CHAPTER 6

Graphs and Plots

Visualizing Data

Graphs and plots are efficient methods to present data. Done properly, a graph can convey an idea better than an entire article.

What a graph should portray is a function of your target audience. So when you plot a graph, bear that in mind. If your target audience is technical people, they might require additional technical information. If your target audience is investors, another approach is required. In this chapter, I won't be discussing what to present and what not; instead, I'll show you *how* to present. Examples include how to plot bar charts and pie charts, how to add markers and control line ticks, how to annotate the graphs with text and arrows, and more.

Regardless of your target audience, some ideas and methodologies always hold true. Sources for data should be accurate and verified. Graphs should be easily reproducible by running the code that generated them. (How many times did your boss ask you to modify your last report? If the graph had been generated with a documented script, doing so would have proved easy enough.) And lastly, your graphs should be aesthetically pleasing. Consulting with colleagues could be beneficial as well: what do they understand from the graph? Was the key idea captured? Is the output professional?

In this chapter we'll discuss the basics of creating and annotating graphs. We'll start by exploring the `plot()` function, continue with text and grid annotation, explore some other types of graphs, and lastly, introduce patches, a method to attach graphical objects to a figure.

The *matplotlib* Package

The *matplotlib* package, available at `http://matplotlib.org/`, is the main graphing and plotting tool used throughout this book. The package is versatile and highly configurable, supporting several graphing interfaces. *Matplotlib*, together with *NumPy* and *SciPy* (see Chapters 7 and 8), provides MATLAB-like graphing capabilities.

The benefits of using *matplotlib* in the context of data analysis and visualization are as follows:

- Plotting data is simple and intuitive.
- Performance is great; output is professional.
- Integration with *NumPy* and *SciPy* (used for signal processing and numerical analysis) is seamless.
- The package is highly customizable and configurable, catering to most people's needs.

The package is quite extensive and allows, for example, embedding plots in a graphical user interface. Currently, the package supports several graphical interfaces, including wxPython (`http://www.wxpython.org/`) and PyGTK (`http://www.pygtk.org/`), to name a couple. However, GUI topics are beyond the scope of the book. We will focus

on plotting graphs, rather than discussing the GUI engine itself. For a full account of *matplotlib*, try the online documentation, available as a PDF document, at http://matplotlib.sourceforge.net/Matplotlib.pdf.

Going forward, you should ensure that you have *matplotlib* installed and working properly. Refer to Chapter 2 if you require additional information on installing the package or visit the package's web site.

Interactive Graphs vs. Image Files

There are several ways you can use *matplotlib*:

- Create dynamic content to be served on a web server. For example, you might generate stock price images on the fly or display traffic information on top of a map.
- Embed it in a graphical user interface, allowing users to interact with an application to visualize data.
- Automatically process data and generate output in a variety of file formats, including JPEG, PNG (Portable Network Graphics—see http://www.libpng.org/pub/png/), PDF, and PostScript (PS). This option is best suited for batch processing a large number of files.
- Run it interactively, with the Python shell in X or Windows. This option is good during the development phases of the code.

Of the preceding options, and in view of the book topics, we'll explore two: generating plots of varying file formats and using *matplotlib* interactively. I typically use both options, depending on the stage of my code. In the early stages of development, I work interactively with a small sample of the data: plot, zoom in, change graph parameters, annotate, and then rinse and repeat. Once my code is ready, I let it loose, so to speak, on the full set of data files. Since that might mean tens if not hundreds of graph windows, I prefer to write them as files instead, and then use an image viewer to view the results one at a time.

So let's start. First and foremost, import *PyLab* as follows:

```
>>> from pylab import *
```

As previously mentioned, this imports *matplotlib*, *NumPy*, and *SciPy*. Although, generally speaking, you shouldn't import everything quite so liberally, you should make an exception in the case of *PyLab*: it's considerably easier to work with the entire package loaded into memory; and unless memory is a constraint, the usability is great. Going forward, I'll assume you've imported *PyLab* as just described.

WHERE DOES THIS FUNCTION COME FROM?

A frustration to some, especially those experienced with Python, is that issuing the command `from pylab import *` will import several packages. How do you know whether a specific function is a part of *NumPy* or *matplotlib*? The solution is simple—use `help`! For example, here's the output from `help(diff)`:

```
>>> help(diff)
Help on function diff in module numpy.lib.function_base:

diff(a, n=1, axis=-1)
    Calculate the nth order discrete difference along given axis.
```

As you can see in the first line, `diff()` is a function from the *NumPy* package, or more specifically, `numpy.lib.function_base`.

Our next step is to plot a graph. We'll plot the list, `[0, 1, 2]`:

```
>>> plot(range(3))
```

```
[<matplotlib.lines.Line2D object at 0x00000000058800F0>]
```

There's no visible output yet (other than *matplotlib*'s response), and the reason for this is that we haven't specified how exactly we want the graph drawn: interactive figures or hard copy files.

Interactive Graphs

Interactive graphs, like the one shown in Figure 6-1, plot the graph in a separate window. If you'd like this option, enter `show()` at the Python shell or call the function `show()` in a script.

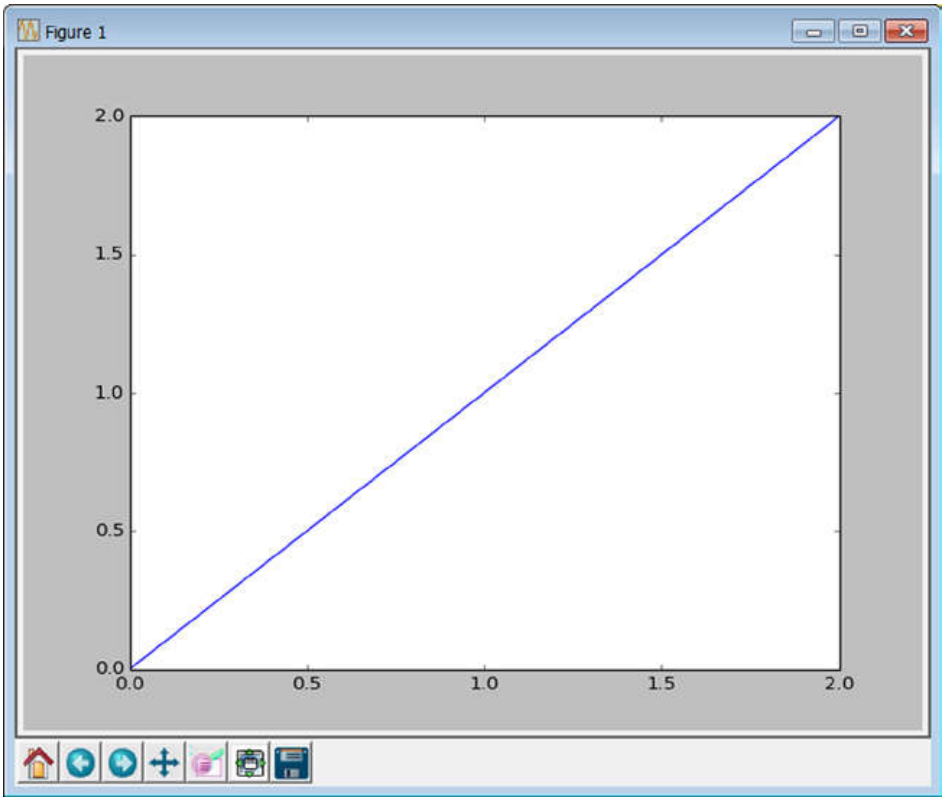

Figure 6-1. *An interactive graph*

The function `show()` opens up an interactive window. Here are several notes about this window:

- The window is numbered, as you can see by the label 'Figure 1.' This is useful if you have several windows and would like subsequent plots to either override or appear on a specific figure. To switch between figures, use the `figure(n)` function, where `n` stands for the figure index. If you'd like a new figure, and don't particularly care about the figure index, issue the command `figure()`, which will create an empty figure with the next available index.

- The x-axis and y-axis were created automatically to fit the data. In a lot of the cases, *matplotlib* does an excellent job of automatically selecting the right axis (as in this example). However, if you want a different range of values to be displayed, that's doable with the `axis()` command; you'll learn more about this later in the chapter in the "Axis" section.
- The location of the mouse is printed on the right corner of the figure. This is very useful if you're trying to zoom in on data and find a specific data point. This functionality is not available when you plot graphs to file (i.e., noninteractive mode).
- You have several buttons on the lower-left side of the figure to allow interaction with the graph. The five left-most buttons are used for zooming and zooming history. The first button from the left (with the house icon) is used to change axes to the original plot axes. The left and right arrow buttons cycle backward and forward through previous axes changes. The fourth button allows you to change the origin of the axes, and the fifth button from the left enables zooming. The sixth button from the left controls the margins of the plot with respect to the containing window. Lastly, the seventh button allows you to save the image to disk.

■ **Note** If you're not using *matplotlib* interactively in Python, be sure to call the function `show()` *after* all graphs have been generated, as it enters a user interface main loop that will stop execution of the rest of your code. The reason behind this behavior is that *matplotlib* is designed to be embedded in a GUI, as well. If you're working from interactive Python, issue the command `ion()` which stands for "interactive on"; this will generate graphs automatically, without the need to call `show()`. To reverse this behavior, issue `ioff()` (which stands for "interactive off").

Savings Graphs to Files

The function `savefig()` enables you to write images of varying formats to a file. Out of the box, *matplotlib* supports several file formats, including PDF, PNG, and PS. The simplest way to generate a file containing a graph is to issue `savefig(filename)`, where `filename` has the extension associated with your selected image format:

```
>>> figure()
```

```
<matplotlib.figure.Figure object at 0x0000000002CAE160>
```

```
>>> plot(arange(3))
```

```
[<matplotlib.lines.Line2D object at 0x000000000826FEF0>]
```

```
>>> savefig('../images/line.png')
>>> from glob import glob
>>> glob('../images/*.png')
```

```
['../images/line.png']
```

I've assumed you're following the directory structure for the book (see Chapter 2) and that you can access the `images` directory via a relative path: `../images`. I've also introduced the *glob* module, which is used to perform wildcard searches of files; you'll learn more about *glob* in Chapter 10.

You should be able to view the figure in most image viewers or your web browser.

■ **Note** *Matplotlib* returns objects as they're created. In the preceding example, the returned plot object is noted by the line, `[<matplotlib.lines.Line2D object at 0x000000000826FEF0>]`. Going forward, I'll omit these responses to make the interactive code easier to follow.

I called the function `savefig()` with the file name extension as part of the string holding the file name, instructing `savefig()` to create a PNG file. Similarly, I could've created a PostScript file by issuing `savefig('line.ps')`.

The dictionary object `FigureCanvasBase.filetypes` holds a list of supported file types in your system:

```
>>> from pprint import pprint
>>> pprint(FigureCanvasBase.filetypes)
```

```
{'eps': 'Encapsulated Postscript',
 'jpeg': 'Joint Photographic Experts Group',
 'jpg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'LaTeX PGF Figure',
 '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'}
```

FINDING WHAT YOU'RE LOOKING FOR IN COMPLEX MODULES

So how did I figure out that `FigureCanvasBase.filetypes` holds the supported file types? Did I read the entire manual? Hardly. Some of the packages we work with are really large, and mastering all the intricacies of variables and objects that control their behavior is not a trivial task. So I use some quick-and-dirty tricks. And although they might not be the "proper" way to do things, they help me get the job done, and that's what really counts. So let me show you what I've done to figure out the available file types.

To figure out the base file types, I issued `savefig()` with a bogus file format: `savefig('a.ext')`. The result was, of course, an exception (`ValueError`); however, I also received some useful information: `"Supported formats: eps, jpeg, jpg, pdf, pgf, png, ps, raw, rgba, svg, svgz, tif, tiff."` But that's not exactly what I wanted; I wanted an enumeration of the file formats, so I could index them rather than parse a string returned by an exception. So I traced back the source of the error from the exception output: the error originated in the file, `C:\Python33\lib\site-packages\matplotlib\backend_bases.py`, at line 2070. Next, I opened up the file `backend_bases.py`, jumped to line 2070, and I started reading the code. Python is a very readable language,

and it didn't take me all that long to figure out that the formats are available via `self.get_supported_filetypes()`, which basically just returns the property `filetypes`. Since `self` points to the container object, I scrolled up some more and found that the function `get_supported_filetypes()` is part of the class `FigureCanvasBase`–hence, `FigureCanvasBase.filetypes`.

Reading the exceptions generated by *matplotlib*, along with exploring modules and their names, also helped me find the objects `matplotlib.colors.cnames` and `matplotlib.colors.ColorConverter.colors`, both listing possible colors (see the section "Colors" later in the chapter). That said, reading the manual is also a very viable option.

You can pass an image format argument to `savefig()` to control the output file generated instead of specifying it in the file name string. You can also control other parameters, such as dots per inch (dpi) and face color for the color of the figure. A more general form of `savefig()` is `savefig(fname[, param1=value1][, param2=value2], ...)`. Table 6-1 lists some parameters. In the examples, assume `fn` is a string containing a file name.

Table 6-1. *savefig( ) Parameters*

Parameter	Description	Default Value	Example
`dpi`	Resolution in dots per inch	None	`savefig(fn, dpi=150)`
`facecolor*`	The figure's face color	`'w'` for white background	`savefig(fn, facecolor='b')`
`transparent`	Whether the figure is transparent.	`False`	`savefig(fn, transparent=True)`
`format`	File format	`'png'`	`savefig('image', format='pdf')`

** Refer to Table 6-4 for a list of color values.*

You can combine several parameters, as in this example:

```
savefig('file', dpi=150, format='png')
```

The function `savefig()` supports additional options; see `help(savefig)` for a full account.

■ **Note** Not all image formats support transparency, especially JPEG. If you require a transparent background, select an image format that supports transparency, such as PNG.

Plotting Graphs

This section details the building blocks of plotting graphs: the `plot()` function and how to control it to generate the output we require. We've used the `plot()` command extensively throughout the book. It's now time to examine it more closely.

