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

As we have seen several times throughout this section, the simplest colorbar can be created with the `plt.colorbar` function (Figure 4-49):

```
In[3]: x = np.linspace(0, 10, 1000)
       I = np.sin(x) * np.cos(x[:, np.newaxis])

       plt.imshow(I)
       plt.colorbar();
```

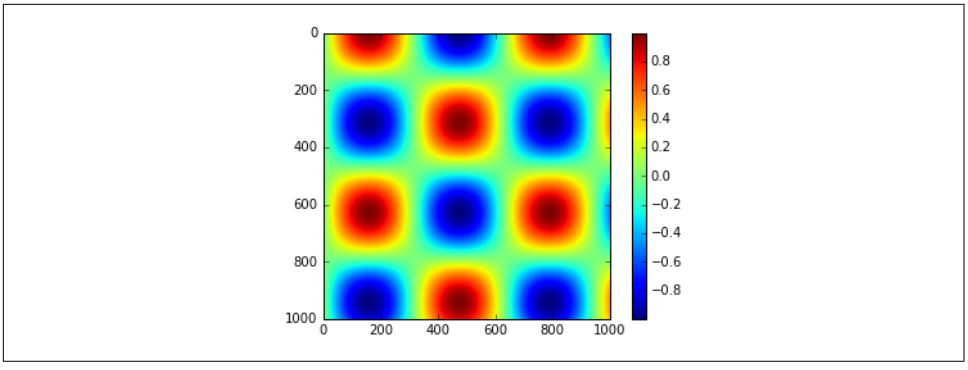

Figure 4-49. A simple colorbar legend

We'll now discuss a few ideas for customizing these colorbars and using them effectively in various situations.

Customizing Colorbars

We can specify the colormap using the `cmap` argument to the plotting function that is creating the visualization (Figure 4-50):

```
In[4]: plt.imshow(I, cmap='gray');
```

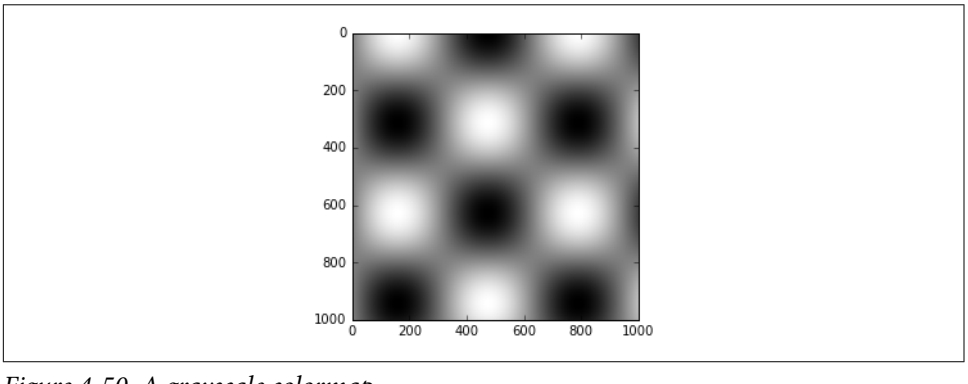

Figure 4-50. A grayscale colormap

All the available colormaps are in the `plt.cm` namespace; using IPython's tab-completion feature will give you a full list of built-in possibilities:

```
plt.cm.<TAB>
```

But being *able* to choose a colormap is just the first step: more important is how to *decide* among the possibilities! The choice turns out to be much more subtle than you might initially expect.

Choosing the colormap

A full treatment of color choice within visualization is beyond the scope of this book, but for entertaining reading on this subject and others, see the article "Ten Simple Rules for Better Figures". Matplotlib's online documentation also has an interesting discussion of colormap choice.

Broadly, you should be aware of three different categories of colormaps:

Sequential colormaps
: These consist of one continuous sequence of colors (e.g., `binary` or `viridis`).

Divergent colormaps
: These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., `RdBu` or `PuOr`).

Qualitative colormaps
: These mix colors with no particular sequence (e.g., `rainbow` or `jet`).

The `jet` colormap, which was the default in Matplotlib prior to version 2.0, is an example of a qualitative colormap. Its status as the default was quite unfortunate, because qualitative maps are often a poor choice for representing quantitative data. Among the problems is the fact that qualitative maps usually do not display any uniform progression in brightness as the scale increases.

We can see this by converting the `jet` colorbar into black and white (Figure 4-51):

```
In[5]:
from matplotlib.colors import LinearSegmentedColormap

def grayscale_cmap(cmap):
    """Return a grayscale version of the given colormap"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    # convert RGBA to perceived grayscale luminance
    # cf. http://alienryderflex.com/hsp.html
    RGB_weight = [0.299, 0.587, 0.114]
    luminance = np.sqrt(np.dot(colors[:, :3] ** 2, RGB_weight))
    colors[:, :3] = luminance[:, np.newaxis]

    return LinearSegmentedColormap.from_list(cmap.name + "_gray", colors, cmap.N)


def view_colormap(cmap):
    """Plot a colormap with its grayscale equivalent"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    cmap = grayscale_cmap(cmap)
    grayscale = cmap(np.arange(cmap.N))
```

```
    fig, ax = plt.subplots(2, figsize=(6, 2),
                           subplot_kw=dict(xticks=[], yticks=[]))
    ax[0].imshow([colors], extent=[0, 10, 0, 1])
    ax[1].imshow([grayscale], extent=[0, 10, 0, 1])
```

```
In[6]: view_colormap('jet')
```

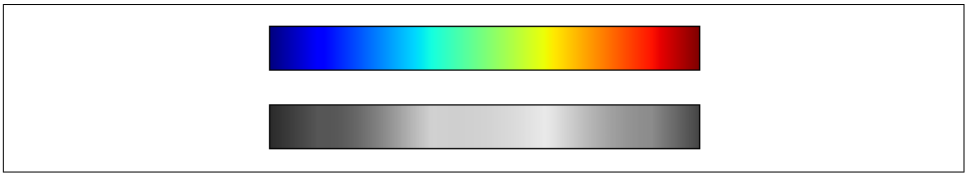

Figure 4-51. The jet colormap and its uneven luminance scale

Notice the bright stripes in the grayscale image. Even in full color, this uneven brightness means that the eye will be drawn to certain portions of the color range, which will potentially emphasize unimportant parts of the dataset. It's better to use a colormap such as `viridis` (the default as of Matplotlib 2.0), which is specifically constructed to have an even brightness variation across the range. Thus, it not only plays well with our color perception, but also will translate well to grayscale printing (Figure 4-52):

```
In[7]: view_colormap('viridis')
```

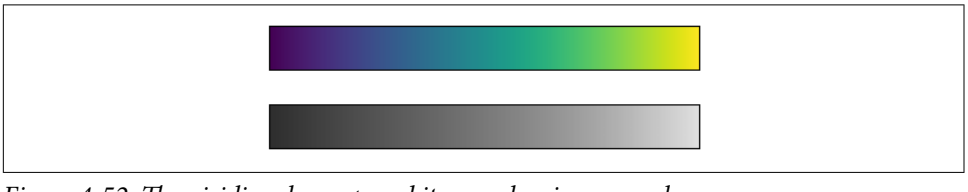

Figure 4-52. The viridis colormap and its even luminance scale

If you favor rainbow schemes, another good option for continuous data is the `cubehelix` colormap (Figure 4-53):

```
In[8]: view_colormap('cubehelix')
```

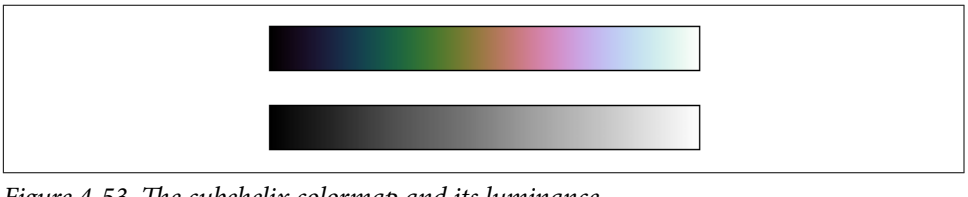

Figure 4-53. The cubehelix colormap and its luminance

For other situations, such as showing positive and negative deviations from some mean, dual-color colorbars such as `RdBu` (short for *Red-Blue*) can be useful. However,

as you can see in Figure 4-54, it's important to note that the positive-negative information will be lost upon translation to grayscale!

```
In[9]: view_colormap('RdBu')
```

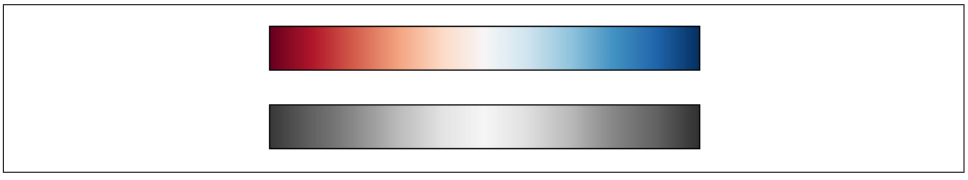

Figure 4-54. The RdBu (Red-Blue) colormap and its luminance

We'll see examples of using some of these color maps as we continue.

There are a large number of colormaps available in Matplotlib; to see a list of them, you can use IPython to explore the `plt.cm` submodule. For a more principled approach to colors in Python, you can refer to the tools and documentation within the Seaborn library (see "Visualization with Seaborn" on page 311).

Color limits and extensions

Matplotlib allows for a large range of colorbar customization. The colorbar itself is simply an instance of `plt.Axes`, so all of the axes and tick formatting tricks we've learned are applicable. The colorbar has some interesting flexibility; for example, we can narrow the color limits and indicate the out-of-bounds values with a triangular arrow at the top and bottom by setting the `extend` property. This might come in handy, for example, if you're displaying an image that is subject to noise (Figure 4-55):

```
In[10]: # make noise in 1% of the image pixels
        speckles = (np.random.random(I.shape) < 0.01)
        I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))

        plt.figure(figsize=(10, 3.5))

        plt.subplot(1, 2, 1)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar()

        plt.subplot(1, 2, 2)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar(extend='both')
        plt.clim(-1, 1);
```

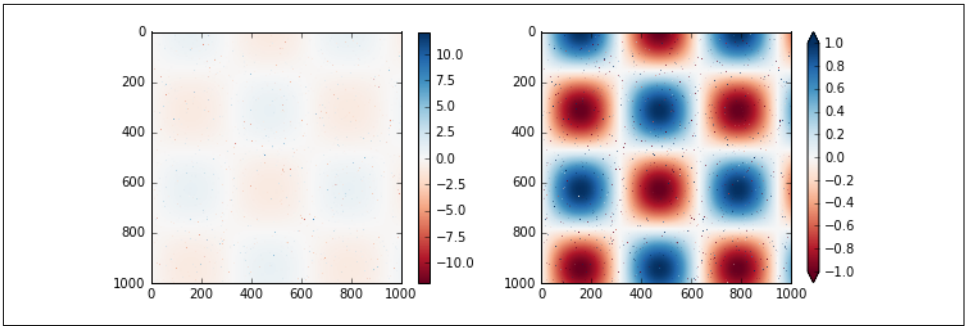

Figure 4-55. Specifying colormap extensions

Notice that in the left panel, the default color limits respond to the noisy pixels, and the range of the noise completely washes out the pattern we are interested in. In the right panel, we manually set the color limits, and add extensions to indicate values that are above or below those limits. The result is a much more useful visualization of our data.

Discrete colorbars

Colormaps are by default continuous, but sometimes you'd like to represent discrete values. The easiest way to do this is to use the `plt.cm.get_cmap()` function, and pass the name of a suitable colormap along with the number of desired bins (Figure 4-56):

```
In[11]: plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
        plt.colorbar()
        plt.clim(-1, 1);
```

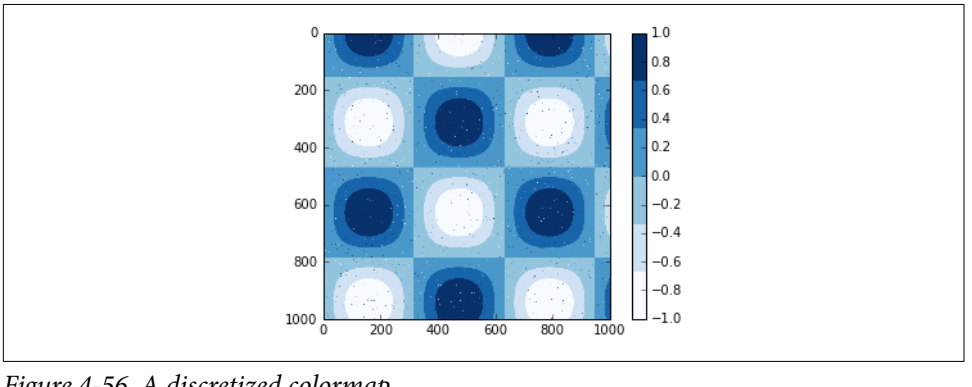

Figure 4-56. A discretized colormap

The discrete version of a colormap can be used just like any other colormap.

Example: Handwritten Digits

For an example of where this might be useful, let's look at an interesting visualization of some handwritten digits data. This data is included in Scikit-Learn, and consists of nearly 2,000 8×8 thumbnails showing various handwritten digits.

For now, let's start by downloading the digits data and visualizing several of the example images with `plt.imshow()` (Figure 4-57):

```
In[12]: # load images of the digits 0 through 5 and visualize several of them
        from sklearn.datasets import load_digits
        digits = load_digits(n_class=6)

        fig, ax = plt.subplots(8, 8, figsize=(6, 6))
        for i, axi in enumerate(ax.flat):
            axi.imshow(digits.images[i], cmap='binary')
            axi.set(xticks=[], yticks=[])
```

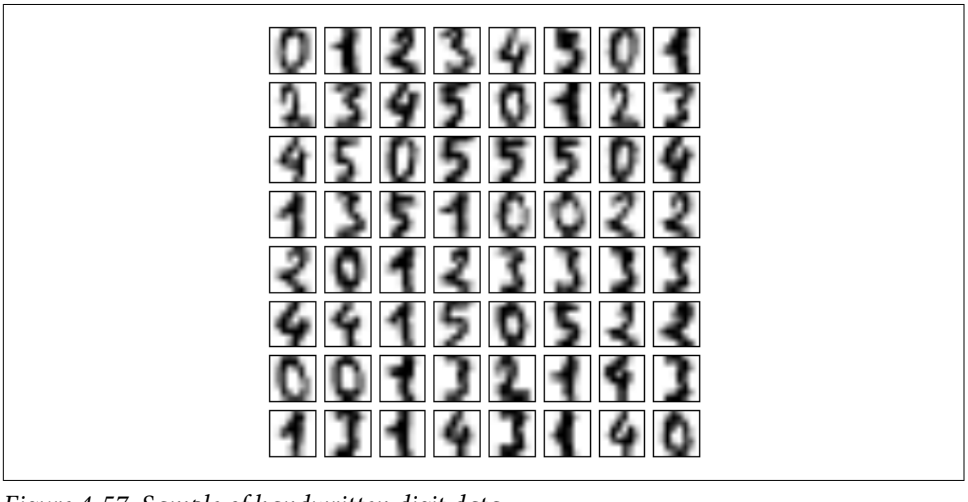

Figure 4-57. Sample of handwritten digit data

Because each digit is defined by the hue of its 64 pixels, we can consider each digit to be a point lying in 64-dimensional space: each dimension represents the brightness of one pixel. But visualizing relationships in such high-dimensional spaces can be extremely difficult. One way to approach this is to use a *dimensionality reduction* technique such as manifold learning to reduce the dimensionality of the data while maintaining the relationships of interest. Dimensionality reduction is an example of unsupervised machine learning, and we will discuss it in more detail in "What Is Machine Learning?" on page 332.

Deferring the discussion of these details, let's take a look at a two-dimensional manifold learning projection of this digits data (see "In-Depth: Manifold Learning" on page 445 for details):

```
In[13]: # project the digits into 2 dimensions using IsoMap
        from sklearn.manifold import Isomap
        iso = Isomap(n_components=2)
        projection = iso.fit_transform(digits.data)
```

We'll use our discrete colormap to view the results, setting the `ticks` and `clim` to improve the aesthetics of the resulting colorbar (Figure 4-58):

```
In[14]: # plot the results
        plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
                    c=digits.target, cmap=plt.cm.get_cmap('cubehelix', 6))
        plt.colorbar(ticks=range(6), label='digit value')
        plt.clim(-0.5, 5.5)
```

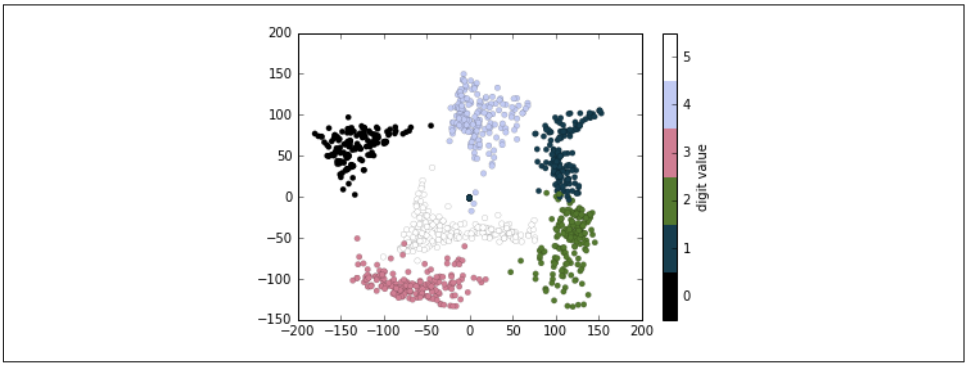

Figure 4-58. Manifold embedding of handwritten digit pixels

The projection also gives us some interesting insights on the relationships within the dataset: for example, the ranges of 5 and 3 nearly overlap in this projection, indicating that some handwritten fives and threes are difficult to distinguish, and therefore more likely to be confused by an automated classification algorithm. Other values, like 0 and 1, are more distantly separated, and therefore much less likely to be confused. This observation agrees with our intuition, because 5 and 3 look much more similar than do 0 and 1.

Customizing Matplotlib: Configurations and Stylesheets

Matplotlib's default plot settings are often the subject of complaint among its users. While much is slated to change in the 2.0 Matplotlib release, the ability to customize default settings helps bring the package in line with your own aesthetic preferences.

Here we'll walk through some of Matplotlib's runtime configuration (`rc`) options, and take a look at the newer *stylesheets* feature, which contains some nice sets of default configurations.

Plot Customization by Hand

Throughout this chapter, we've seen how it is possible to tweak individual plot settings to end up with something that looks a little bit nicer than the default. It's possible to do these customizations for each individual plot. For example, here is a fairly drab default histogram (Figure 4-81):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
       import numpy as np

       %matplotlib inline
```

```
In[2]: x = np.random.randn(1000)
       plt.hist(x);
```

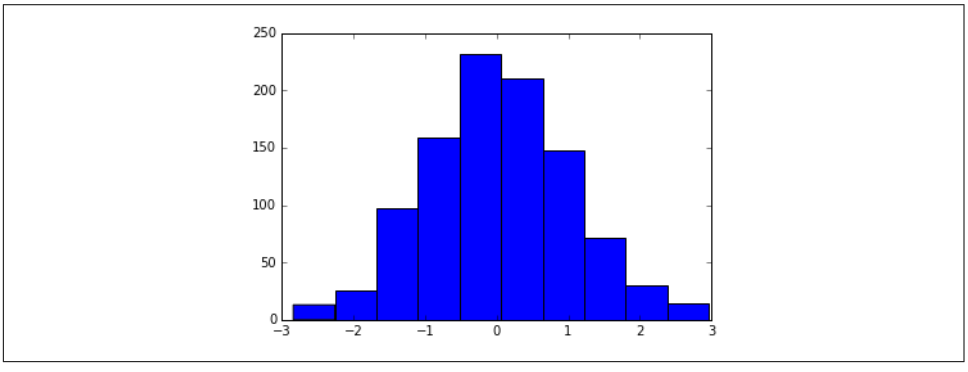

Figure 4-81. A histogram in Matplotlib's default style

We can adjust this by hand to make it a much more visually pleasing plot, shown in Figure 4-82:

```
In[3]: # use a gray background
       ax = plt.axes(axisbg='#E6E6E6')
       ax.set_axisbelow(True)

       # draw solid white grid lines
       plt.grid(color='w', linestyle='solid')

       # hide axis spines
       for spine in ax.spines.values():
           spine.set_visible(False)

       # hide top and right ticks
       ax.xaxis.tick_bottom()
       ax.yaxis.tick_left()

       # lighten ticks and labels
       ax.tick_params(colors='gray', direction='out')
       for tick in ax.get_xticklabels():
           tick.set_color('gray')
       for tick in ax.get_yticklabels():
           tick.set_color('gray')