The `plot()` function is highly customizable, accommodating various options, including plotting lines and/or markers, line widths, marker types and sizes, colors, and a legend to associate with each plot. The functionality of `plot()` is similar to that of MATLAB (`http://www.mathworks.com`) and GNU-Octave (`http://www.gnu.org/software/octave/`) with some minor differences, mostly due to the fact that Python has a different syntax from MATLAB and GNU-Octave.

Lines and Markers

We'll begin by creating a vector to plot using *NumPy* (see Chapter 7 for a full account of the *NumPy* package):

```
>>> figure()
>>> y = array([1, 2, -1, 1])
>>> plot(y)
>>> show()
```

If you don't have a GUI installed with *matplotlib*, replace `show()` with `savefig('filename')` and open the generated image file in an image viewer.

■ **Note** Going forward, I'll omit the `show()` call from listings. Be sure to issue `show()` or `savefig()` if you'd like to follow along.

I passed the vector `y` as an input to `plot()`. As a result, `plot()` drew a graph of the vector `y` using auto-incrementing integers for an x-axis. Which is to say that, if you don't supply x-axis values, `plot()` will automatically generate one for you: `plot(y)` is equivalent to `plot(range(len(y)), y)`. So let's supply x-axis values (denoted by variable `t`):

```
>>> figure()
>>> y = array([1, 2, -1, 1])
>>> t = array([10, 11, 12, 13])
>>> plot(t, y)
>>> show()
```

The call to function `figure()` generates a new figure to plot on, so we don't overwrite the previous figure.

Let's look at some more options. Next, we want to plot `y` as a function of `t`, but display only markers, not lines. This is easily done:

```
>>> figure()
>>> plot(t, y, 'o')
>>> show()
```

To select a different marker, replace the character `'o'` with another marker symbol. Table 6-2 lists some popular choices; issuing `help(plot)` provides a full account of the available markers.

Table 6-2. *A Sampling of Available Plot Markers*

Character	Marker Symbol
`'o'`	Circle
`'^'`	Upward-pointing triangle
`'s'`	Square
`'+'`	Plus
`'x'`	Cross (multiplication)
`'D'`	Diamond

Much as there are different markers, there are also different line styles, a few of which are listed in Table 6-3.

Table 6-3. *A Sampling of Available Plot Line Styles*

Character(s)	Line Style
`'-'`	Solid line
`'--'`	Dashed line
`'-.'`	Dash-dot line
`':'`	Dotted line

If you'd like both markers and lines, concatenate the symbols for line styles and markers. To plot a dash-dot line and diamond symbols as markers, issue the following:

```
>>> plot(t, y, 'D-.')
>>> show()
```

Figure 6-2 shows the output of the examples in this section.

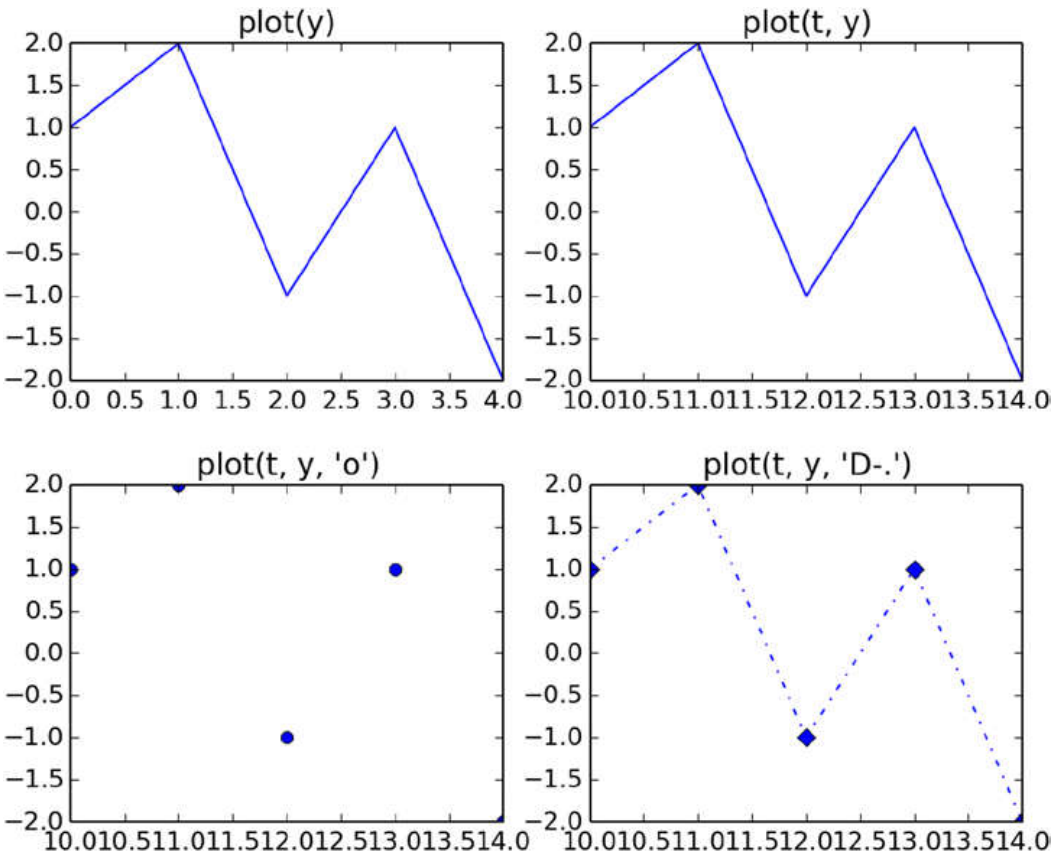

Figure 6-2. *Output of the previous examples: a line plot with no x values (top-left), a line plot with x values (top-right), a marker plot (bottom-left), and a marker and line plot (bottom-right)*

Plotting Several Graphs on One Figure

We use graphs to visualize data and compare it. What's more natural than displaying several graphs in one plot, so we can compare results? There are two ways to do that in *matplotlib*. The first way is to add more vectors to the `plot()` function:

```
>>> plot(t, y, t, 2*y)
```

You can also do so like this:

```
>>> plot(t, y, '+', t, 2*y, 's-')
```

The second way is by calling `plot()` repeatedly. Sometimes you might have only partial data to plot. Say you have vector `y`, but then you modify it and want to print both the original vector and the newly modified vector. What do you do? One option would be to store the intermediate value, but what if you have 20 of those? That would mean calling `plot()` with some 20+ arguments.

When you call `plot()` with an already existing figure, there are two possible outcomes. One is that the figure is erased, and the new plot is drawn. The other is that the figure is not erased, and the new plot is added to the figure. This behavior is determined by the hold status of the figure. You can control the hold status with the `hold()` function: calling `hold(True)` will ensure new plots don't erase the figure, whereas `hold(False)` will do the opposite. Issuing the command `hold()` with no arguments will toggle the hold status. As a general rule, it's best to specify the hold behavior you require and not rely on the default behavior; that is, `hold(True)` or `hold(False)`.

Line Widths and Marker Sizes

Next, we'll look at controlling line widths and marker sizes. We do so by passing `linewidth` (or `lw` for short) and `markersize` (or `ms` for short) arguments to `plot()`, as shown in Listing 6-1. Both arguments accept a floating point value; the default value is 1.

Listing 6-1. Plotting Several Lines in One Graph with Different Line Styles and Markers

```
I = arange(6)
plot(I, sin(I), 'o', I, cos(I), '-', lw=3, ms=8)
title("plot(I, sin(I), 'o', I, cos(I), '-', lw=3, ms=8)")
show()
```

Figure 6-3 shows the results of this example.

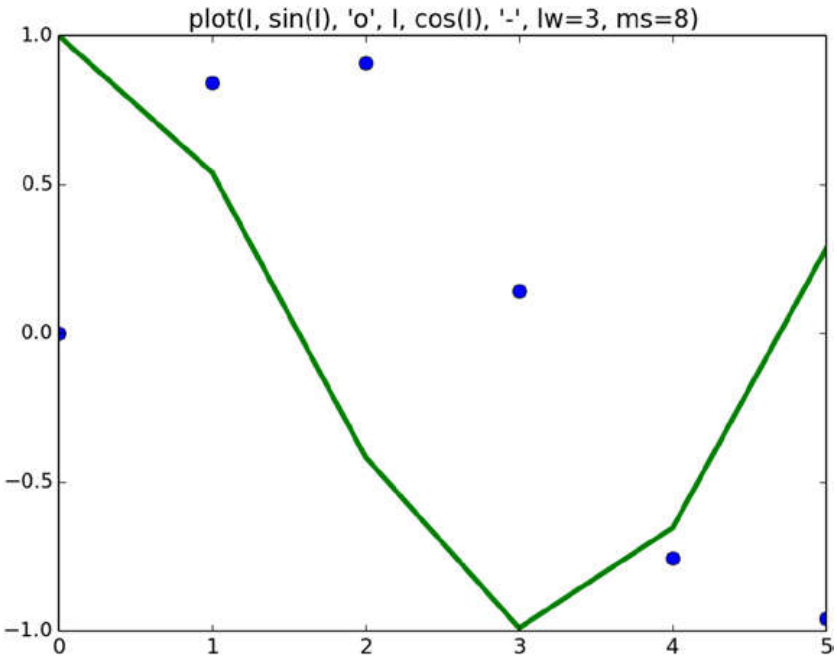

Figure 6-3. *Plotting several graphs in one figure*

When you plot multiple lines in one `plot()` function call, the parameters `linewidth` and `markersize` control *all* the lines in the same `plot()` command. If you'd like different lines with different line styles or different marker sizes in the same figure, draw each plot with an individual call to the `plot()` function and use the `hold()` function, as shown in Listing 6-2.

Listing 6-2. Different Line Widths in One Graph

```
figure(); hold(True)
I = arange(6)
plot(I, sin(I), lw=4)
plot(I, cos(I), lw=2)
show()
```

Colors

Last on our list of plotting basics is controlling color. Just as with marker style and line style, you can control color with one character, according to the list in Table 6-4.

Table 6-4. *The Color Character Look-Up Table*

Character	Color
'b'	Blue
'c'	Cyan
'g'	Green
'k'	Black
'm'	Magenta
'r'	Red
'w'	White
'y'	Yellow

As you might have noticed, *matplotlib* automatically chooses a different color for subsequent line plots if a color is not specified. You can select your preferred color by supplying a color character:

```
>>> figure()
>>> I = arange(6)
>>> plot(I, sin(I), 'k+-', I, cos(I), 'm:')
>>> show()
```

This will plot two lines: the first is a black line with plus markers and a connecting solid line, and the second is a magenta dotted line.

If you'd like a color that does not appear in Table 6-4, you can choose one from the dictionary object, `matplotlib.colors.cnames`. The dictionary contains a better color selection and has more than a hundred values. And lastly, if that dictionary is not enough, you can provide an explicit Red-Green-Blue (RGB) value. If you're using the dictionary values or an explicit RBG value, you have to provide the `color` argument as a parameter to a `plot()` call:

```
>>> plot(randn(5), 'y', lw=5) # 'y' from the color table
>>> plot(randn(5), color='yellowgreen', lw=5) # using matplotlib.colors.cname
>>> plot(randn(5), color='#ffff00', lw=5) # explicit yellow RGB
```

See `help(matplotlib.colors)` for additional color information.

In the preceding example, I've used the function `randn(n)` to generate a random vector of size 5.

Controlling the Graph

For a graph to convey an idea aesthetically, the data, although highly important, is not everything. The grid and grid lines, combined with a proper selection of axis and labels, present additional layers of information that add clarity and contribute to overall graph presentation.

Now that we have the basics of plotting lines and markers covered, let's turn to controlling the figure by controlling the x-axis and y-axis behavior and setting grid lines.

Axis

The `axis()` function controls the behavior of the x-axis and y-axis ranges. If you do not supply a parameter to `axis()`, the return value is a tuple in the form `(xmin, xmax, ymin, ymax)`. You can use `axis()` to set the new axis ranges by specifying new values: `axis([xmin, xmax, ymin, ymax])`. If you'd like to set or retrieve only the x-axis values or y-axis values, do so by using the functions `xlim(xmin, xmax)` or `ylim(ymin, ymax)`, respectively.

Other than the range limits, the function `axis()` also accepts the following values: `'auto'`, `'equal'`, `'tight'`, `'scaled'`, and `'off'`. The value `'auto'`—the default behavior—allows `plot()` to select what it thinks are the best values. The value `'equal'` forces each x value to be the same *length* as each y value, which is important if you're trying to convey physical distances, such as in a GPS plot. The value `'tight'` causes the axis to change so that the maximum and minimum values of x and y both touch the edges of the graph. The value `'scaled'` changes the x-axis and y-axis ranges so that x and y have both the same length (i.e., aspect ratio of 1). Lastly, calling `axis('off')` removes the axis and labels.

To illustrate these axis behaviors, I'll plot a circle (see Listing 6-3).

Listing 6-3. Plotting a Circle

```
R = 1.2
I = arange(0, 2*pi, 0.01)
plot(sin(I)*R, cos(I)*R)
show()
```

The reason I chose a circle of radius 1.2 is that, in the case of a radius with "nicer" numbers (e.g., 1.0 or 2.0), the automatic axis solution works very well, and it's hard to show the effects of the different axis options.

Figure 6-4 shows the results of applying different axis values to this circle.