       # control face and edge color of histogram
       ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
```

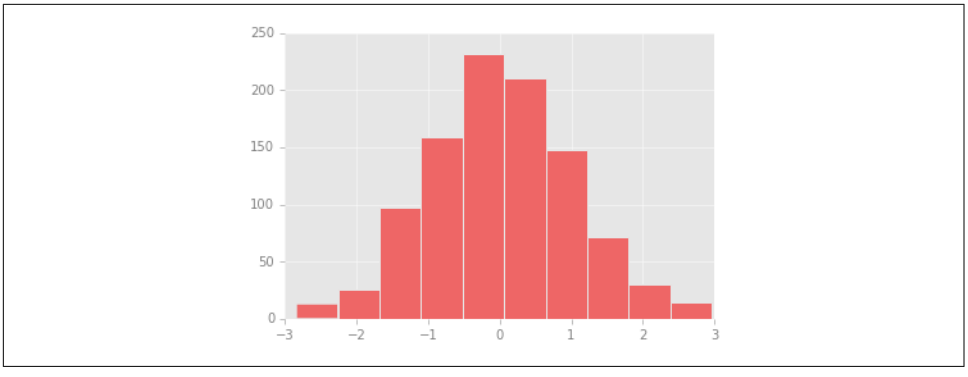

Figure 4-82. A histogram with manual customizations

This looks better, and you may recognize the look as inspired by the look of the R language's `ggplot` visualization package. But this took a whole lot of effort! We definitely do not want to have to do all that tweaking each time we create a plot. Fortunately, there is a way to adjust these defaults once in a way that will work for all plots.

Changing the Defaults: rcParams

Each time Matplotlib loads, it defines a runtime configuration (`rc`) containing the default styles for every plot element you create. You can adjust this configuration at any time using the `plt.rc` convenience routine. Let's see what it looks like to modify the `rc` parameters so that our default plot will look similar to what we did before.

We'll start by saving a copy of the current `rcParams` dictionary, so we can easily reset these changes in the current session:

```
In[4]: IPython_default = plt.rcParams.copy()
```

Now we can use the `plt.rc` function to change some of these settings:

```
In[5]: from matplotlib import cycler
       colors = cycler('color',
                       ['#EE6666', '#3388BB', '#9988DD',
                        '#EECC55', '#88BB44', '#FFBBBB'])
       plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
              axisbelow=True, grid=True, prop_cycle=colors)
       plt.rc('grid', color='w', linestyle='solid')
       plt.rc('xtick', direction='out', color='gray')
       plt.rc('ytick', direction='out', color='gray')
       plt.rc('patch', edgecolor='#E6E6E6')
       plt.rc('lines', linewidth=2)
```

With these settings defined, we can now create a plot and see our settings in action (Figure 4-83):

```
In[6]: plt.hist(x);
```

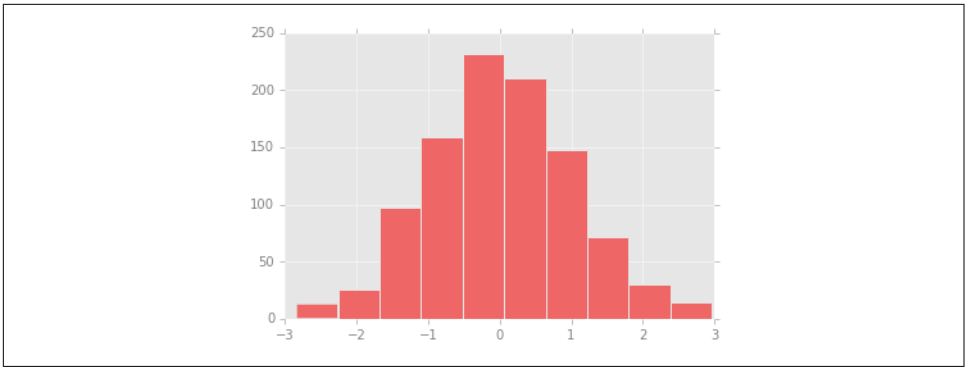

Figure 4-83. A customized histogram using rc settings

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))
```

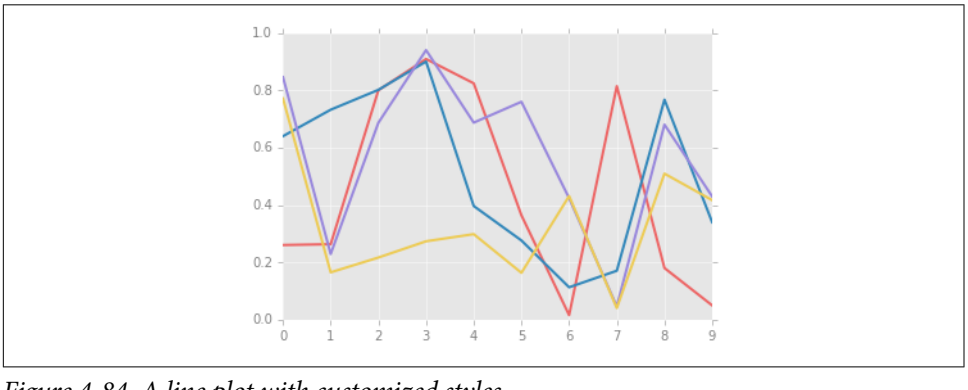

Figure 4-84. A line plot with customized styles

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()
```

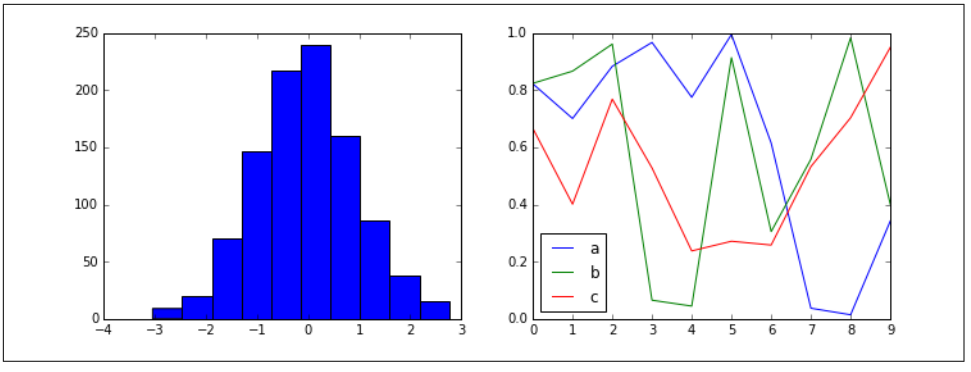

Figure 4-85. Matplotlib's default style

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()
```

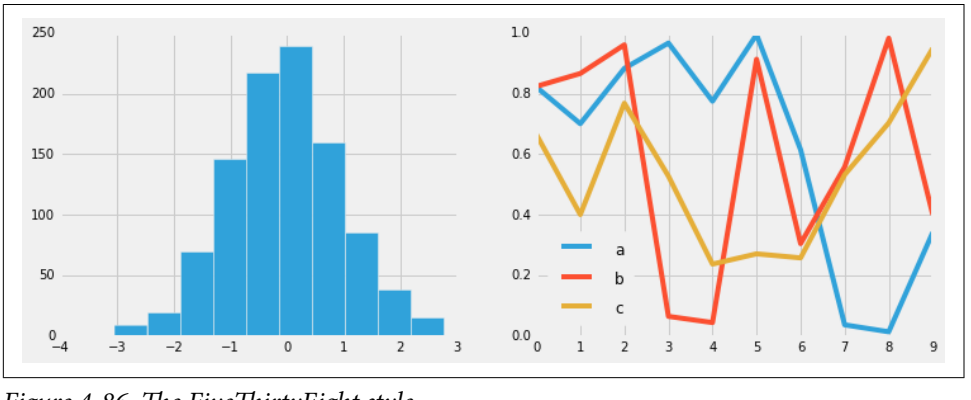

Figure 4-86. The FiveThirtyEight style

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()
```

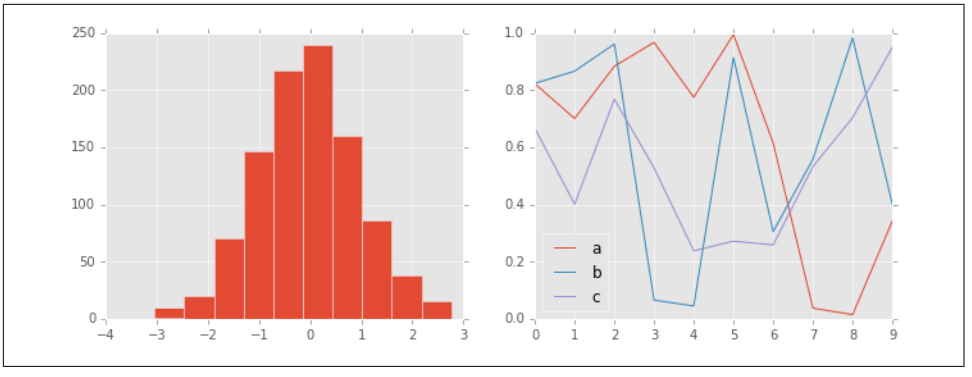

Figure 4-87. The ggplot style

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()
```

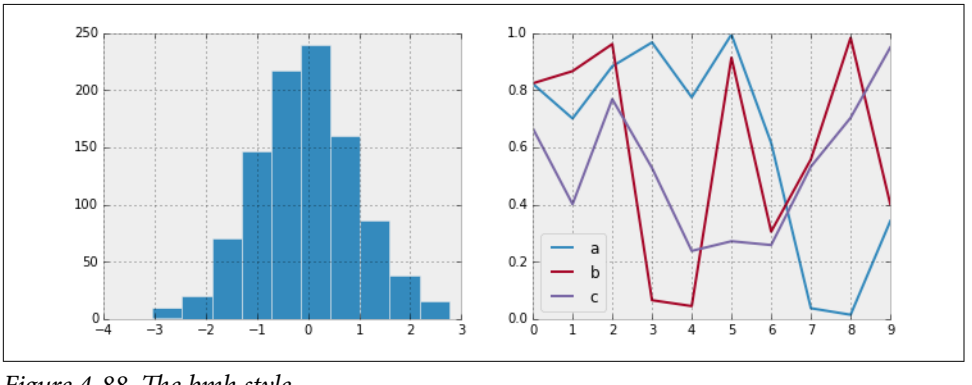

Figure 4-88. The bmh style

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()
```

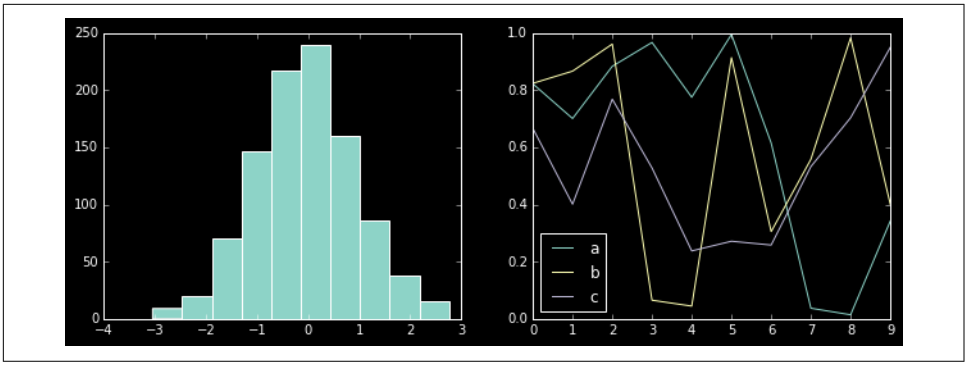

Figure 4-89. The dark_background style

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()
```

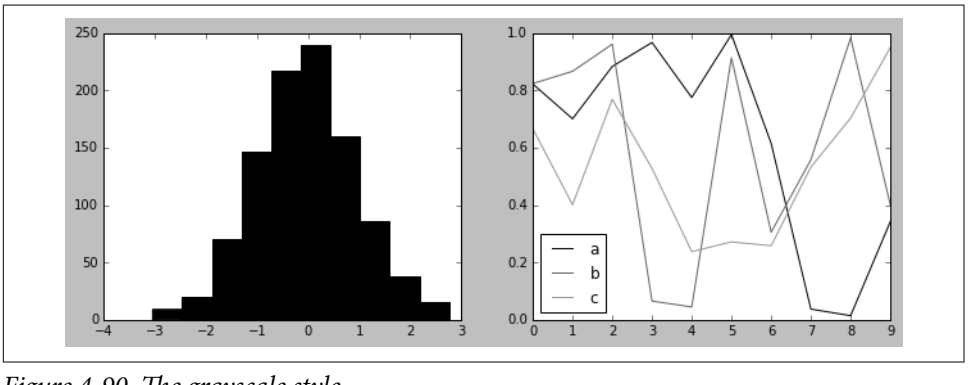

Figure 4-90. The grayscale style

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()
```

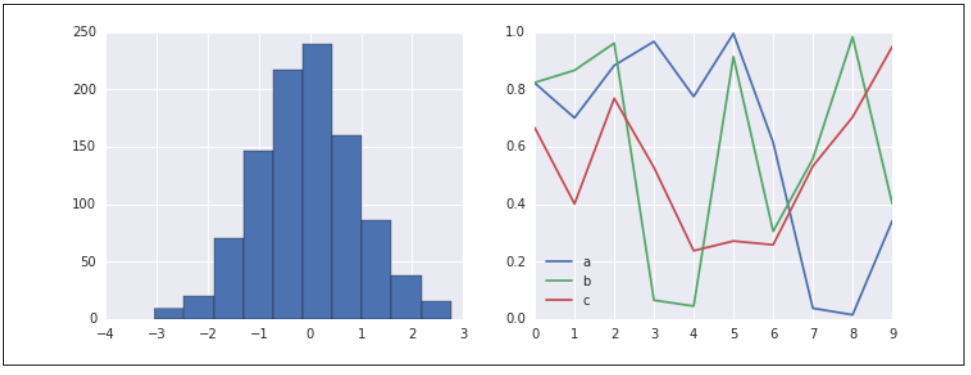

Figure 4-91. Seaborn's plotting style

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.

Visualization with Seaborn

Matplotlib has proven to be an incredibly useful and popular visualization tool, but even avid users will admit it often leaves much to be desired. There are several valid complaints about Matplotlib that often come up:

- Prior to version 2.0, Matplotlib's defaults are not exactly the best choices. It was based off of MATLAB circa 1999, and this often shows.
- Matplotlib's API is relatively low level. Doing sophisticated statistical visualization is possible, but often requires a *lot* of boilerplate code.
- Matplotlib predated Pandas by more than a decade, and thus is not designed for use with Pandas `DataFrames`. In order to visualize data from a Pandas `DataFrame`, you must extract each `Series` and often concatenate them together into the right format. It would be nicer to have a plotting library that can intelligently use the `DataFrame` labels in a plot.

An answer to these problems is Seaborn. Seaborn provides an API on top of Matplotlib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas `DataFrames`.

To be fair, the Matplotlib team is addressing this: it has recently added the `plt.style` tools (discussed in "Customizing Matplotlib: Configurations and Stylesheets" on page 282), and is starting to handle Pandas data more seamlessly. The 2.0 release of the library will include a new default stylesheet that will improve on the current status quo. But for all the reasons just discussed, Seaborn remains an extremely useful add-on.

Seaborn Versus Matplotlib

Here is an example of a simple random-walk plot in Matplotlib, using its classic plot formatting and colors. We start with the typical imports:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
       %matplotlib inline
       import numpy as np
       import pandas as pd
```

Now we create some random walk data:

```
In[2]: # Create some data
       rng = np.random.RandomState(0)
       x = np.linspace(0, 10, 500)
       y = np.cumsum(rng.randn(500, 6), 0)
```

And do a simple plot (Figure 4-111):

```
In[3]: # Plot the data with Matplotlib defaults
       plt.plot(x, y)
       plt.legend('ABCDEF', ncol=2, loc='upper left');
```

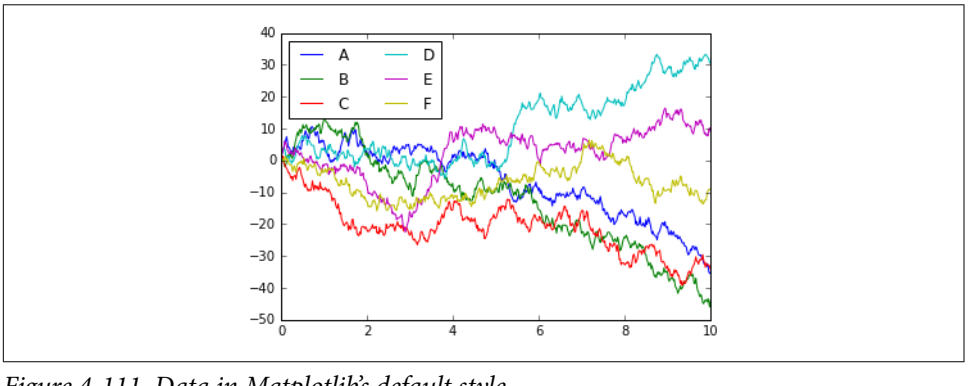

Figure 4-111. Data in Matplotlib's default style

Although the result contains all the information we'd like it to convey, it does so in a way that is not all that aesthetically pleasing, and even looks a bit old-fashioned in the context of 21st-century data visualization.

Now let's take a look at how it works with Seaborn. As we will see, Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's `set()` method. By convention, Seaborn is imported as `sns`:

```
In[4]: import seaborn as sns
       sns.set()
```

Now let's rerun the same two lines as before (Figure 4-112):

```
In[5]: # same plotting code as above!
       plt.plot(x, y)
       plt.legend('ABCDEF', ncol=2, loc='upper left');
```

Figure 4-112. Data in Seaborn's default style

Ah, much better!

Exploring Seaborn Plots

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Let's take a look at a few of the datasets and plot types available in Seaborn. Note that all of the following *could* be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood), but the Seaborn API is much more convenient.

Histograms, KDE, and densities

Often in statistical data visualization, all you want is to plot histograms and joint distributions of variables. We have seen that this is relatively straightforward in Matplotlib (Figure 4-113):

```
In[6]: data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
       data = pd.DataFrame(data, columns=['x', 'y'])

       for col in 'xy':
           plt.hist(data[col], normed=True, alpha=0.5)
```

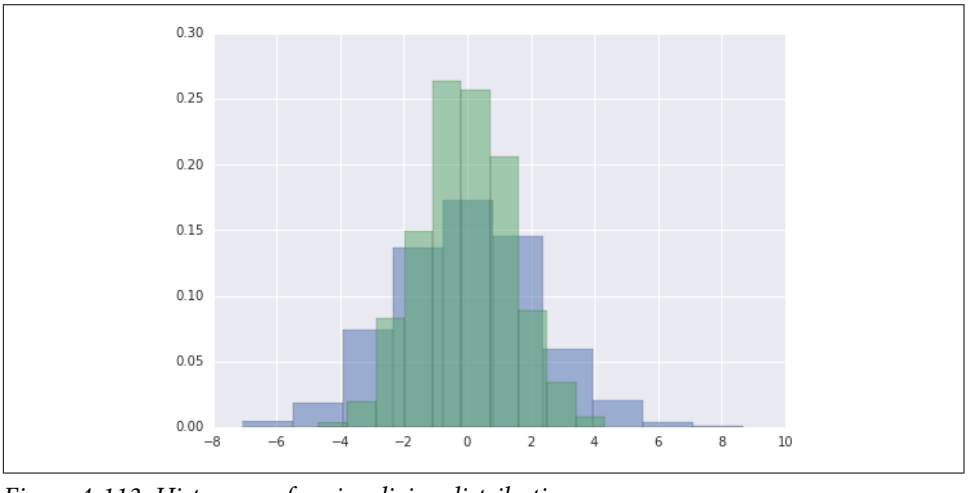

Figure 4-113. Histograms for visualizing distributions

Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with `sns.kdeplot` (Figure 4-114):

```
In[7]: for col in 'xy':
           sns.kdeplot(data[col], shade=True)
```

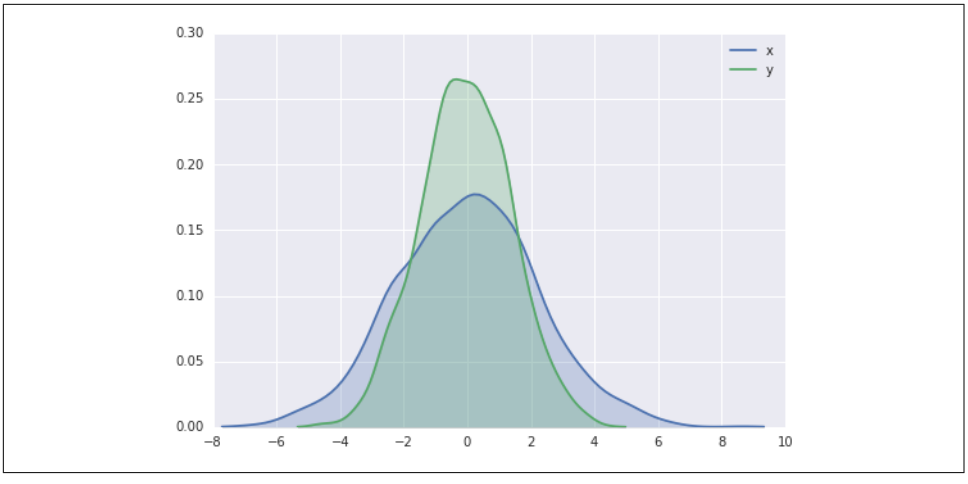

Figure 4-114. Kernel density estimates for visualizing distributions

Histograms and KDE can be combined using `distplot` (Figure 4-115):

```
In[8]: sns.distplot(data['x'])
       sns.distplot(data['y']);
```