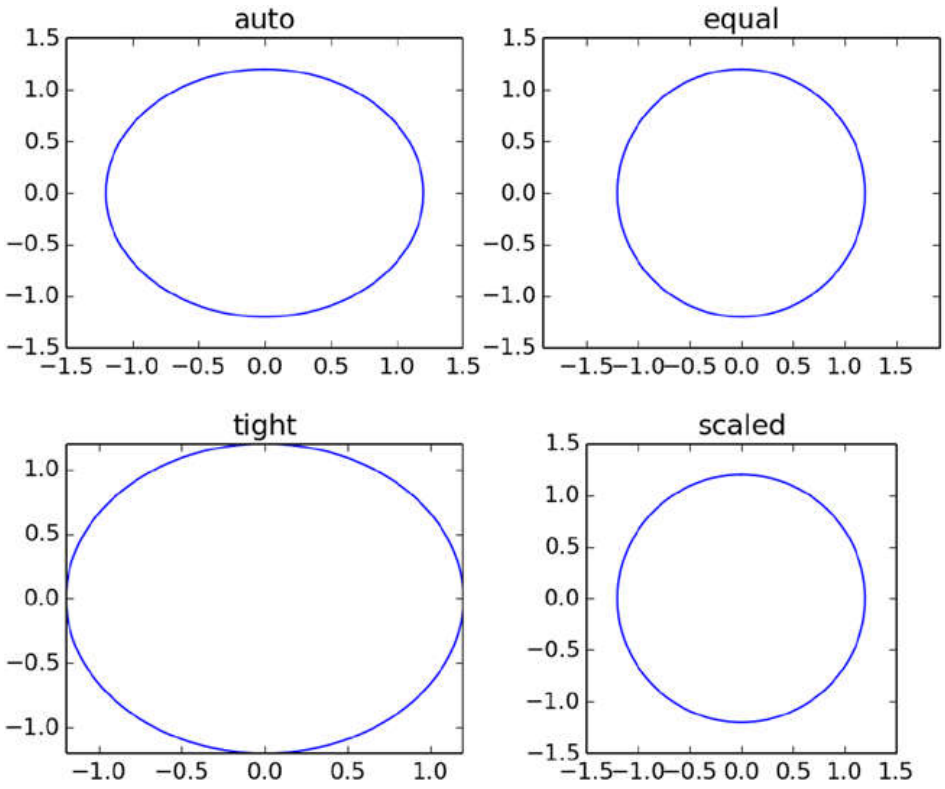

Figure 6-4. *Controlling axis behavior*

Grid and Ticks

The function `grid()` draws a grid in the current figure. The grid is composed of a set of horizontal and vertical dashed lines coinciding with the x ticks and y ticks. You can toggle the grid by calling `grid()` or set it to be either visible or hidden by using `grid(True)` or `grid(False)`, respectively.

To control the ticks (and effectively change the grid lines, as well), use the functions `xticks()` and `yticks()`, (see Listing 6-4 and Figure 6-5). The functions behave similarly to `axis()` in that they return the current ticks if no parameters are passed; you can also use these functions to set ticks once parameters are provided. The functions take an array holding the tick values as numbers and an optional tuple containing text labels. If the tuple of labels is not provided, the tick numbers are used as labels.

Listing 6-4. Grid and Tick Example

```
R = 1.2
I = arange(0, 4*pi, 0.01)
plot(sin(I)*R, cos(0.5*I)*R)
axhline(color='gray')
axvline(color='gray')
grid()
xticks([-1, 0, 1], ('Negative', 'Neutral', 'Positive'))
yticks(arange(-1.5, 2.0, 1))
show()
```

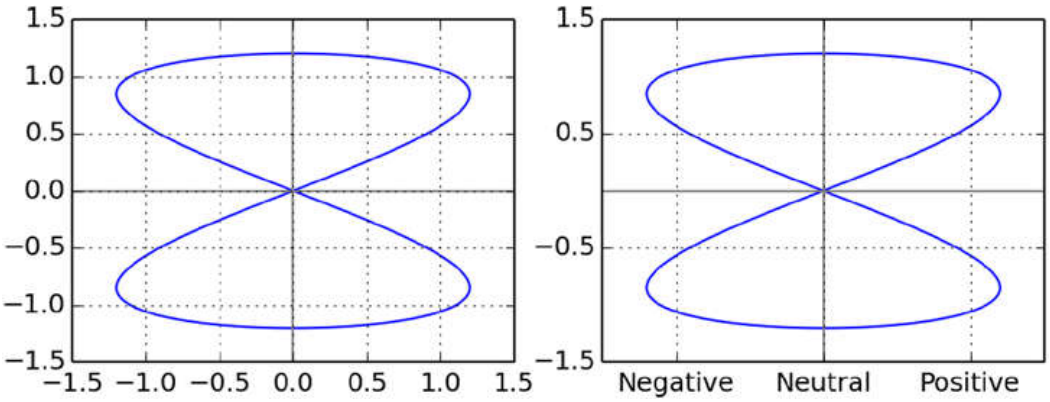

Figure 6-5. *Controlling the grid and axis: the left graph shows the default `xticks()`, while the right graph displays labels*

Figure 6-5 shows the output generated from Listing 6-4 without issuing the last two calls to `xticks()` and `yticks()` (left graph) and with `xticks()` and `yticks()` calls (right graph). Notice the labels on the x-axis.

I've also used the functions `axvline()` and `axhline()`, which plot a line across the x-axis and y-axis, respectively. The `axhline()` and `axvline()` functions accept many parameters, including `color`, `linewidth`, and `linestyle`, to name a few.

Subplots

In some of the previous figures in this chapter, I've displayed several smaller graphs in one figure; these are known as *subplots*. The `subplot()` function splits the figure into subplots and selects the current subplot. The subplots are numbered from left to right, top to bottom, so the upper-left subplot is 1, and the lower-right subplot is equivalent to the number of subplots. Notice that this is different from the default counting behavior used in Python: numbers start at 1 and not at 0.

To split the figure into 2-by-2 subplots and select the upper-left subplot for plotting, issue `subplot(2, 2, 1)`. Alternatively, you can pass the string `'221'`, which does the same thing: `subplot('221')`. It's also possible to combine subplots of different sizes in one figure. This is a bit tricky and requires subplotting with different subplot sizes. Listing 6-5 shows an example that generates a subplot on the upper half of the figure and two subplots on the lower part of the figure, the results of which you can see in Figure 6-6.

Listing 6-5. Subplots of Varying Sizes

```
figure()
subplot(2, 1, 1)
title('Upper half')
subplot(2, 2, 3)
title('Lower half, left side')
subplot(2, 2, 4)
title('Lower half, right side')
show()
```

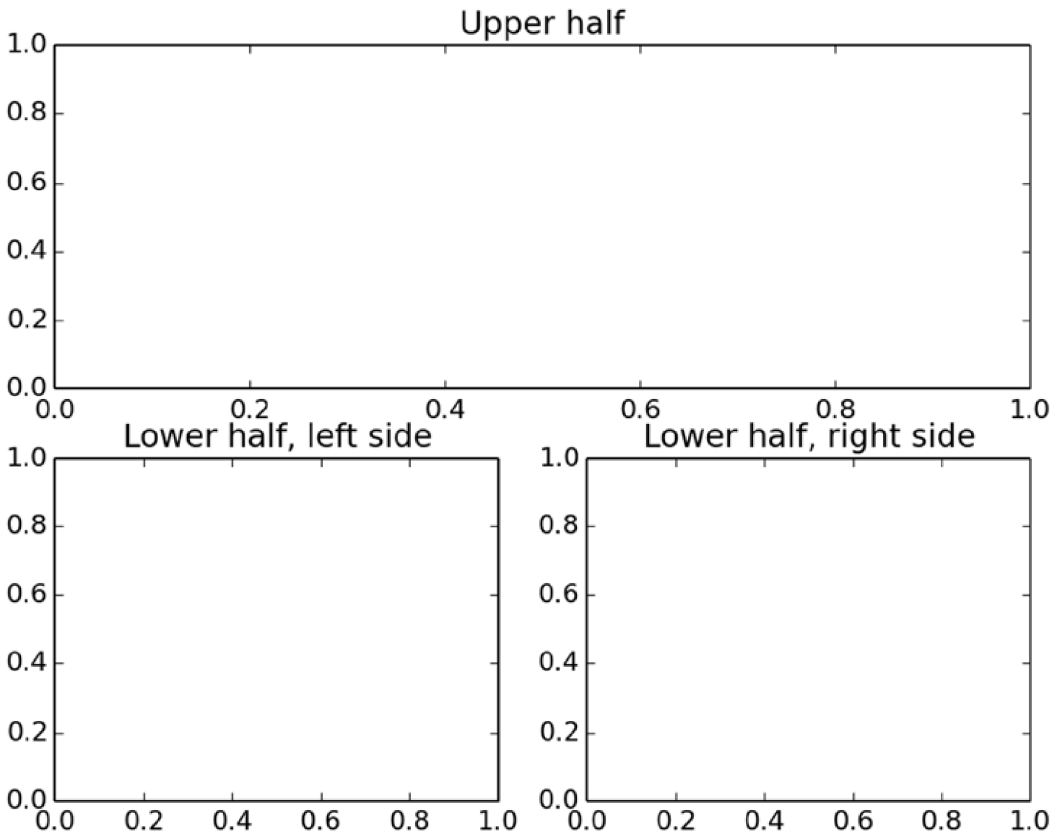

Figure 6-6. *Subplots of varying sizes*

■ **Tip** Subplots are especially useful in visualizing several aspects of the same data. For example, Figure 1-5 in the GPS example in Chapter 1 shows x and y coordinates in one subplot and velocity in another subplot. Events (e.g., speeding) are marked in both, providing a visual link between the two subplots.

Erasing the Graph

The functions `cla()` and `clf()` clear the axes and the figure, respectively. These functions are useful when you're working with an interactive environment and would like to clear the current axes (i.e., set the axes to default values and clear the plotted lines). It's also possible to clear the figure altogether, erasing also the axes and subplots, using the `clf()` function.

Lastly, you can choose to close the figure window altogether; this is done by calling the function `close()`. If you provide a number to `close()`, the figure associated with the number is closed. So, `close(1)` will close Figure 1, leaving the other figures open. If you'd like to close all the figures, issue `close('all')`.

Adding Text

There are several options to annotate your graph with text. You've already seen some, such as using the `xticks()` and `yticks()` function. The following functions will give you more control over text in a graph.

Title

The function `title(str)` sets `str` as a title for the graph and appears above the plot area. The function accepts the arguments listed in Table 6-5.

Table 6-5. *Text Arguments*

Argument	Description	Values
`fontsize`	Controls the font size	`'large'`, `'medium'`, `'small'`, or an actual size (i.e., `50`)
`verticalalignment` or `va`	Controls the vertical alignment	`'top'`, `'baseline'`, `'bottom'`, `'center'`
`horizontalalignment` or `ha`	Controls the horizontal alignment	`'center'`, `'left'`, `'right'`

All alignments are based on the default location, which is centered above the graph. Thus, setting `ha='left'` will print the title starting at the middle (horizontally) and extending to the right. Similarly, setting `ha='right'` will print the title ending in the middle of the graph (horizontally). The same applies for vertical alignment. Here's an example of using the `title()` function:

```
>>> title('Left aligned, large title', fontsize=24, va='baseline')
```

Axis Labels and Legend

The functions `xlabel()` and `ylabel()` are similar to `title()`, only they're used to set the x-axis and y-axis labels, respectively. Both these functions accept the text arguments from Table 6-5:

```
>>> xlabel('time [seconds]')
```

Next on our list of text functions is `legend()`. The `legend()` function adds a legend box and associates a plot with text:

```
>>> I = arange(0, 2*pi, 0.1)
>>> plot(I, sin(I), '+-', I, cos(I), 'o-')
>>> legend(['sin(I)', 'cos(I)'])
>>> show()
```

The legend order associates the text with the plot. Had I called `legend()` with the inverted list, the result would be a wrong legend.

An alternative approach is to specify the `label` argument with the `plot()` function call, and then issue a call to `legend()` with no parameters:

```
>>> I = arange(0, 2*pi, 0.1)
>>> plot(I, sin(I), '+-', label='sin(I)')
>>> plot(I, cos(I), 'o-', label='cos(I)')
>>> legend()
>>> show()
```

Figure 6-7 shows the addition of an x-axis label and legend.

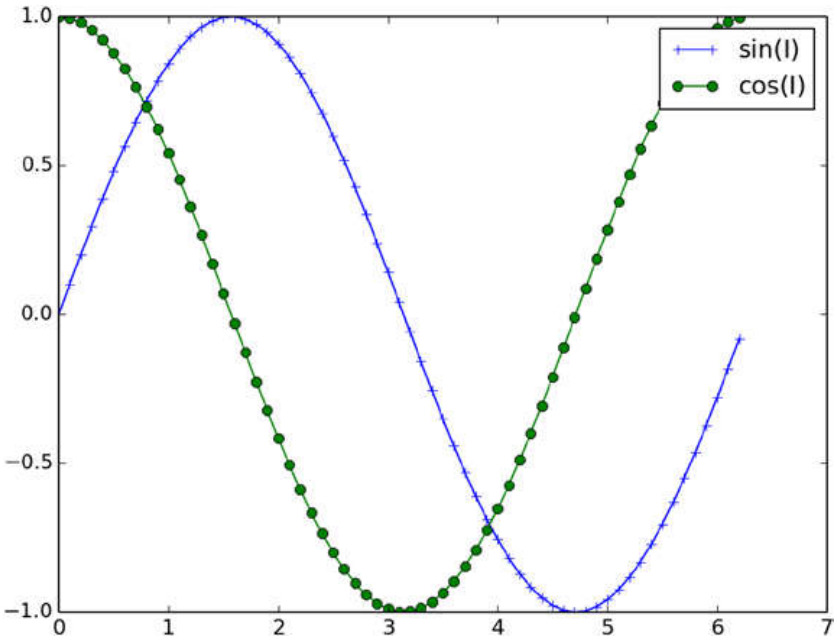

Figure 6-7. *Adding an x-axis label and legend*

You can also control the location of the legend box via the `loc` parameter. This is important if you don't want the legend text to hide the graph line. `loc` can take one of the following values: `'best'`, `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Instead of using strings, you can use numbers: `'best'` corresponds to `0`, `'upper left'` corresponds to `1`, and `'center'` corresponds to `10`. Using the value `'best'` moves the legend to a spot less likely to hide data; however, performance-wise there may be some impact.

The function `legend()` has additional options that let you add a drop shadow and control the spacing between the text within the legend, among other things. Consult with the interactive help for additional information.

Text Rendering

The `text(x, y, str)` function accepts the coordinates in graph units `x`, `y` and the string to print, `str`. It also renders the string on the figure. You can modify the text alignment using the arguments in Table 6-5. The following will print text at location (0, 0):

```
>>> figure()
>>> plot([-1, 1], [-1, 1])
>>> text(0, 0, 'origin', va='center', ha='center')
>>> show()
```

The function `text()` has many other arguments, such as `rotation` (which was used in Chapter 1) and `fontsize`. See `help(text)` for a complete list of arguments.

Mathematical Symbols and Expressions

Last on our list of text-related functions is one that renders mathematical symbols and expressions. The syntax to use mathematical symbols provided by *matplotlib* is similar to that of TEX. To render mathematical expressions, use a raw string and enclose your mathematical expression with $ signs. For Greek letters, start with a slash, followed by the name of the letter. So to print Alpha (α), your string should be `r'$\alpha$'`. Fractions can be created using the `\frac{num}{den}` notation; for example, `r'$\frac{\pi}{4}$'` is the symbol π divided by four. Subscripts are denoted with an underscore, so to render the text a_i, write `r'$a_i$'`.

It is beyond the scope of the book to cover the entire TEX syntax supported by *matplotlib*. For additional information, refer to the online *matplotlib* web site (at the time of the writing of this book, the following link was available: http://matplotlib.sourceforge.net/users/mathtext.html). That said, whenever you encounter a mathematical expression in this book, you're more than likely be able to figure out how it works with the small subset of commands presented in this section.

Example: A Summary Graph

The example script in Listing 6-6 summarizes the functions we've discussed up to this point: `plot()` for plotting; `title()`, `xlabel()`, `ylabel()`, and `text()` for text annotations; and `xticks()`, `ylim()`, and `grid()` for grid control.

Listing 6-6. A Plot Summary Example

```
I = arange(0, 2*pi+0.1, 0.1)
plot(I, sin(I), label='sin(I)')
title('Function f(x)=sin(x)')
xlabel('x [rad]', va='bottom')
ylabel('sin(x)')
text(pi/2, 1, 'Max value', ha='center', va='bottom')
text(3*pi/2, -1, 'Min value', ha='center', va='top')
xticks(linspace(pi/2, 2*pi, 4), (r'$\frac{\pi}{2}$', r'$\pi$', \
    r'$\frac{3\pi}{2}$', r'$2 \pi$'), fontsize=20)
xlim([0, 2*pi])
ylim([-1.2, 1.2])
grid()
show()
```

The result of this example appears in Figure 6-8.

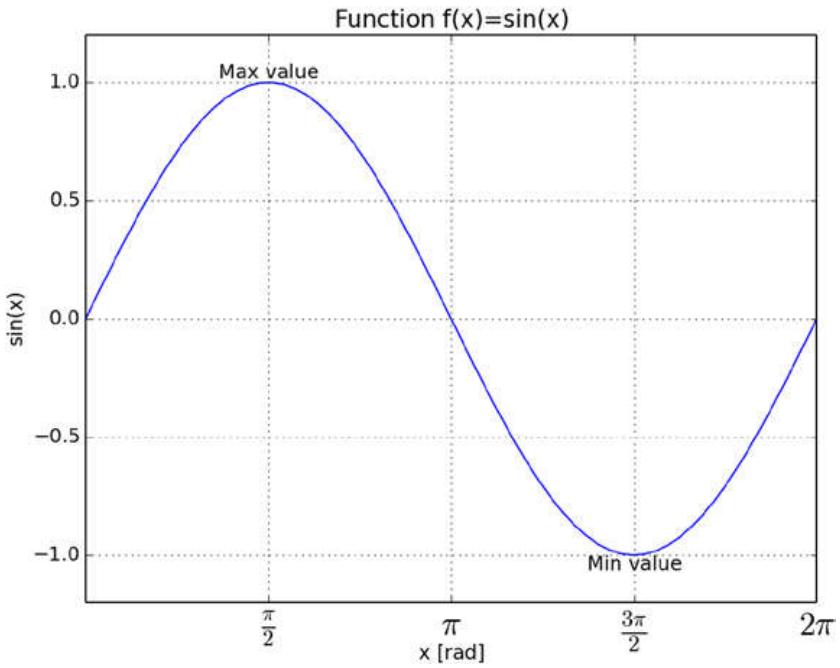

Figure 6-8. *A plot summary example*

More Graph Types

While the regular line and marker plots are excellent visualization tools, they're hardly the only ones. This section provides a quick overview of some other 2-D graph options.

Bar Charts

A favorite of many, a bar chart allows quantitative comparison of several values. To use a bar chart, call the function `bar(left, height)`, where `left` is the x-coordinates of the bar and `height` is the bar height. The function `bar()` allows for considerable customization; issuing `help(bar)` will provide most of the details.

Example: GDP, N Top Countries

For this example, which plots the purchasing power parity (GDP) of various countries, you'll need the CIA GDP Rank Order file, available from the CIA World Factbook (`https://www.cia.gov/library/publications/the-world-factbook/rankorder/rawdata_2001.txt`); this is a tab-delimited file, perfect for easy data processing. I'll assume that you've downloaded the file and saved it in folder `Ch6/data`; the source code resides in `Ch6/src`, and the output files are located in `Ch6/images`.

To begin, we'll define a function to read the data (we will use it in several examples in this chapter). The code in Listing 6-7 should be saved under the file, `src/read_world_data.py`.

Listing 6-7. Function read_world_data.py

```
import csv, re

def read_world_data(N=10, fn='../data/rawdata_2001.txt'):
    """A function to read CIA World Factbook file.

N is the number of countries to process."""

    # initialize return lists
    gdp, labels= [], []

    # read the data and process it
    for i, row in enumerate(csv.reader(open(fn), delimiter='\t')):
        # remove the dollar, comma and space characters
        gdp_value = re.sub(r'[\$, ]', '', row[2])

        # store data in billions of dollars
        gdp.append(float(gdp_value)/1e9)
        labels.append(row[1].strip())

        # stop analyzing the data after N countries have been processed
        if i == N-1:
           break
    return (gdp, labels)
```

The function reads data from the first N countries and returns their GDP alongside the country names. I've used two modules. The first, the *csv* module, reads the data, which is tab delimited. The second, the *re* module, gets rid of the dollar sign, comma, and space characters in the GDP value field.

Armed with `read_world_data()` function, we turn to plot the bar chart (see Listing 6-8).

Listing 6-8. Plotting the GDP Bar Chart

```
# a script to plot GDP bar chart
from pylab import *
from read_world_data import read_world_data

# initialize variables, N is the number of countries
N = 5

gdp, labels = read_world_data(N)

# plot the bar chart
bar(arange(N), gdp, align='center')

# annotate with text
xticks(arange(N), labels, rotation=-10)
for i, val in enumerate(gdp):
    text(i, val/2, str(val), va='center', ha='center', color='yellow')
ylabel('$ (Billions)')
title('GDP rank, data from CIA World Factbook')
show()
```

The script by now should be quite readable. Notice that I've imported the function `read_world_data()` from a separate file, `read_world_data.py`, which was defined in Listing 6-7. I've also chosen to rotate the text labels on the horizontal axis by 10 degrees in the call to `xticks()`. The rotation ensures that the labels for `'United States'` and `'European Union'` do not overlap.

Figure 6-9 shows our bar chart so far.

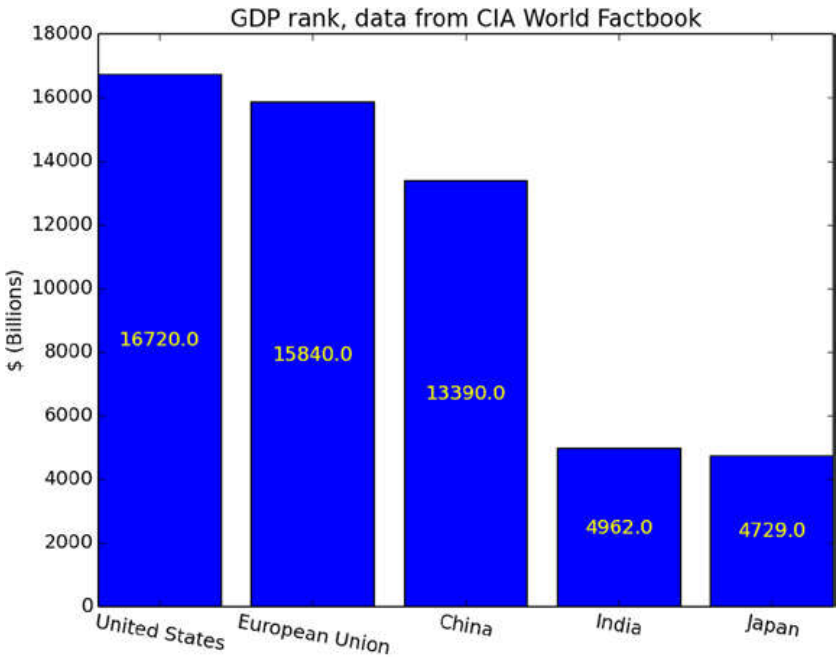

Figure 6-9. *A bar chart showing World GDP rank*

It's also possible to add error bars. To add an error bar equivalent to ±1000 billion dollars (talk about an error, eh?), add this line to the script shown in Listing 6-8, just after the `bar()` function call:

```
errorbar(arange(N), gdp, 1000*ones(N), color='k')
```

Finally, the function `barh()` plots a horizontal bar chart instead of a vertical one, should you require one (but in this case, remember to switch the x- and y-axis labels and ticks).

Histograms

Histograms are charts that show the frequency, or occurrence, of values. In *matplotlib*, the function `hist()` is used to calculate and draw the histogram chart. At a minimum, you must supply an array of values. You can control the number of cells in a histogram by specifying them as follows: `hist(values, numcells)`. Alternatively, you can specify the histogram bins `hist(values, bins)`, where `bins` is a list holding histogram bin values. The return value from `hist()` is a tuple of probabilities, bins, and patches. Patches are used to create the bars; I'll go into more detail in the "Patches" section later in the chapter.

The function `hist()` has other customization options, including the histogram orientation (vertical or horizontal), the alignment of bars, and more. Again, refer to the interactive help: `help(hist)`.

Example: GDP, Histogram

We turn again to the GDP ranks from the CIA World Factbook; this time we want to plot a histogram of the N largest economies. Again, we use the `read_world_data()` function implemented in the previous example (see Listing 6-9).

Listing 6-9. Plotting a GDP Histogram

```
# a script to plot GDP histogram
from pylab import *
from read_world_data import read_world_data

# initialize variables; N is the number of countries, B is the bin size
N, B = 20, 1000

gdp, labels = read_world_data(N)

# plot the histogram
prob, bins, patches = hist(gdp, arange(0, max(gdp)+B,B), align='left')

# annotate with text
for i, p in enumerate(prob):
    percent = '%d%%' % (p/N*100)
    # only annotate non-zero values
    if percent != '0%':
        text(bins[i], p, percent,
            rotation=45, va='bottom', ha='center')
ylabel('Number of countries')
xlabel('Income, billions of dollars')
title('GDP histogram, %d largest economies' % N)

# some axis manipulations
xlim(-B/2, bins[-1]-B/2)
show()
```

Figure 6-10 shows the resulting graph.

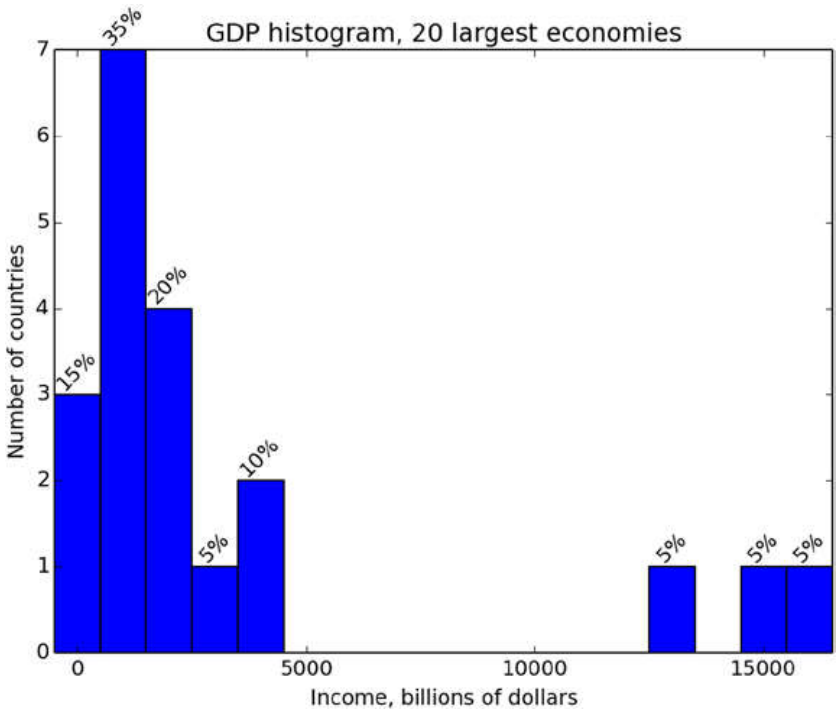

Figure 6-10. *A GDP histogram, with the N largest economies*

Again, the script should prove quite readable. I'd like you to turn your attention to what might appear an odd modification I've made to the x-axis using the call to function `xlim()`. The purpose of this call is to modify the default behavior of the x-axis ranges. The motivation behind this modification is that, since I've chosen `'center'` for the histogram bins, the automatic x-axis range includes negative values. This is because the leftmost bin is centered at zero, but has a width, part of it in the negative x-axis. I didn't like this behavior and chose to override it by manually setting the axis. Instead of setting a fixed number, I've modified the x-axis by subtracting half the bin width, `B/2`, from the axis.

As a general rule, when you modify default behavior like this, try to use parameters as much as possible (in the preceding example, you would use the parameter `B`, not the value `1000`, and retrieve current values with `xlim()`); this will allow for more flexible scripts that cater to a wider range of input values.

Pie Charts

Pie charts are as simple to use as bar charts. The function that implements pie charts is `pie(x)`, where `x` holds the values to be charted.

Example: GDP, Pie Chart

Listing 6-10 presents a script to generate a pie chart, shown in Figure 6-11; again, this example uses the function, `read_world_data()`.