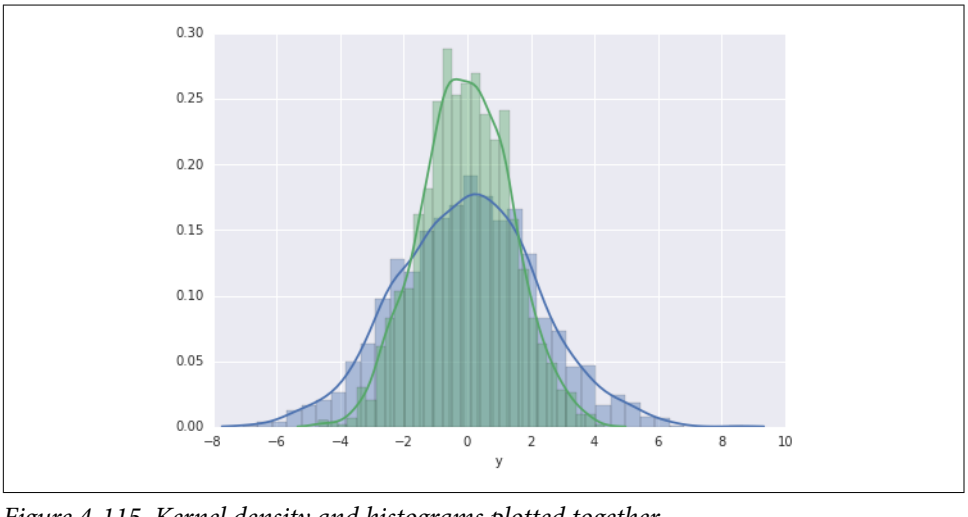

Figure 4-115. Kernel density and histograms plotted together

If we pass the full two-dimensional dataset to `kdeplot`, we will get a two-dimensional visualization of the data (Figure 4-116):

```
In[9]: sns.kdeplot(data);
```

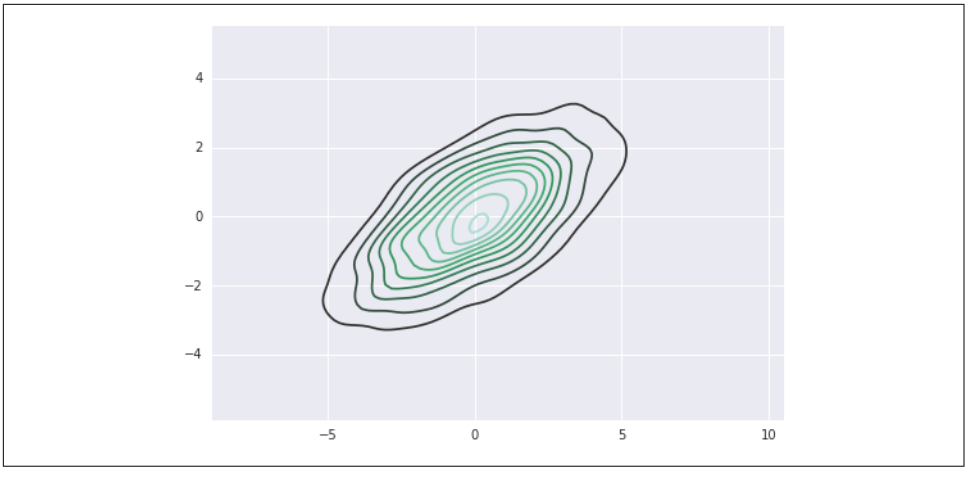

Figure 4-116. A two-dimensional kernel density plot

We can see the joint distribution and the marginal distributions together using `sns.jointplot`. For this plot, we'll set the style to a white background (Figure 4-117):

```
In[10]: with sns.axes_style('white'):
            sns.jointplot("x", "y", data, kind='kde');
```

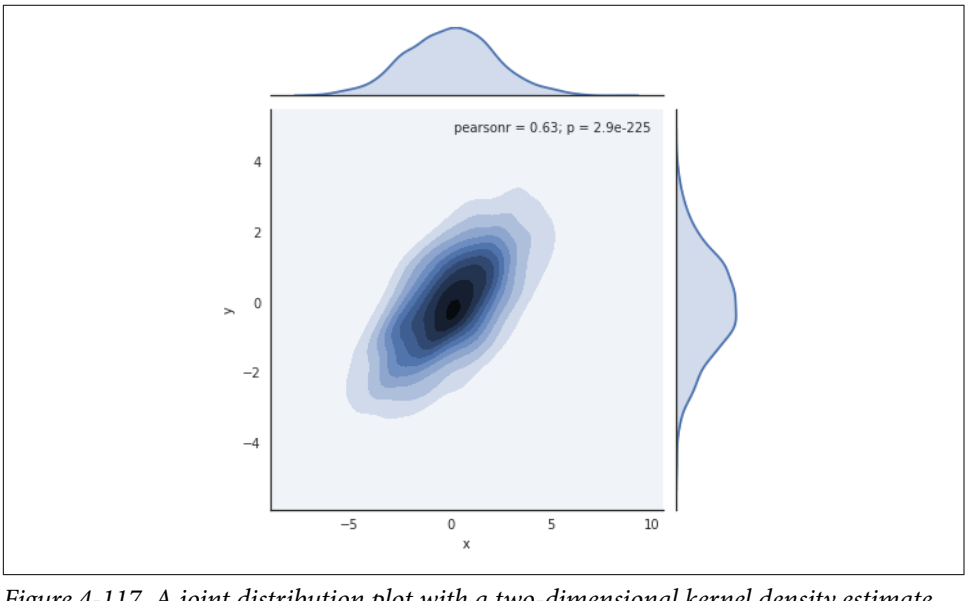

Figure 4-117. A joint distribution plot with a two-dimensional kernel density estimate

There are other parameters that can be passed to `jointplot`—for example, we can use a hexagonally based histogram instead (Figure 4-118):

```
In[11]: with sns.axes_style('white'):
            sns.jointplot("x", "y", data, kind='hex')
```

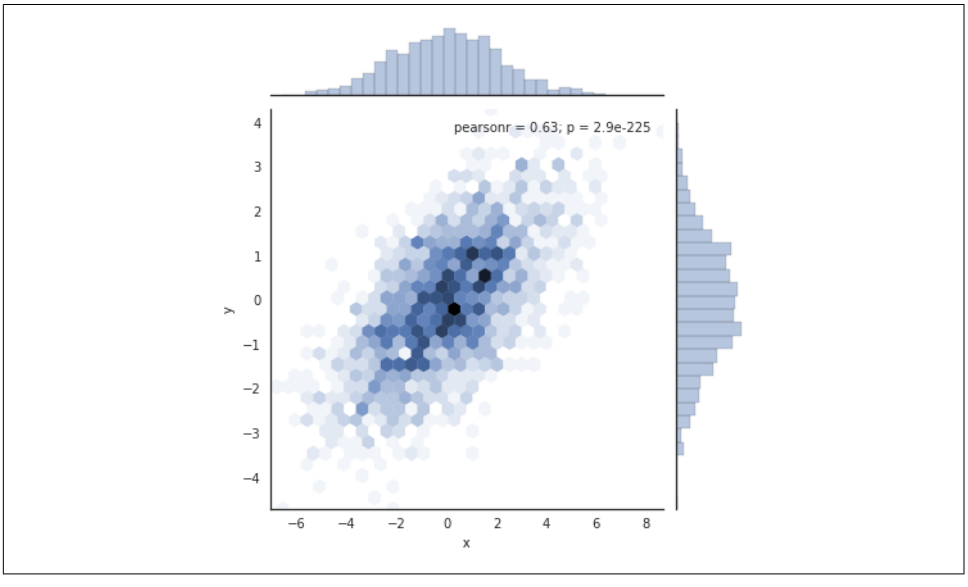

Figure 4-118. A joint distribution plot with a hexagonal bin representation

Pair plots

When you generalize joint plots to datasets of larger dimensions, you end up with *pair plots*. This is very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

We'll demo this with the well-known Iris dataset, which lists measurements of petals and sepals of three iris species:

```
In[12]: iris = sns.load_dataset("iris")
        iris.head()

Out[12]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
         2           4.7          3.2           1.3          0.2  setosa
         3           4.6          3.1           1.5          0.2  setosa
         4           5.0          3.6           1.4          0.2  setosa
```

Visualizing the multidimensional relationships among the samples is as easy as calling `sns.pairplot` (Figure 4-119):

```
In[13]: sns.pairplot(iris, hue='species', size=2.5);
```

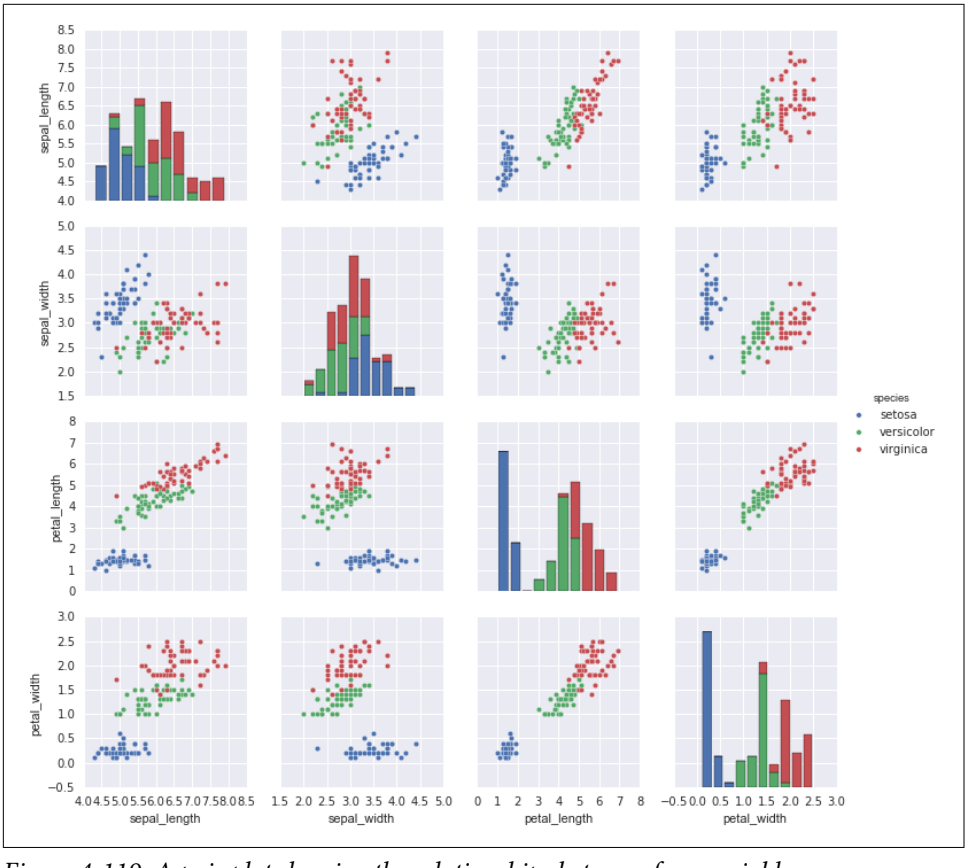

Figure 4-119. A pair plot showing the relationships between four variables

Faceted histograms

Sometimes the best way to view data is via histograms of subsets. Seaborn's `FacetGrid` makes this extremely simple. We'll take a look at some data that shows the amount that restaurant staff receive in tips based on various indicator data (Figure 4-120):

```
In[14]: tips = sns.load_dataset('tips')
        tips.head()

Out[14]:    total_bill   tip     sex smoker  day    time  size
         0       16.99  1.01  Female     No  Sun  Dinner     2
         1       10.34  1.66    Male     No  Sun  Dinner     3
         2       21.01  3.50    Male     No  Sun  Dinner     3
```

```
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