Listing 6-10. Plotting a GDP Pie Chart

```
# a script to plot GDP pie chart
from pylab import *
from read_world_data import read_world_data
```

```
# initialize variables, N is the number of countries
N = 10

gdp, tags = read_world_data(N)

# plot the bar chart
pie(gdp, labels=tags)
axis('equal')
title('GDP rank, data from CIA World Factbook')
show()
```

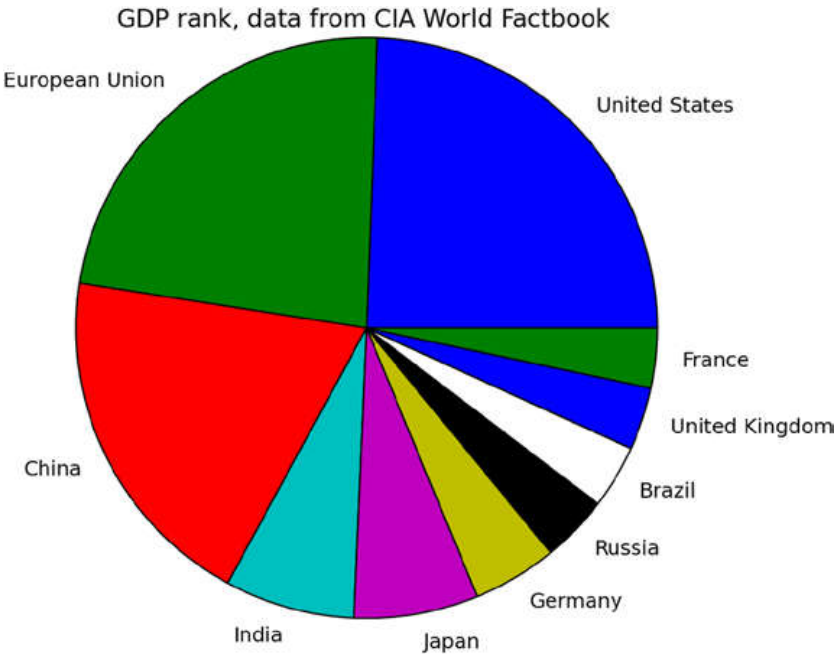

Figure 6-11. *A GDP pie chart, with the N largest economies*

I decided to use the variable `tags` instead of `labels`, so that the call to `pie()` would be a little less confusing. Had I stuck with the original name, `labels`, the call to pie would've been `pie(gdp, labels=labels)`. This still would've worked, but it would seem a bit confusing, in my opinion.

Logarithmic Plots

The functions `semilgox()` and `semilogy()` are used to plot the x-axis and y-axis in a logarithmic scale, respectively. Logarithmic plots of type `semilogy()` are common when plotting power or intensity values, such as those of the Richter magnitude scale, which measures seismic energy. Likewise, measurements of quantities used with frequencies, for example, are commonly plotted on a logarithmic x-scale denoting octaves and decades. There's also the option of using a `loglog()` plot, which means that both the x-axis and y-axis are logarithmic. This is the case for Bode plots, which are common in engineering fields.

All three functions, `semilogx()`, `semilogy()`, and `loglog()`, can be modified with arguments similar to those presented with the `plot()` function.

The function `logspace(start, stop, numpoints=50, endpoint=True, base=10.0)` can be useful in creating a range of values to be plotted with the preceding functions. The `start` and `stop` values are the exponent values. The function `logspace()` generates logarithmically spaced values between 10^{start} and 10^{stop}. You can decide whether the end value, 10^{stop}, is returned by specifying `endpoint = True`. If you'd like a base other than 10, set `base` to the value you require:

```
>>> figure()
>>> I = 2*logspace(1, 5, 5)
>>> I
```

```
array([  2.00000000e+01,   2.00000000e+02,   2.00000000e+03,
         2.00000000e+04,   2.00000000e+05])
```

```
>>> semilogx(I, [20, 19, 8, 2, 2], '+-', lw=2)
>>> title('Logarithmic plot, semilogx()')
>>> xlabel('Frequency [Hz]')
>>> ylabel('Amplitude [dB]')
>>> grid()
>>> show()
```

Figure 6-12 shows the results of the preceding example.

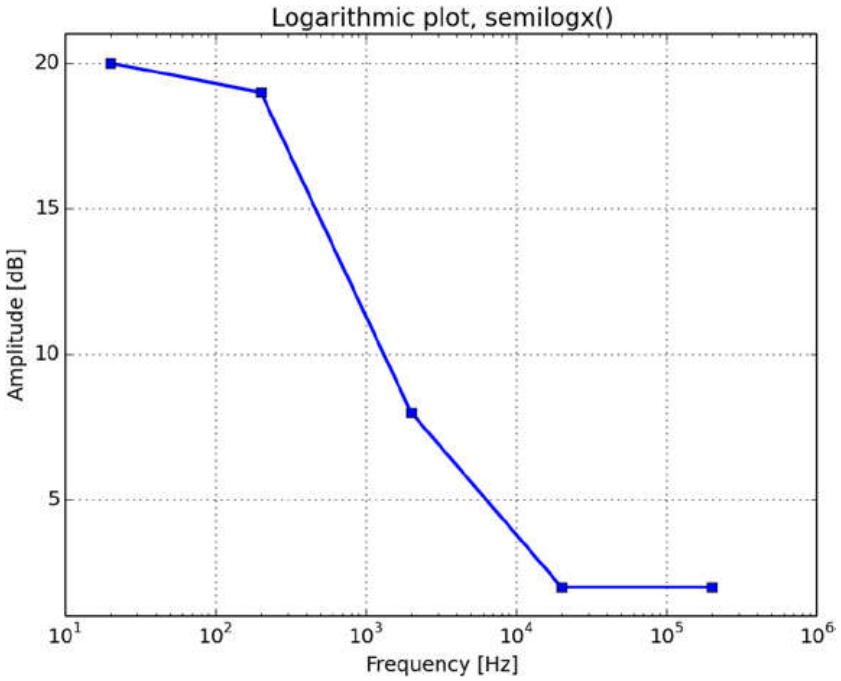

Figure 6-12. *A logarithmic plot*

Notice that, when plotting with `semilogx()`, `semilogy()`, and `loglog()`, the labels are the original values, not the logarithms of the values. If you'd like to print the logarithmic values, you should probably use a regular `plot()` function with `log()` or `log10()` of the values. This is useful, for example, in estimating the energy in decibels (dB):

```
>>> def db(x):
...     """Returns the value of x, in decibels."""
...     return 20*log10(abs(x))
...
>>> plot(db(array([1000, 980, 970, 400, 30, 2, 1, 1]))
>>> title('Logarithmic plot, dB)')
>>> xlabel('Frequency [Hz]')
>>> ylabel('Amplitude [dB]'
>>> xticks(arange(8), 10**arange(8))
>>> grid()
>>> show()
```

Polar Plots

Polar plots draw polar coordinate values: a radius at a given angle. Polar plots are commonly used to draw antenna radiation patterns, as they depict the energy the antenna transmits at any given angle. Polar plots are implemented using the `polar(theta, r)` function.

To set the labels along the radius, use the `rgrids(radii, labels)` function, which works similarly to `xticks()` and `yticks()`. If you don't provide the `labels` value, the `radii` values are used as labels. You can also set the `angle` at which the labels are plotted (the default is 22.5 degrees). Similarly, the function `thetagrids()` plots the angle ticks and labels, as demonstrated in Listing 6-11.

Listing 6-11. A Polar Plot

```
theta = arange(0, 2*pi, 0.01)
polar(theta, cos(theta), theta, -cos(theta))
rgrids([0.5, 1.0], ['Half', 'Full'])
theta_labels = ['0', r'$\frac{\pi}{2}$', r'$\pi$', r'$\frac{3\pi}{2}$']
thetagrids(arange(0, 360, 90), theta_labels, fontsize=20)
title(r'A polar plot of $\pm cos(\theta)$', va='bottom')
show()
```

Figure 6-13 shows the resulting polar plot.

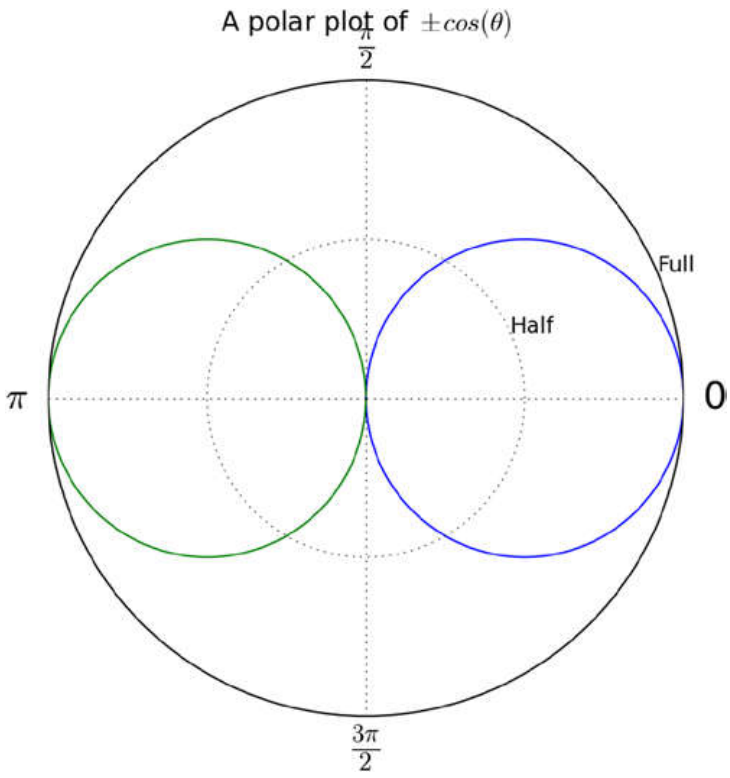

Figure 6-13. *A polar plot*

In the title, I've used the ± symbol denoted by `'$\pm$'`.

Stem Plots

Stem plots draw a vertical line from (x, 0) to (x, y) for every (x, y) value, as well as a marker at (x, y). Stem plots are used to denote discrete data and are popular for plotting filtering windows (see Listing 6-12).

Listing 6-12. A Stem Plot of Filter Windows

```
from pylab import *
N = [4, 8, 16, 64]
for i, n in enumerate(N):
    subplot(2, 2, i+1)
    stem(arange(n), hamming(n))
    xticks(arange(0, n+1, n/4))
    yticks([0, 0.5, 1])
    xlim(-0.5, n+0.5)
    title('N=%d' % n)
show()
```

Figure 6-14 shows the results of this listing.

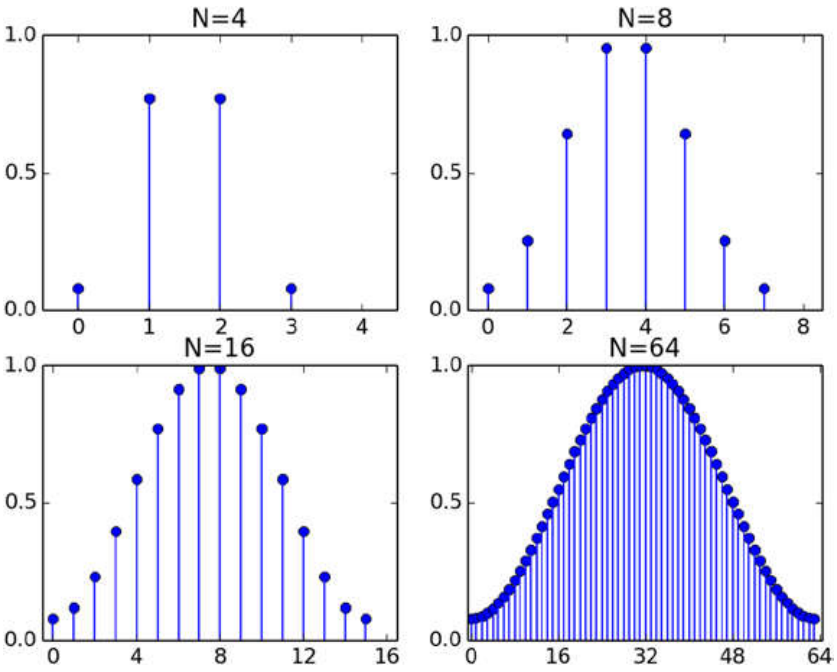

Figure 6-14. *Stem plots of a Hamming window with different N values*

In the preceding example, I've used the `hamming()` function to create a Hamming window, commonly used in filtering values.

Additional Graphs

Matplotlib also supports a great number of graphs used to depict more complex data. Here's a short list of some of the graphs available:

- The functions `contour()` and `contourf()` are used for contour plots. Contour plots draw a line connecting equal (x, y) value pairs. They're used in weather maps, where they detail lines of equal pressure or temperature. In topographical maps, they detail the terrain; and more.
- The function `specgram()` displays the frequency contents of data over time. `specgram()` can be used, for example, to plot the frequencies of a sound wave as a function of time.
- Both the `contour()` and `specgram()` functions rely on a color map to depict the data. Color maps are a relation between a value and a color. *Matplotlib* provides a set of color maps that include such names as `autumn()` and `hot()` to ease the selection of a color map.
- The function `quiver()` implements quiver plots, which are typically used to describe fields in physics. The quiver plot is a set of arrows that depict the force at each point (direction and magnitude).

Example: Plotting the Frequency Content of a Signal

Sometimes, it's of value to plot the frequencies a signal is composed of as a function of time. For example, in an audio signal, a different frequency means a different note, so plotting frequencies as a function of time is a possible "musical visualization."

In this example, shown in Listing 6-13, we create a signal composed of several discrete frequencies and display those frequencies as a function of time using a `specgram()`.

Listing 6-13. A Specgram of a Signal

```
from pylab import *

Fs = 256
times = [3, 7, 5]
frequencies = [100, 20, 80]

y = array([])
for t, f in zip(times, frequencies):
    x = cos(2*pi*arange(t*Fs)/Fs*f)
    y = append(y, x)

specgram(y, 256, Fs)
xlabel('Time [sec]')
ylabel('Frequency [Hz]')
axis([0, 14.5, 0, 128])
show()
```

I set the frequency of sampling at 256 samples per second and created a signal composed of 100 hertz (Hz) for 3 seconds, 20 Hz for 7 seconds, and then 80 Hz for 5 seconds. I then plotted the signal using `specgram()`, with the results shown in Figure 6-15.

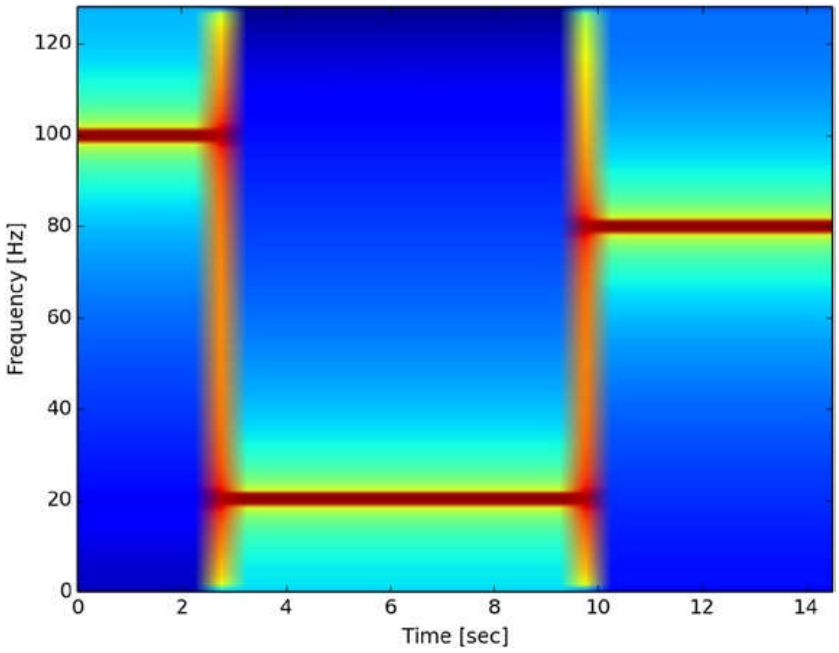

Figure 6-15. *A specgram*

Figure 6-15 clearly shows that in the first 2 seconds, the frequency is 100 Hz; in the next 8 seconds, the frequency is 20 Hz; and in the last 5 seconds, the signal's frequency is 80 Hz.

■ **Note** If you look closely at the figure, you'll notice there's half-a-second shift in the specgram. This is due to an overlapping window of size 128 samples. See `help(specgram)` for information on the overlapping window.

You can change the colors used to display the specgram using a color map function. Simply issue `hot()` or `autumn()` at the end of the script, and then observe the results. See `help(colormaps)` for a full account of available colormaps.

Example: A Repelling Force Field

The following example illustrates the use of `quiver()` to depict a force field. At each point in the figure, an arrow points at the direction of the acting force, as well as its magnitude, which is denoted by the size of the arrow:

```
from pylab import *

x = arange(-5, 6, 1)
y = arange(-5, 6, 1)

u, v = meshgrid(x, y)
quiver(u, v)

xticks(range(len(x)), x)
yticks(range(len(y)), y)

axis([-1, 11, -1, 11])
axis('scaled')
title('A repelling force field!')
show()
```

I used the function `meshgrid(x, y)`, which generates two matrices: the first is a matrix of repeating values of `x`, and the second is a matrix of repeating values of `y`. The output is used to plot the quiver, shown in Figure 6-16. I then updated the axis to reflect the proper ranges.

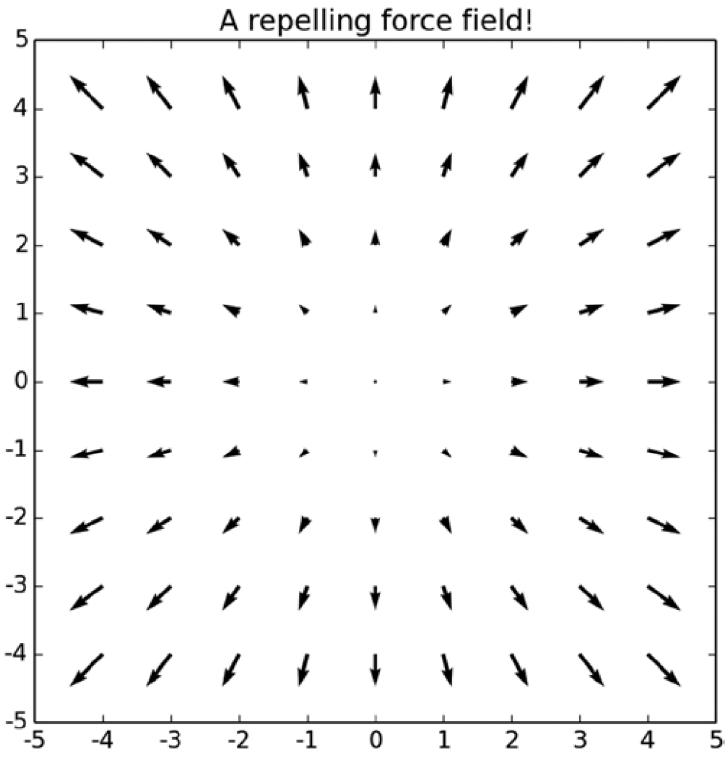

Figure 6-16. *A quiver plot depicting a force field*

Example: A Contour Plot

A contour plot (also known as a contour map) is a plot composed of contour lines. Each contour line joins points of equal value. Examples of contour plots include topographical maps that describe valleys and hills, and meteorological maps that show isobars (lines of equal pressure).

To generate contour plots, you need a matrix corresponding to the function z=f(x, y), where x and y are the independent variables (e.g., latitude and longitude on topographical maps), and z is the function value (e.g., height, in topographical maps).

In the following example, I generate values of x from -5 to 5 with a call to `x = arange(-5, 5, 0.1)`. I do the same with y values. However, the function z=f(x, y) requires a matrix, whereas `x` and `y` are vectors. So the next step involves creating a grid from `x` and `y`. This is done with a call to `u, v = meshgrid(x, y)`; `u` corresponds to the `x` values, while `v` corresponds to the `y` values. The beauty of using a grid is that now I can write functions as I normally would—in math. For example, a unit sphere's equation is $x^2+y^2+z^2=1$, and the half top unit sphere is $z=\sqrt{(1-x^2-y^2)}$. This translates to `z = sqrt(1-u**2-v**2)` in Python, but you'll have to take care of the square root of negative values (see the "Example: A Non-Rectangular Contour Plot" section later in this chapter). In the contour example, I've chosen an interesting function: `z = 2*u**2+v**2-u*v+10*u`. Listing 6-14 shows the code and the resulting contour plot.

Listing 6-14. A Contour Map

```
from pylab import *

x = arange(-5, 5, 0.1)
y = arange(-5, 5, 0.1)
u, v = meshgrid(x, y)
```

```
z = 2*u**2+v**2-u*v+10*u
contour(x, y, z, 10)
title('A contour map')
show()
```

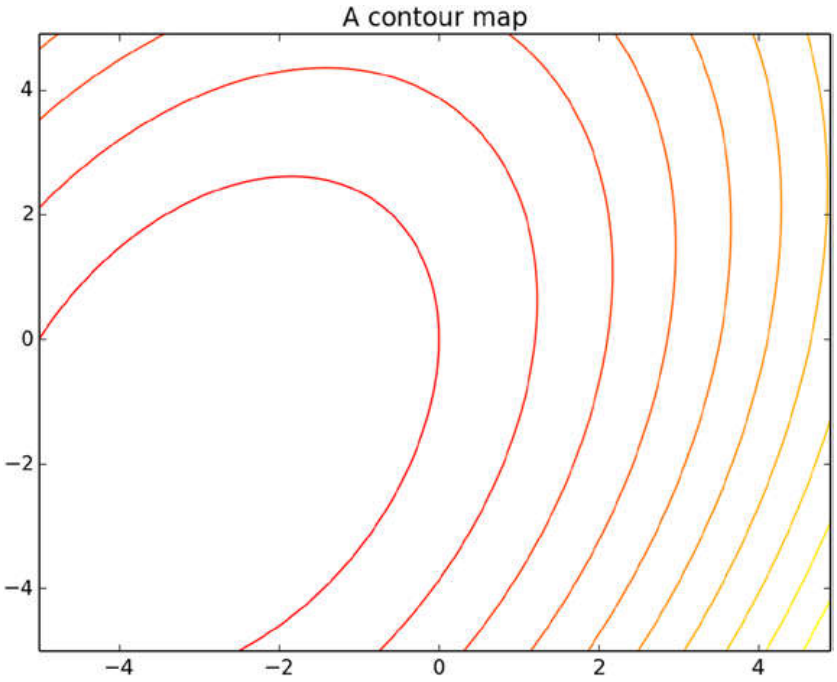

Figure 6-17. *A contour map of the function, $z=2x^2+y^2-xy+10x$*

Example: A Non-Rectangular Contour Plot

What if your grid data in the preceding example is not a rectangle? Suppose it's a trapezoid or a circle. In that case, how would you go about generating a contour plot? The trick is to generate an enclosing rectangular grid and set the values that are outside your region of interest to `None`.

The code in Listing 6-15 generates a filled contour plot using the function `contour()`. The listing shows both a rectangular grid and a non-rectangular grid of the contours of a sphere. Negative values of $z^2=1-x^2-y^2$ are outside the sphere, and they are either set to zero or not plotted.

Listing 6-15. A Filled Contour Map of a Sphere with a Rectangular Grid and a Non-Rectangular Grid

```
from pylab import *

x = linspace(-1.5, 1.5, 1000)
y = linspace(-1.5, 1.5, 1000)
u, v = meshgrid(x, y)

subplot(2, 2, 1)
z = 1-u**2-v**2
z[nonzero(z<0)] = 0
contourf(x, y, sqrt(z), 10)
```

```
title('A contour map')
axis('equal')
axis('off')

subplot(2, 2, 2)
z = 1-u**2-v**2
z[nonzero(z<0)] = None
contourf(x, y, sqrt(z), 10)
title('Non-rectangular contour map')
axis('equal')
axis('off')
show()
```

Notice how I've used the function `nonzero()` to set the values of `z` to either `0` or `None` per the required grid.

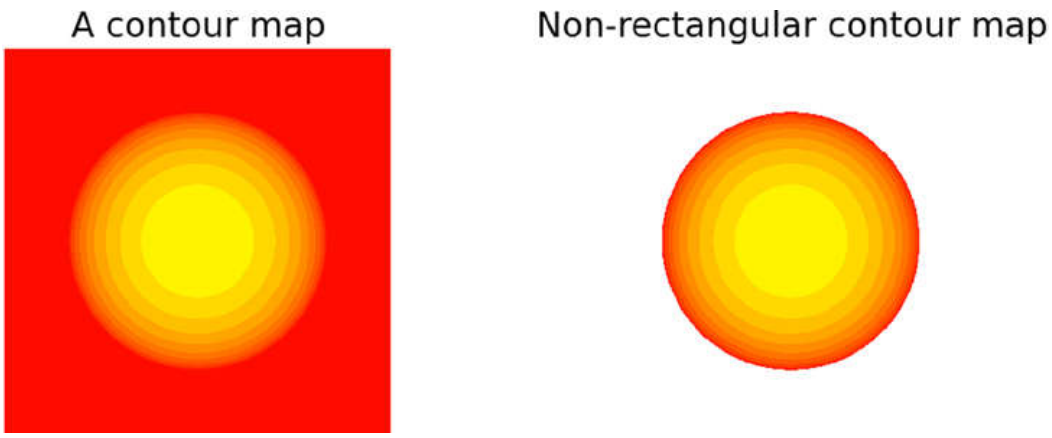

Figure 6-18. *A filled contour map of a sphere with a rectangular grid (left) and a non-rectangular grid (right)*

Getting and Setting Values

As you start plotting and generating visual output, you'll find that you're using more and more of the "helper" functions, functions that don't necessarily plot the data, but instead control the graph behavior and arrange labels just the way you want them.

So far we've used two methods to modify a plot behavior. One used dedicated functions like `axis()`, `xlim()`, and `ylim()` to control the plot ranges. The other method passed arguments to functions, such as the `rotation` argument in the `text()` function.

A third method is available, one that uses the object-oriented design of *matplotlib*. It involves two functions, `setp()` and `getp()`, that retrieve and set a *matplotlib* object's parameters. The benefit of using `setp()` and `getp()` is that automation is easily achieved.

Up to this point, we've suppressed the output from *matplotlib*, so that the interactive scripts would be easier to follow. But now let's look at those outputs. Whenever you issue a `plot()` command, *matplotlib* returns a *list* of *matplotlib* objects. This is important; the return value from calling `plot()` is a list of objects, not the *matplotlib* object itself, even if you only have one line to plot:

```
>>> from pylab import *
>>> p = plot(arange(5))
```

```
>>> type(p)
```

```
<class 'list'>
```

```
>>> type(p[0])
```

```
<class 'matplotlib.lines.Line2D'>
```

The function `setp(matobj)` prints a list of properties you can set for `matobj`, where `matobj` is a *matplotlib* object. The function accepts either a list of *matplotlib* objects or just one object:

```
>>> setp(p[0])
```

```
agg_filter: unknown
alpha: float (0.0 transparent through 1.0 opaque)
animated: [True | False]
antialiased or aa: [True | False]
axes: an :class:`~matplotlib.axes.Axes` instance
clip_box: a :class:`matplotlib.transforms.Bbox` instance
clip_on: [True | False]
```

(Note that additional output of `setp()` was suppressed.)

If you're not sure of what values a parameter can take, issue `setp(p, 'param')`:

```
>>> setp(p[0], 'visible')
```

```
visible: [True | False]
```

To hide the plot, you could issue the following:

```
>>> setp(p[0], visible=False)
```

Or you could set the label associated with a line by issuing the following:

```
>>> setp(p[0], label='Line1')
>>> legend()
```

The function `setp()` also accepts lists of *matplotlib* objects, in which case all the *matplotlib* objects in the list will be set.