```
In[15]: tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']

        grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
        grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15));
```

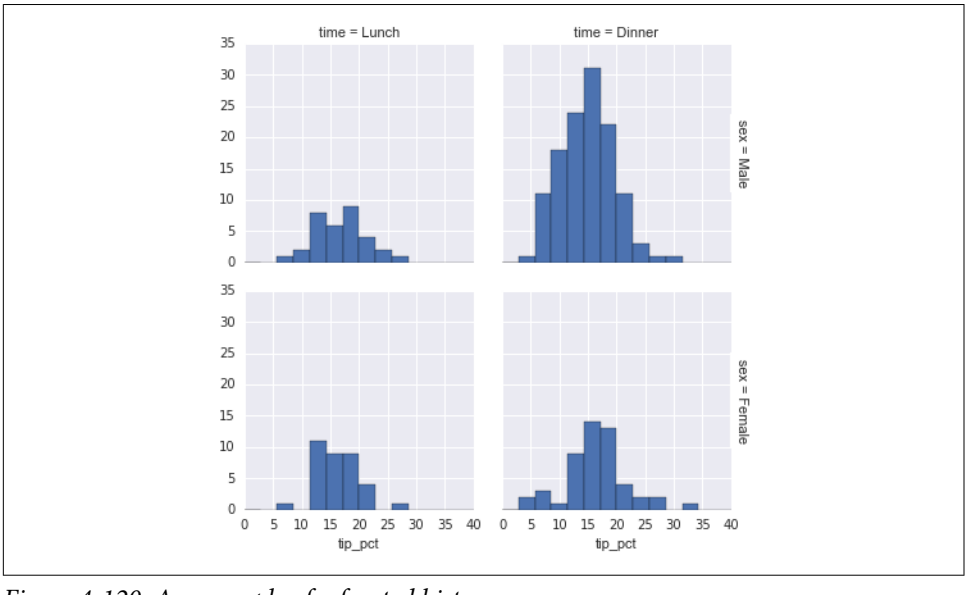

Figure 4-120. An example of a faceted histogram

Factor plots

Factor plots can be useful for this kind of visualization as well. This allows you to view the distribution of a parameter within bins defined by any other parameter (Figure 4-121):

```
In[16]: with sns.axes_style(style='ticks'):
            g = sns.factorplot("day", "total_bill", "sex", data=tips, kind="box")
            g.set_axis_labels("Day", "Total Bill");
```

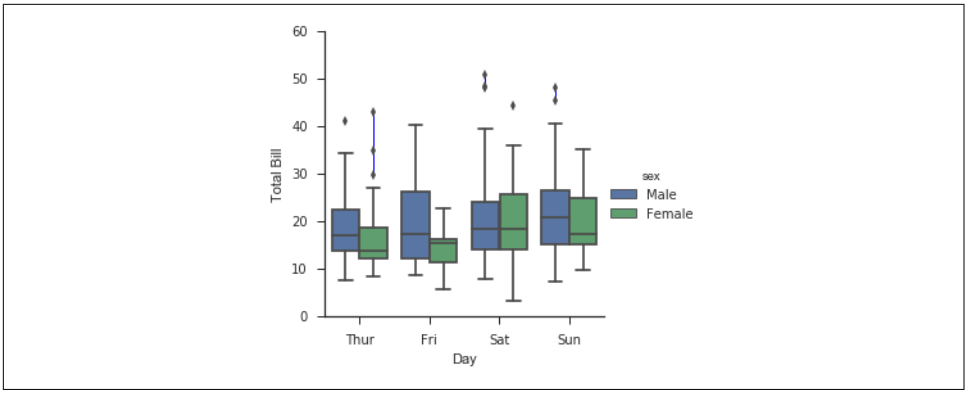

Figure 4-121. An example of a factor plot, comparing distributions given various discrete factors

Joint distributions

Similar to the pair plot we saw earlier, we can use `sns.jointplot` to show the joint distribution between different datasets, along with the associated marginal distributions (Figure 4-122):

```
In[17]: with sns.axes_style('white'):
            sns.jointplot("total_bill", "tip", data=tips, kind='hex')
```

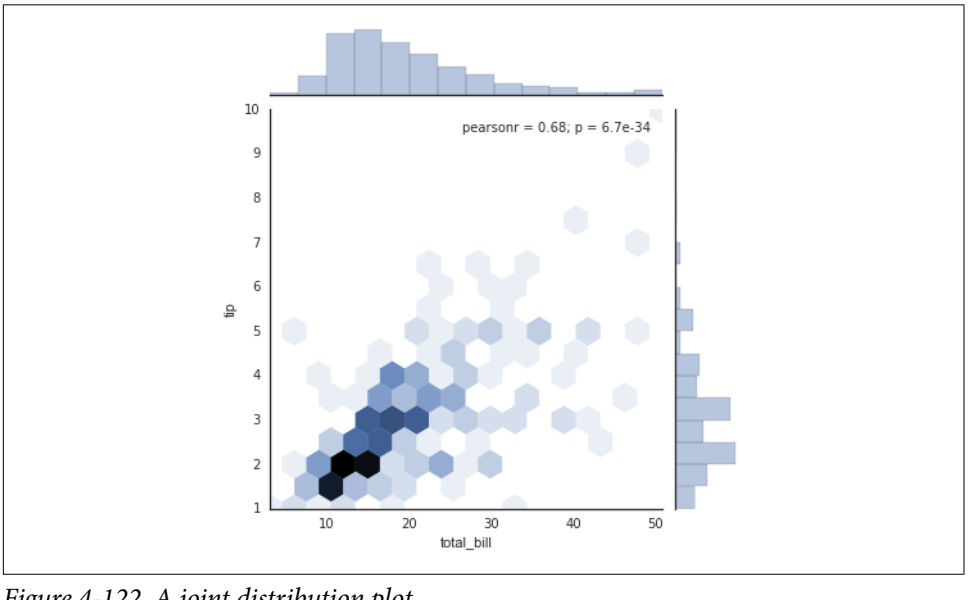

Figure 4-122. A joint distribution plot

The joint plot can even do some automatic kernel density estimation and regression (Figure 4-123):

```
In[18]: sns.jointplot("total_bill", "tip", data=tips, kind='reg');
```

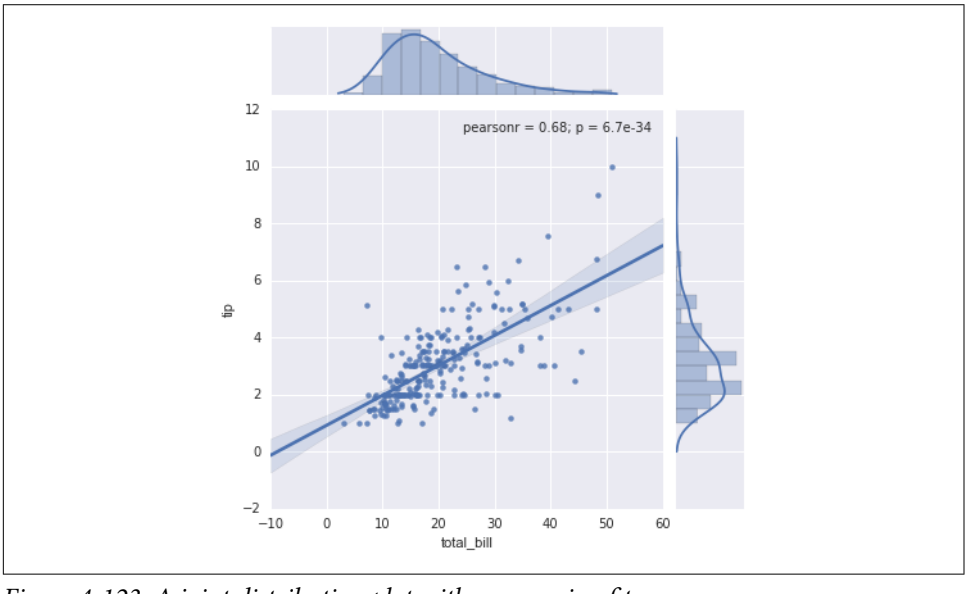

Figure 4-123. A joint distribution plot with a regression fit

Bar plots

Time series can be plotted with `sns.factorplot`. In the following example (visualized in Figure 4-124), we'll use the Planets data that we first saw in "Aggregation and Grouping" on page 158:

```
In[19]: planets = sns.load_dataset('planets')
        planets.head()
```

```
Out[19]:            method  number  orbital_period   mass  distance  year
         0  Radial Velocity       1         269.300   7.10     77.40  2006
         1  Radial Velocity       1         874.774   2.21     56.95  2008
         2  Radial Velocity       1         763.000   2.60     19.84  2011
         3  Radial Velocity       1         326.030  19.40    110.62  2007
         4  Radial Velocity       1         516.220  10.50    119.47  2009
```

```
In[20]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=2,
                               kind="count", color='steelblue')
            g.set_xticklabels(step=5)
```

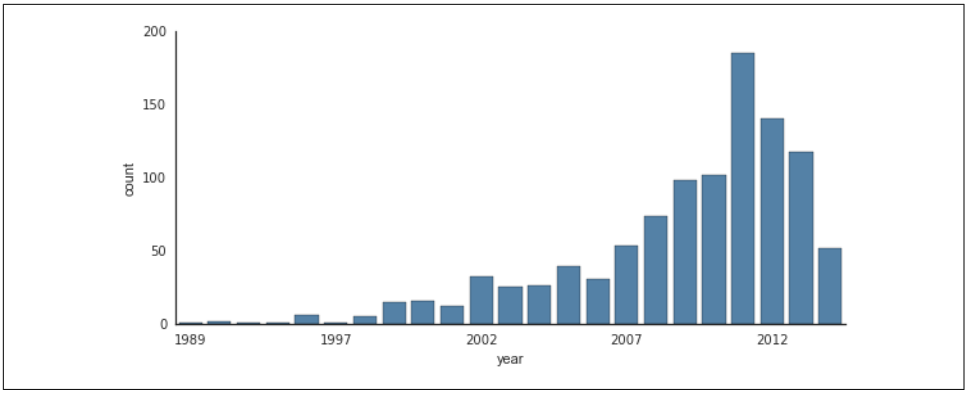

Figure 4-124. A histogram as a special case of a factor plot

We can learn more by looking at the *method* of discovery of each of these planets, as illustrated in Figure 4-125:

```
In[21]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=4.0, kind='count',
                               hue='method', order=range(2001, 2015))
            g.set_ylabels('Number of Planets Discovered')
```

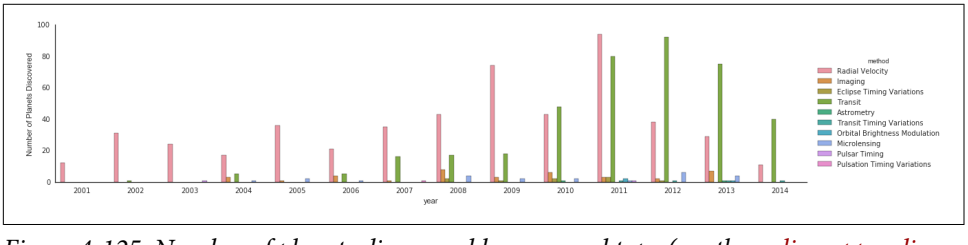

Figure 4-125. Number of planets discovered by year and type (see the online appendix for a full-scale figure)

For more information on plotting with Seaborn, see the Seaborn documentation, a tutorial, and the Seaborn gallery.

Example: Exploring Marathon Finishing Times

Here we'll look at using Seaborn to help visualize and understand finishing results from a marathon. I've scraped the data from sources on the Web, aggregated it and removed any identifying information, and put it on GitHub where it can be downloaded (if you are interested in using Python for web scraping, I would recommend *Web Scraping with Python* by Ryan Mitchell). We will start by downloading the data from the Web, and loading it into Pandas:

```
In[22]:
# !curl -O https://raw.githubusercontent.com/jakevdp/marathon-data/
# master/marathon-data.csv
```

```
In[23]: data = pd.read_csv('marathon-data.csv')
        data.head()
```

```
Out[23]:    age gender     split     final
         0   33      M  01:05:38  02:08:51
         1   32      M  01:06:26  02:09:28
         2   31      M  01:06:49  02:10:42
         3   38      M  01:06:16  02:13:45
         4   31      M  01:06:32  02:13:59
```

By default, Pandas loaded the time columns as Python strings (type `object`); we can see this by looking at the `dtypes` attribute of the `DataFrame`:

```
In[24]: data.dtypes
```

```
Out[24]: age        int64
         gender    object
         split     object
         final     object
         dtype: object
```

Let's fix this by providing a converter for the times:

```
In[25]: def convert_time(s):
            h, m, s = map(int, s.split(':'))
            return pd.datetools.timedelta(hours=h, minutes=m, seconds=s)

        data = pd.read_csv('marathon-data.csv',
                           converters={'split':convert_time, 'final':convert_time})
        data.head()
```

```
Out[25]:    age gender    split    final
         0   33      M 01:05:38 02:08:51
         1   32      M 01:06:26 02:09:28
         2   31      M 01:06:49 02:10:42
         3   38      M 01:06:16 02:13:45
         4   31      M 01:06:32 02:13:59
```

```
In[26]: data.dtypes
```

```
Out[26]: age                 int64
         gender             object
         split     timedelta64[ns]
         final     timedelta64[ns]
         dtype: object
```

That looks much better. For the purpose of our Seaborn plotting utilities, let's next add columns that give the times in seconds:

```
In[27]: data['split_sec'] = data['split'].astype(int) / 1E9
        data['final_sec'] = data['final'].astype(int) / 1E9
        data.head()
```

```
Out[27]:    age gender    split    final  split_sec  final_sec
          0   33      M 01:05:38 02:08:51     3938.0     7731.0
          1   32      M 01:06:26 02:09:28     3986.0     7768.0
          2   31      M 01:06:49 02:10:42     4009.0     7842.0
          3   38      M 01:06:16 02:13:45     3976.0     8025.0
          4   31      M 01:06:32 02:13:59     3992.0     8039.0
```

To get an idea of what the data looks like, we can plot a `jointplot` over the data (Figure 4-126):

```
In[28]: with sns.axes_style('white'):
            g = sns.jointplot("split_sec", "final_sec", data, kind='hex')
            g.ax_joint.plot(np.linspace(4000, 16000),
                            np.linspace(8000, 32000), ':k')
```

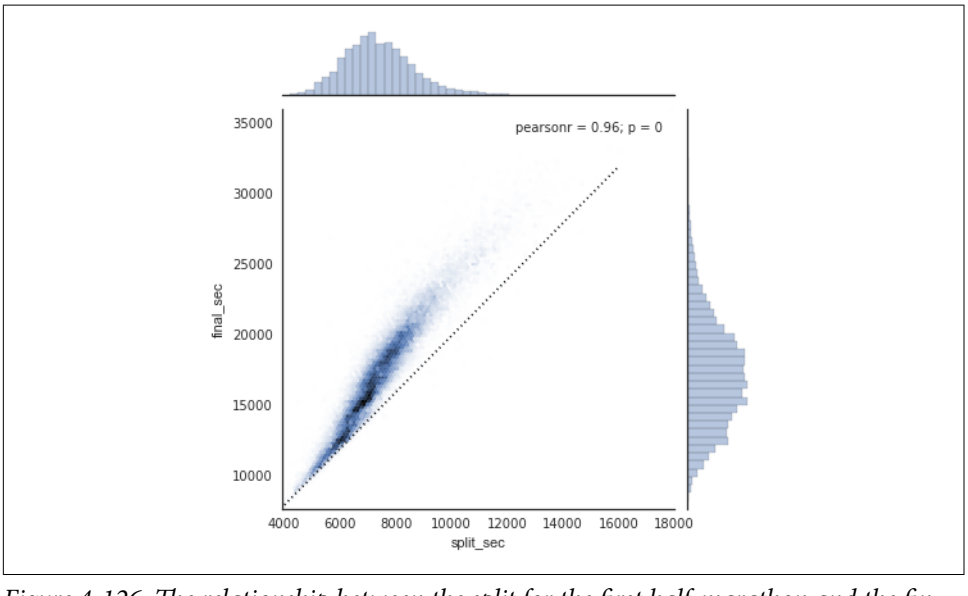

Figure 4-126. The relationship between the split for the first half-marathon and the finishing time for the full marathon

The dotted line shows where someone's time would lie if they ran the marathon at a perfectly steady pace. The fact that the distribution lies above this indicates (as you might expect) that most people slow down over the course of the marathon. If you have run competitively, you'll know that those who do the opposite—run faster during the second half of the race—are said to have "negative-split" the race.

Let's create another column in the data, the split fraction, which measures the degree to which each runner negative-splits or positive-splits the race:

```
In[29]: data['split_frac'] = 1 - 2 * data['split_sec'] / data['final_sec']
        data.head()
```

```
Out[29]:    age gender    split    final  split_sec  final_sec  split_frac
          0   33      M 01:05:38 02:08:51     3938.0     7731.0   -0.018756
          1   32      M 01:06:26 02:09:28     3986.0     7768.0   -0.026262
          2   31      M 01:06:49 02:10:42     4009.0     7842.0   -0.022443
          3   38      M 01:06:16 02:13:45     3976.0     8025.0    0.009097
          4   31      M 01:06:32 02:13:59     3992.0     8039.0    0.006842
```

Where this split difference is less than zero, the person negative-split the race by that fraction. Let's do a distribution plot of this split fraction (Figure 4-127):

```
In[30]: sns.distplot(data['split_frac'], kde=False);
        plt.axvline(0, color="k", linestyle="--");
```

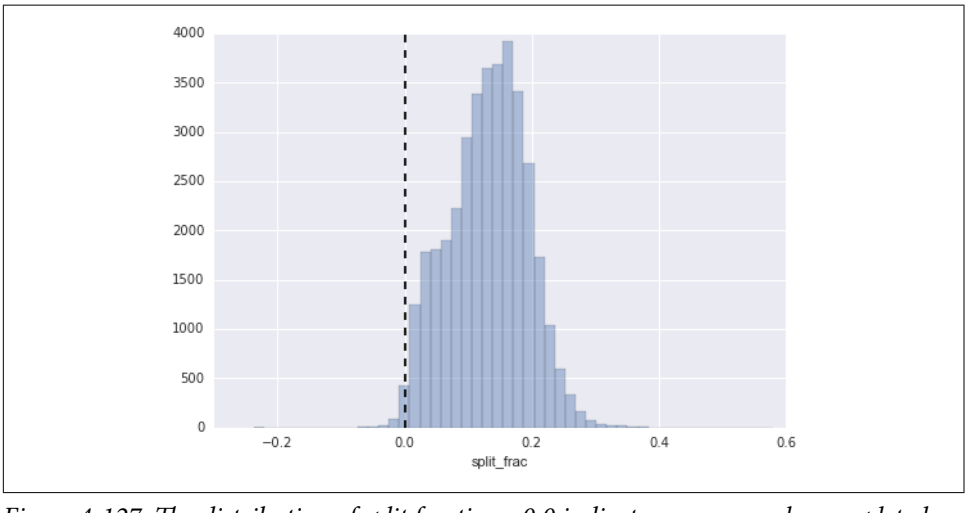

Figure 4-127. The distribution of split fractions; 0.0 indicates a runner who completed the first and second halves in identical times

```
In[31]: sum(data.split_frac < 0)
```

```
Out[31]: 251
```

Out of nearly 40,000 participants, there were only 250 people who negative-split their marathon.

Let's see whether there is any correlation between this split fraction and other variables. We'll do this using a `pairgrid`, which draws plots of all these correlations (Figure 4-128):