■ **Note** To query acceptable parameters, enclose the parameter to be queried in quotes: `setp(p, 'linewidth')`. To set a parameter value, do not include the quotes, but do use an assignment: `setp(p, linewidth=4)`.

Similarly, to retrieve values, use the `getp()` function. The function `getp()` is a little less forgiving in that it requires one *matplotlib* object, not a list of objects. The variable `p` that we've been using up to this point is really a list of one *matplotlib* object, so we need to index it:

```
>>> getp(p[0], 'linewidth')
```

```
1.0
```

Setting Figure and Axis Parameters

In the preceding examples, we stored the return value from the call to the function `plot()`, which is a *matplotlib* object of a line. Specifically, we stored the line we drew—well, a list containing one line. But how do we modify the behavior of the figure or the axis?

The function `gcf()` returns a handle to the current figure. The function `gca()` returns a handle to the current axis. Armed with these, we can now modify the axis and figure parameters.

To set the y label, instead of calling `ylabel('Y value')`, we could issue this command:

```
>>> setp(gca(), ylabel='Y value')
```

But what are the benefits of using `setp()` in this manner over simply calling `ylabel()`? The answer is automation. Let's turn to an example.

Example: Modifying Subplot Parameters

Suppose you'd like to write a function that receives a figure number and then modifies all the subplot titles in the figure (if they exist) to numbered titles. For example, for a figure of 2-by-2 subplots in use, you'd like the subplot titles to be from 1 to 4 (if they all exist). In this case, you don't know in advance how many subplots are in a figure.

This is an ideal case for using `setp()` and `getp()`, as demonstrated in Listing 6-16.

Listing 6-16. Numbering Subplots

```
from pylab import *
def number_subplots(fignum):
    """Numbers the subplots in a figure."""

    # switch to the requested figure
    figure(fignum)

    fig = gcf()

    for i, fig_axe in enumerate(getp(fig, 'axes')):
        fig_axe.set_title(str(i+1))

    axis()
show()
```

There are a few things to keep in mind regarding the function, `number_subplots()`. First, we set the focus to the figure we'd like to work on by calling `figure(fignum)`. Next, we retrieve a handle to the figure with `gcf()`. The following step assumes some knowledge of the *matplotlib* object structure. But even if you're not familiar with

the structure, it's pretty simple to figure out what's going on by exploring the objects. To illustrate this, create a simple figure with two empty subplots:

```
>>> figure()
>>> ax1 = subplot(2, 1, 1)
>>> ax2 = subplot(2, 1, 2)
```

Now retrieve the current figure properties with `getp(gcf())`:

```
>>> getp(gcf())
```

```
agg_filter = None
alpha = None
animated = False
axes = [<matplotlib.axes.AxesSubplot object at 0x00000000...
children = [<matplotlib.patches.Rectangle object at 0x0000000...
```

(I've removed the extra output lines as they're not important for the discussion.)

Now look closely at two properties: `axes` and `children`. The parameter `axes` hold a list of values, and the parameter `children` holds another list of values. Further examination shows that the `axes` objects are all contained within the `children` values. In reality, these are the two axes for the two subplots. So, to get a list of these, we can simply call `getp(gcf(), 'axes')`, as the code indeed does. We then set the titles and call the `axis()` function to force a redraw.

There's a caveat in the implementation of the function `number_subplots()`: numbering is performed in accordance with the creation of the subplots. That is, if the bottom-left subplot was created before the top-left subplot, it will have the smaller title value associated with it and not the regular subplot numbering (left to right, top to bottom). If you'd like to change this, you'll have to look at the positions of the subplots and assign numbers accordingly. This is somewhat more complex and not all that educational, so I've opted to leave it out of the discussion.

A lot of the parameters that are accessible via `setp()` and `getp()` are also accessible by means of dedicated functions. Instead of setting the y-axis label parameter with `setp()`, you can call the `ylabel()` function. When possible, I prefer using the function version over `setp()` and `getp()` because I think it's easier to follow.

Exploring the *matplotlib* object through the `dir()` built-in function is also a very good method for probing the capabilities of a *matplotlib* object. Most of the functions are self-explanatory and let you set and retrieve values associated with a *matplotlib* object. If you're not sure, use the `help()` function in an interactive Python session. Based on a partial comparison, I've determined that *matplotlib* object methods are equivalent to the properties available with `getp()` and `setp()`. This means you can use either:

```
>>> matobj = gcf()
>>> [func for func in dir(matobj) if func.startswith('get')]
```

```
['get_agg_filter', 'get_alpha', 'get_animated', 'get_axes', 'get_children',
'get_clip_box', 'get_clip_on', 'get_clip_path', 'get_contains',
'get_default_bbox_extra_artists', 'get_dpi', 'get_edgecolor', 'get_facecolor',
'get_figheight', 'get_figure', 'get_figwidth', 'get_frameon', 'get_gid',
'get_label', 'get_path_effects', 'get_picker', 'get_rasterized',
'get_size_inches', 'get_sketch_params', 'get_snap', 'get_tight_layout',
'get_tightbbox', 'get_transform', 'get_transformed_clip_path_and_affine',
'get_url', 'get_visible', 'get_window_extent', 'get_zorder']
```

A final note: working with `setp()` and `getp()` or the set and get methods of the *matplotlib* object is an advanced topic. These functions allow more exact control of the behavior of plots and graphs, but they are not easy to master. Moreover, they require a good understanding of the *matplotlib* object hierarchy. Regardless of the complexity, I believe this is an important concept. As you draw more graphs and deal with more data, you'll find that the default functionality, although great, isn't *exactly* what you want. And in these cases, turning to `setp()` and `getp()` is a good option. At this point, I hope you've seen enough of their benefits and how to use them that you will feel comfortable experimenting on your own.

Patches

So far we've worked with text and lines, which are both implemented as *matplotlib* objects. But sometimes those two objects are not enough. A third object, the patch, allows us to draw other types of shapes that don't necessarily fall under the category of a line or text.

To work with patches, you assign them to an already existing graph because, in a sense, patches are "patched" on top of a figure. Table 6-6 gives a partial listing of available patches. In this table, the notation `xy` indicates a list or tuple of (x, y) values.

Table 6-6. *Available Patches*

Patch	Description
`Arrow(x, y, dx, dy)`	An arrow, starting at location `(x, y)` and ending at location `(x+dx, y+dy)`
`Circle(xy, r)`	A circle centered at `xy` and radius `r`
`Ellipse(xy, w, h, angle)`	An ellipse centered at `xy`, of width `w`, height `h`, and rotated `angle` degrees
`Polygon([xy1, xy2, xy3,...])`	A polygon made of vertices specified by `xy` points
`Wedge(xy, r, theta1, theta2)`	A wedge (part of a circle) centered at `xy`, of radius `r`, starting at angle `theta1` and ending at angle `theta2`
`Rectangle(xy, w, h)`	A rectangle, starting at `xy`, of width `w` and height `h`

To use patches, follow these steps:

1. Draw a graph.
2. Create a patch object.
3. Attach the patch object to the figure using the `add_patch()` function.

■ **Note** This might seem like considerable effort to add an arrow patch, for example; however, these three steps can be folded neatly into one line. To draw an arrow from (0, 0) to (1, 1), issue `gca().add_patch(Arrow(0, 0, 1, 1))`.

Example: Adding Arrows to a Graph

In this example, we'll draw a graph and connect every two points on the graph with an arrow.

First draw a simple graph:

```
>>> x = arange(10)
>>> y = x**2
>>> plot(x, y)
```

Now create a list of all the arrows:

```
>>> arrows = [(x0, y0, dx, dy) for (x0, y0, dx, dy) in \
... zip(x, y, diff(x), diff(y))]
```

The next part is a bit tricky. First, the function `diff()` creates a difference of every two elements in a vector. For example, `diff([1, 2, 3, 30])` is `[1, 1, 27]`. This is exactly what we need for our `dx` and `dy` values for the `Arrow()` function. Second, we combine `x`, `y`, `dx`, and `dy` using the `zip()` function and return a list of tuples by using a list comprehension. Luckily for us, `zip()` uses the shortest vector, so even though `diff()` vectors are shorter by 1, it's not an issue.

Now all that's left is to iterate through the list comprehension and attach an arrow to the graph:

```
>>> cur_axes = gca()
>>> for x0, y0, dx, dy in arrows:
...     cur_axes.add_patch(Arrow(x0, y0, dx, dy))
>>> title('Arrows!')
>>> show()
```

Figure 6-19 shows the added arrows.

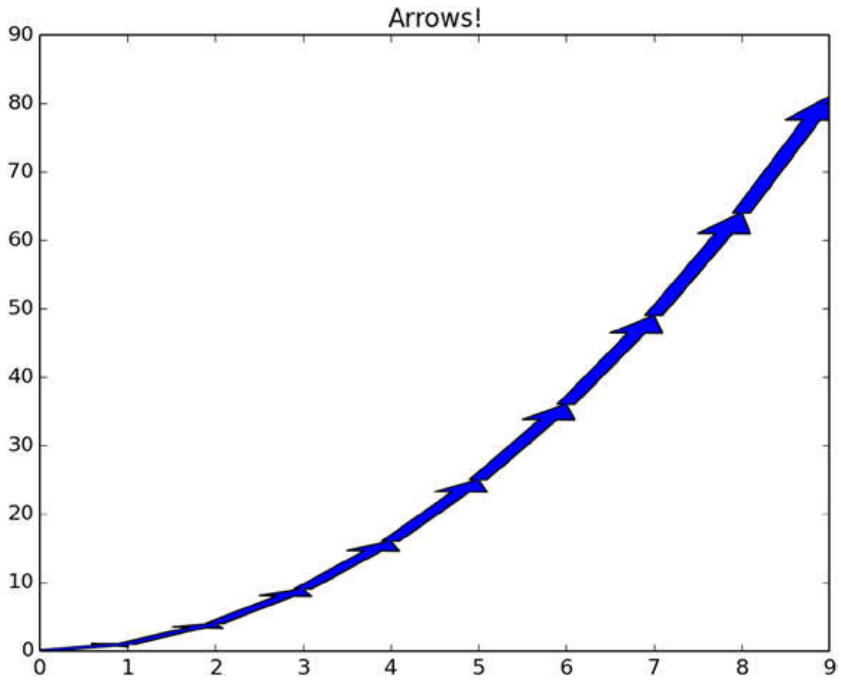

Figure 6-19. *Patching arrows*

Needless to say, `Arrow()` (and other patches) can be customized considerably; you can adjust color, length, width, and more.

Example: Some Other Patches

The code in Listing 6-17 generates a list of patch objects and attaches them to a figure. The figure is originally empty.

Listing 6-17. Some Patches

```
from pylab import *

# Import Ellipse and Wedge to current namespace
from matplotlib.patches import Ellipse, Wedge

# a list of some patches
my_patches = [
    Arrow(0, 4, 0, -4, facecolor='g'),
    Circle([-2, 2], 1.5, linewidth=4, fc='orange'),
    Ellipse([2, 3], 4, 1, 45.0, edgecolor='r'),
    Polygon([[4, 2], [3, 3], [1, -1], [3, -1]], ls='dashed', fill=False),
    Wedge([-1, 0], 3, 200, 300, fc='m', ec='m'),
    Rectangle([1, -2], 3, -2, fill=False, lw=5, ec='r')
    ]

# draw a figure
figure()
axis([-5, 5, -5, 5])

# add the patches
cur_ax = gca()
for p in my_patches:
    cur_ax.add_patch(p)

title('Patches')
show()
```

Figure 6-20 shows the results of the code in Listing 6-17.

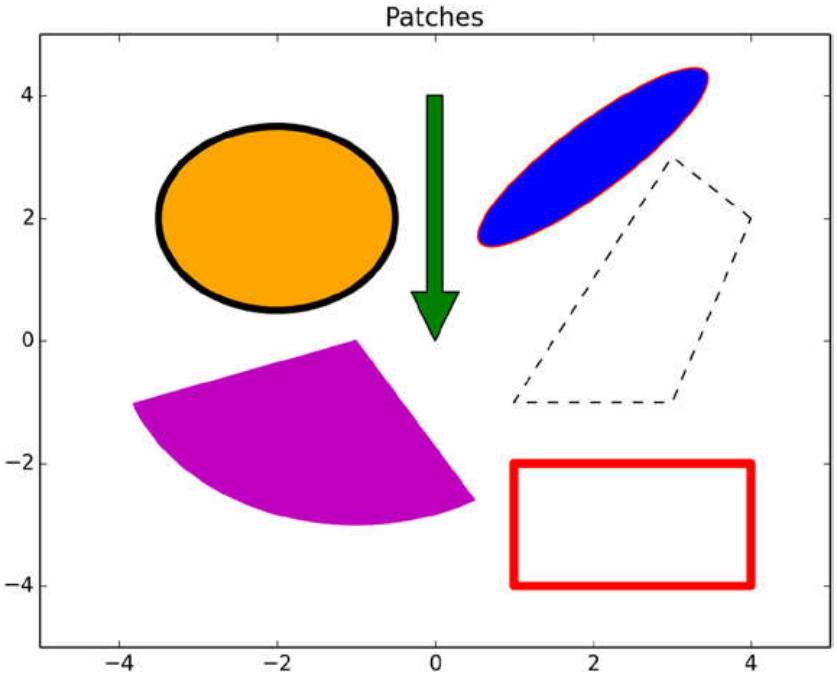

Figure 6-20. *Some patches*

The patch objects `Ellipse` and `Wedge` are not automatically imported to the current namespace when you issue `from pylab import *` (unlike `Arrow`, `Circle`, `Polygon`, and `Rectangle`). Therefore, I've manually imported them to the namespace with the statement `from matplotlib.patches import Ellipse, Wedge`.

I've also passed arguments to the patches to show how to use them: `facecolor` (or `fc`), `edgecolor` (or `ec`), `linestyle` (or `ls`), `linewidth` (or `lw`), and `fill`.

3D Plots

In the first edition of this book, *matplotlib* did not support 3D plots. With newer versions of *matplotlib* also came 3D plots via the *mplot3d* toolkit (see http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html), which is part of *matplotlib*.

Continuing with our sphere from the "Example: A Contour Plot" section, we now turn to drawing an actual 3D plot of the sphere. The function `plot_surface()` plots a surface in 3D (see Listing 6-18).

Listing 6-18. A 3D Surface Plot of a Sphere

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

x = linspace(-1, 1, 100)
y = linspace(-1, 1, 100)
u, v = meshgrid(x, y)

fig = figure()
z = 1-u**2-v**2
z[nonzero(z<0)] = 0
```

```
ax = fig.gca(projection='3d')
ax.plot_surface(u, v, z)
title('A sphere on a plane')
show()
```

After the *PyLab* import, I import the *mplot3d* toolkit.

To generate a 3D plot, the first thing you have to do is create an `Axed3D` object with the parameter, `projection='3d'`. You do this in the line, `ax = fig.gca(projection='3d')`. The variable `ax` is an `Axes3D` object; to access the 3D functions, you have to use `Axes3D`'s member functions (e.g., `ax.plot_surface()` in Listing 6-18).

The resulting 3D plot is shown in Figure 6-21.

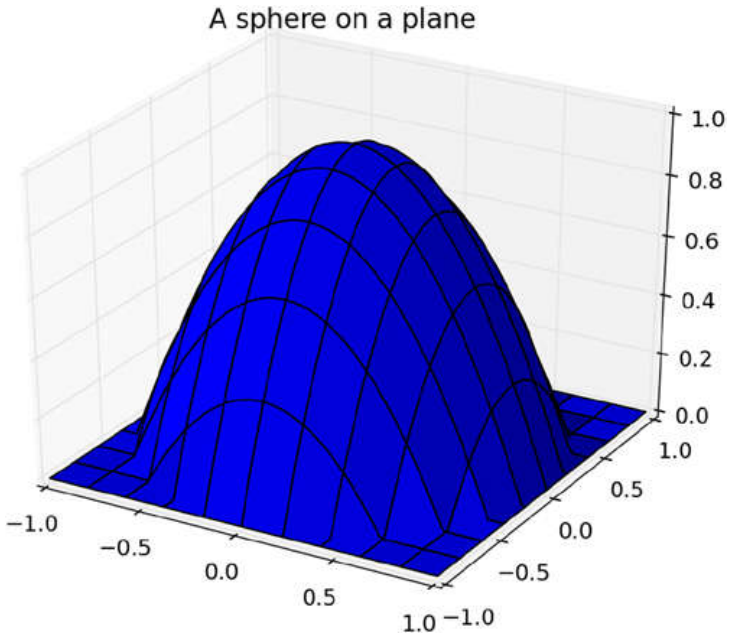

Figure 6-21. *A 3D plot of a sphere on a plane*

Additional 3D plots include `contour()` and `contourf()` (simply replace `ax.plot_surface()` with `ax.contour()` or `ax.contourf()` in Listing 6-18). Figure 6-22 shows the 3D contour plots.

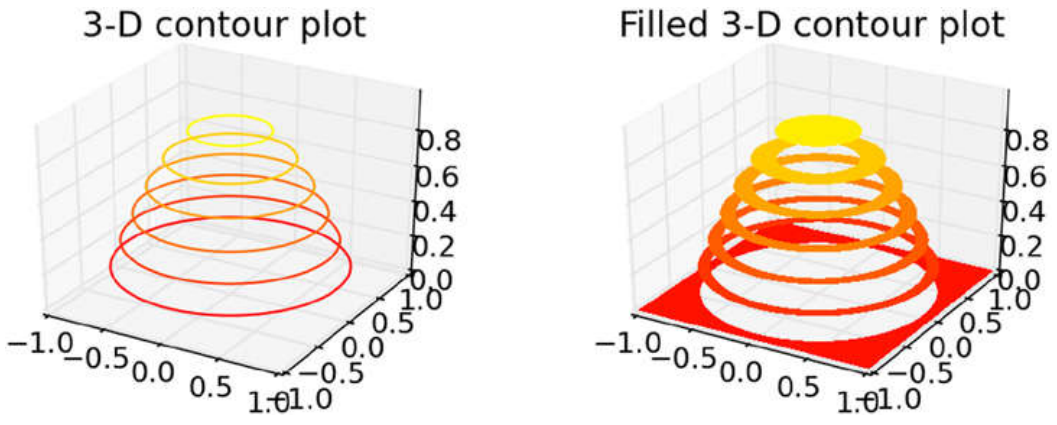

Figure 6-22. *3D contour plots*

The *mplot3d* toolkit also supports plotting 3D line plots (function `plot()`) and scatter plots (function `scatter()`), as shown in Listing 6-19.

Listing 6-19. 3D Line Plots

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

N = 50
theta = linspace(0, 2*pi, N)
x = cos(theta)
y = sin(theta)
z = linspace(0, 1, N)

fig = figure()

# first 3D subplot
ax1 = fig.add_subplot(2, 2, 1, projection='3d')
ax1.plot(x, y, z)
title('3D line plot')

# second 3D subplot
ax2 = fig.add_subplot(2, 2, 2, projection='3d')
x = cos(theta) + randn(N)/10
y = sin(theta) + randn(N)/10
z = linspace(0, 1, N)
ax2.scatter(x, y, z)
title('3D scatter plot')
show()
```

In this example I've also introduced 3D subplots. 3D subplots are added with a call to `fig.add_subplot(..., **projection='3d'**)`. Notice that to create a 3D subplot, you have to pass the parameter `projection='3d'` when you create the subplot. The results of Listing 6-19 are shown in Figure 6-23.

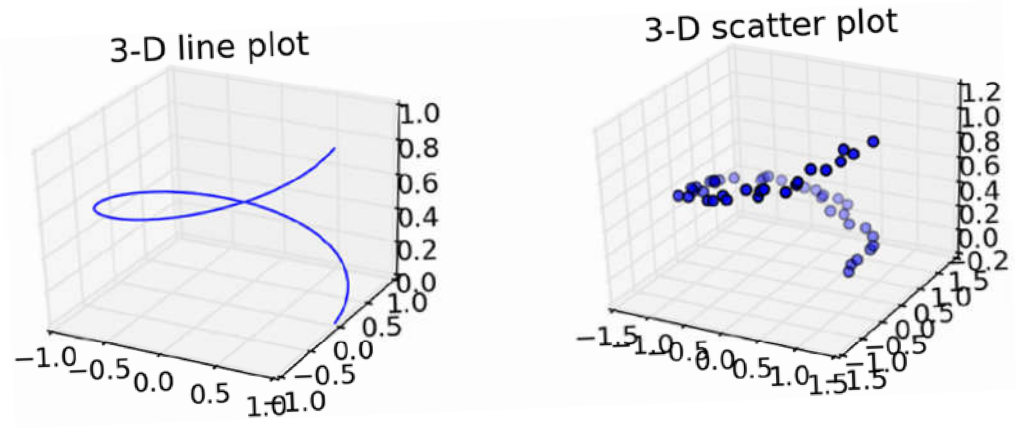

Figure 6-23. *3D line plots*

The Basemap Toolkit

The *basemap* toolkit (see http://matplotlib.org/basemap/) is based on *matplotlib* and enables plotting on 2D data maps in Python. The following section, "Example: French Airports," uses many of *basemap*'s features and should provide enough information to get you up and running with the package. For a full account, refer to *basemap*'s documentation.

Example: French Airports

In this example, I plot the locations of French airports on a map of France. To do this, I use information contained in METAR weather reports. METAR weather reports are used by pilots while preparing for flight. They contain information such as visibility and wind speed. Each station has a designated four-character code, where the two first characters designate the country and the remaining two characters are associated with the airfield. The METAR station codes for France start with LF. For example, LFPB is the station code of the Paris, Le Bourget airport. A full list of these stations is available at the National Center for Atmospheric Research (a direct link: http://weather.rap.ucar.edu/surface/stations.txt). Begin by downloading this file and saving it in the directory, `../data/stations.txt`. The code presented in Listing 6-20 assumes the file is available at that directory location. You'll also need to download and install the *basemap* toolkit (http://matplotlib.org/basemap/users/download.html). Refer to Chapter 2 if you require a manual install.

Listing 6-20. French Airports

```
from pylab import *
from mpl_toolkits.basemap import Basemap

# read METAR airport codes, retrieve lat/lon values
airports = {}
metar_codes = open('../data/stations.txt').readlines()
for row in metar_codes:
    code = row[20:24] # METAR code
    if(code.startswith('LF')):
        lat = float(row[39:41])+float(row[42:44])/60
        lon = float(row[47:50])+float(row[51:53])/60
        if(row[53]=='W'): lon = -lon
        if(row[44]=='S'): lat = -lat
        name = row[:20].split('/')[0].strip().title()
        airports[code] = [lon, lat, name]

# draw the map
m = Basemap(llcrnrlon=-5.0, llcrnrlat=41.0, urcrnrlon=10.0, urcrnrlat=51.5,
    projection='merc', resolution='h', suppress_ticks=False)
m.drawcoastlines()
m.bluemarble(scale=0.5)
m.drawmapboundary(fill_color='lightblue')
m.drawcountries(linewidth=1)
m.drawmapscale(-2.0, 41.5, -2.0, 41.5, 500, units='km')
axis('off')

# plot the airports
for site, [lon, lat, name] in airports.items():
    xpt, ypt = m(lon, lat)
```

```
    text(xpt, ypt, ' '+name, va='center', ha='left', fontsize=9,
        color='white')
    plot(xpt, ypt, 'wo')
title('METAR sites in France')
show()
```

First we import *PyLab* and the *basemap* toolkit. Next, we initialize a dictionary object named `airports`. The dictionary object `airports` uses the METAR code as key and a list with the longitude, latitude, and airfield name as a value. We then proceed to read the entire stations.txt file and store it in `metar_codes` variable as a list of strings; each string corresponds to a line in the file. Next, we retrieve the station METAR codes. We do this by slicing each string in `metar_codes` at location 20-24; the value is stored in variable `code`. Since METAR codes for France start with LF (you can verify this by reading the stations.txt file and scrolling to the FRANCE section), we check whether `code` starts with `'LF'`. If it does, we start processing the airfield data. We again use string slicing and convert the strings to float values to store longitude and latitude information. Notice the division by 60; this is needed to translate degrees to fractions of degrees. Also, in case of the Western hemisphere, longitude values are negative; in the case of the Southern hemisphere, latitude values are negative.

Up to this point, I've used basic Python capabilities, but now I will turn to the *basemap* toolkit. First, I create a *basemap* object named `m`, and I initialize that object to show the area of France. The initialization parameter `llcrnrlon` stands for lower-left-corner-longitude; similarly, the initialization parameter `urcrnrlat` stands for upper-right-corner-latitude. I decide on the `'merc'` (Mercator) projection, and that projection determines how to project a sphere (the earth) on a plain (our plot). The full list of projections is available at http://matplotlib.org/basemap/api/basemap_api.html. Next, I draw the coastlines (`m.coastlines()`), add blue marble coloring (`m.bluemarble()`), draw the country boundaries (`m.drawmapboundary()`), and then set the map scale to kilometers (`m.drawmapscale()`).

The last part plots the airports in our dictionary object, `airports`. We do this by converting longitude and latitude values with the *basemap* object, `m`, as follows: `xpt, ypt = m(lon, lat)`.

The result is shown in Figure 6-24.

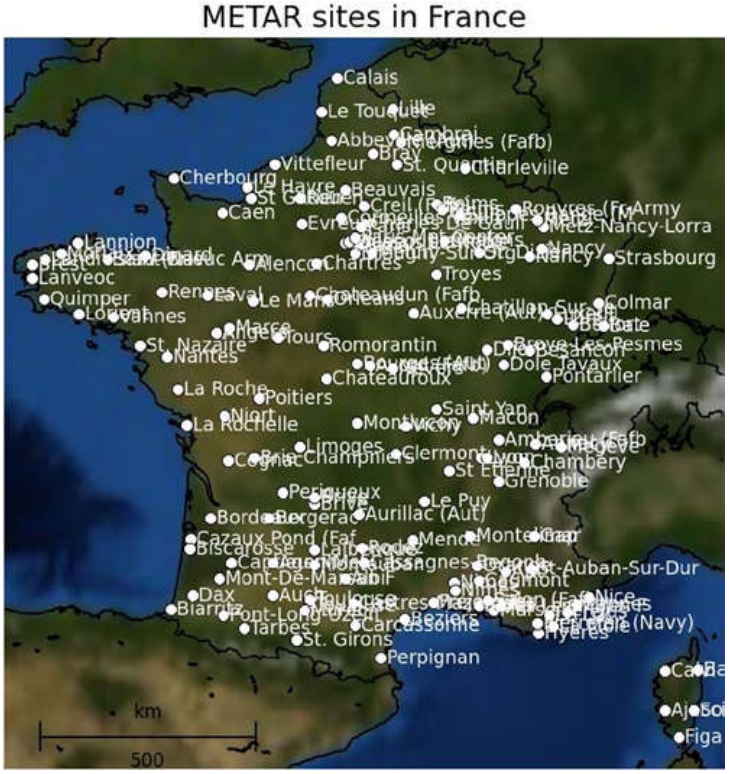

Figure 6-24. *METAR sites in France*

Final Notes and References

We've explored the *matplotlib* package, a rich package that supports plotting in Python. The strong suit of *matplotlib* is that it makes it easy to plot simple and complex graphs with a high number of customization options. If you're not familiar with the package, then try exploring it with IPython's tab completion, complemented by `help()`, trial and error, and the manual. Taken together, these tips should yield excellent results in no time.

The chapter's examples cover a wide range of topics and situations related to graphs and plots. And while your needs may be different, you should now have the tools you need to explore the *matplotlib* package on your own.

The *matplotlib* web site is an excellent source of information, and I encourage you to explore it and learn more about the package:

- The *matplotlib* website, http://matplotlib.org/
- The *Mplot3d* toolkit, http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
- The *basemap* toolkit, http://matplotlib.org/basemap/