```
In[32]:
g = sns.PairGrid(data, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
                 hue='gender', palette='RdBu_r')
g.map(plt.scatter, alpha=0.8)
g.add_legend();
```

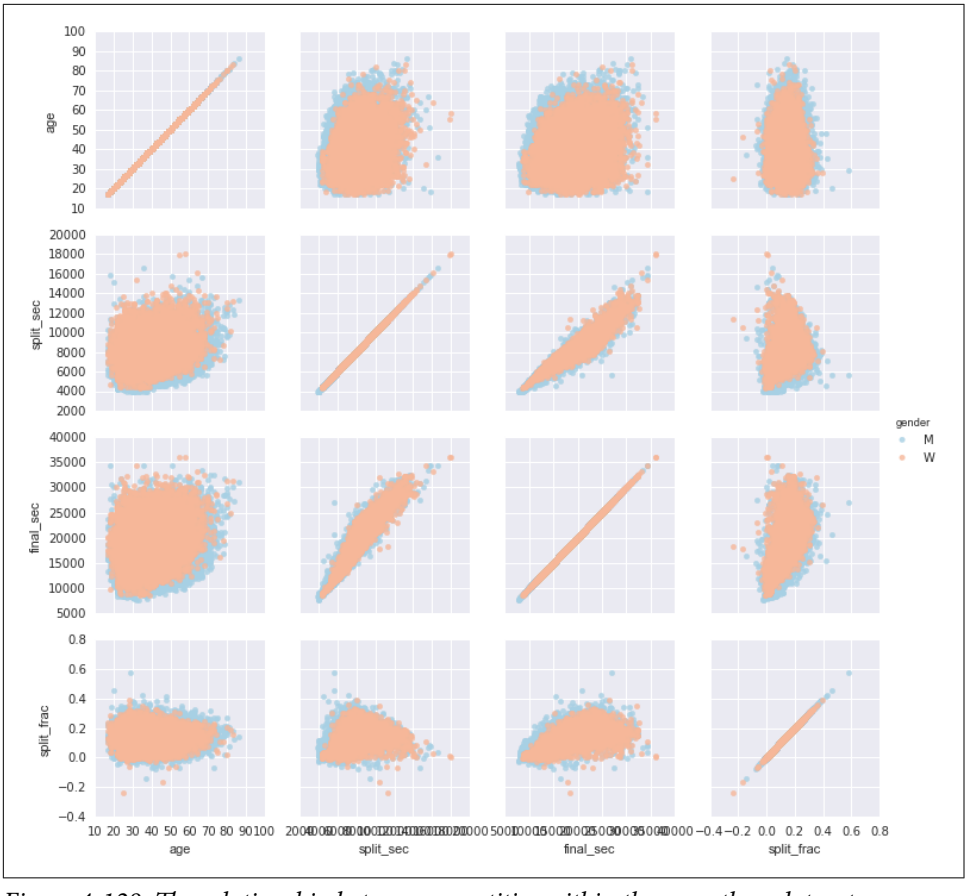

Figure 4-128. The relationship between quantities within the marathon dataset

It looks like the split fraction does not correlate particularly with age, but does correlate with the final time: faster runners tend to have closer to even splits on their marathon time. (We see here that Seaborn is no panacea for Matplotlib's ills when it comes to plot styles: in particular, the *x*-axis labels overlap. Because the output is a simple Matplotlib plot, however, the methods in "Customizing Ticks" on page 275 can be used to adjust such things if desired.)

The difference between men and women here is interesting. Let's look at the histogram of split fractions for these two groups (Figure 4-129):

```
In[33]: sns.kdeplot(data.split_frac[data.gender=='M'], label='men', shade=True)
        sns.kdeplot(data.split_frac[data.gender=='W'], label='women', shade=True)
        plt.xlabel('split_frac');
```

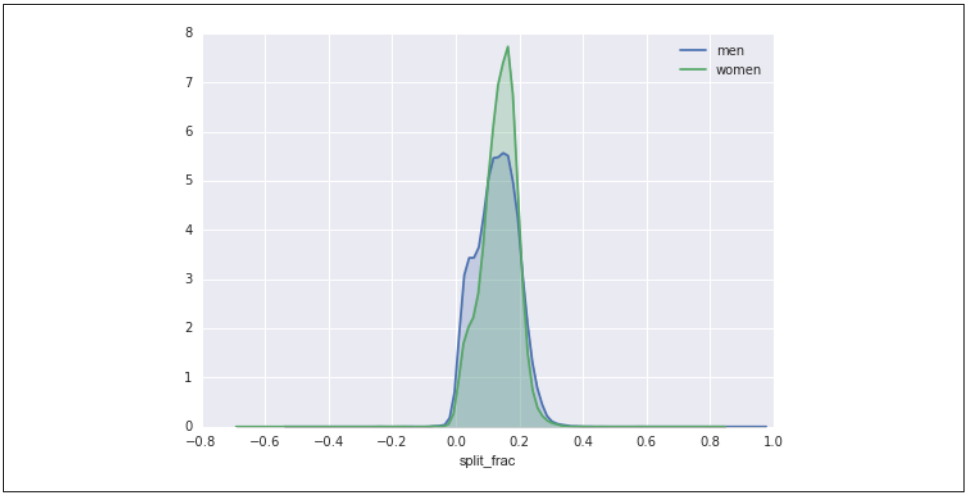

Figure 4-129. The distribution of split fractions by gender

The interesting thing here is that there are many more men than women who are running close to an even split! This almost looks like some kind of bimodal distribution among the men and women. Let's see if we can suss out what's going on by looking at the distributions as a function of age.

A nice way to compare distributions is to use a *violin plot* (Figure 4-130):

```
In[34]:
sns.violinplot("gender", "split_frac", data=data,
               palette=["lightblue", "lightpink"]);
```

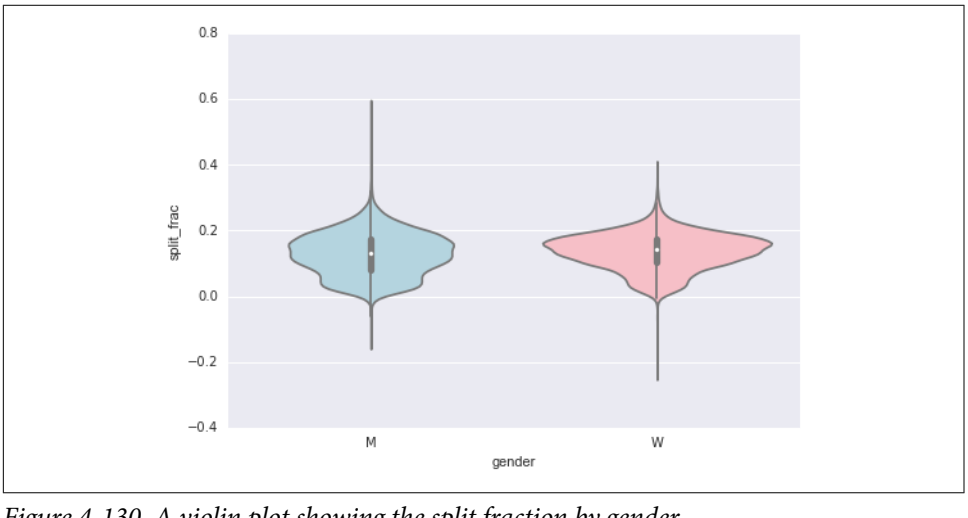

Figure 4-130. A violin plot showing the split fraction by gender

This is yet another way to compare the distributions between men and women.

Let's look a little deeper, and compare these violin plots as a function of age. We'll start by creating a new column in the array that specifies the decade of age that each person is in (Figure 4-131):

```
In[35]: data['age_dec'] = data.age.map(lambda age: 10 * (age // 10))
data.head()
```

```
Out[35]:
   age gender    split    final  split_sec  final_sec  split_frac  age_dec
0   33      M 01:05:38 02:08:51     3938.0     7731.0   -0.018756       30
1   32      M 01:06:26 02:09:28     3986.0     7768.0   -0.026262       30
2   31      M 01:06:49 02:10:42     4009.0     7842.0   -0.022443       30
3   38      M 01:06:16 02:13:45     3976.0     8025.0    0.009097       30
4   31      M 01:06:32 02:13:59     3992.0     8039.0    0.006842       30
```

```
In[36]:
men = (data.gender == 'M')
women = (data.gender == 'W')

with sns.axes_style(style=None):
    sns.violinplot("age_dec", "split_frac", hue="gender", data=data,
                   split=True, inner="quartile",
                   palette=["lightblue", "lightpink"]);
```

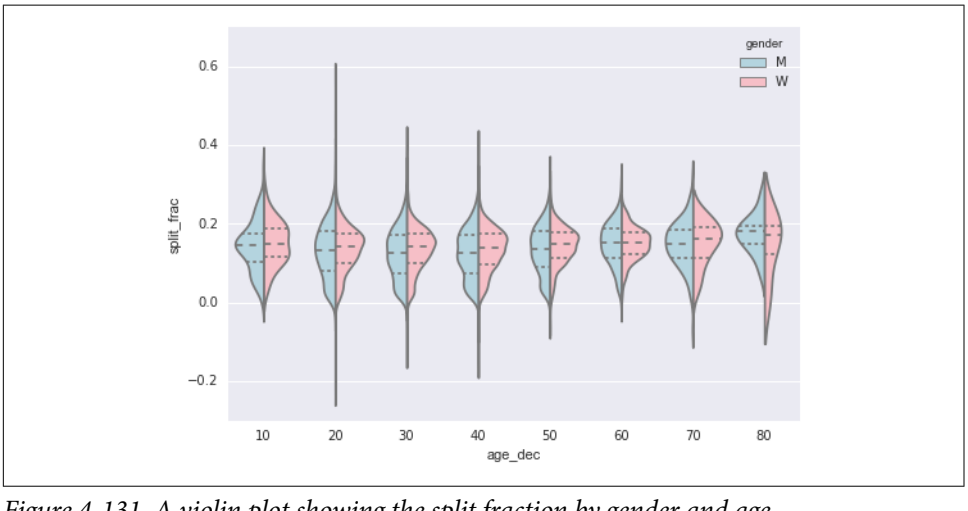

Figure 4-131. A violin plot showing the split fraction by gender and age

Looking at this, we can see where the distributions of men and women differ: the split distributions of men in their 20s to 50s show a pronounced over-density toward lower splits when compared to women of the same age (or of any age, for that matter).

Also surprisingly, the 80-year-old women seem to outperform *everyone* in terms of their split time. This is probably due to the fact that we're estimating the distribution from small numbers, as there are only a handful of runners in that range:

```
In[38]: (data.age > 80).sum()

Out[38]: 7
```

Back to the men with negative splits: who are these runners? Does this split fraction correlate with finishing quickly? We can plot this very easily. We'll use `regplot`, which will automatically fit a linear regression to the data (Figure 4-132):

```
In[37]: g = sns.lmplot('final_sec', 'split_frac', col='gender', data=data,
                       markers=".", scatter_kws=dict(color='c'))
        g.map(plt.axhline, y=0.1, color="k", ls=":");
```

Figure 4-132. Split fraction versus finishing time by gender

Apparently the people with fast splits are the elite runners who are finishing within ~15,000 seconds, or about 4 hours. People slower than that are much less likely to have a fast second split.

Further Resources

Matplotlib Resources

A single chapter in a book can never hope to cover all the available features and plot types available in Matplotlib. As with other packages we've seen, liberal use of IPython's tab-completion and help functions (see "Help and Documentation in IPython" on page 3) can be very helpful when you're exploring Matplotlib's API. In addition, Matplotlib's online documentation can be a helpful reference. See in particular the