CHAPTER 21

Project 2: Painting a Pretty Picture

In this project, you learn how you can create graphics in Python. More specifically, you create a PDF file with graphics helping you visualize data that you read from a text file. While you could get such functionality from a regular spreadsheet, Python gives you much more power, as you'll see when you get to the second implementation and automatically download your data from the Internet.

In the previous chapter, we looked at HTML and XML—and here is another acronym, which I guess you're probably familiar with: PDF, short for Portable Document Format. PDF is a format created by Adobe that can represent any kind of document with graphics and text. The PDF file is not really editable (as, say, a Microsoft Word file would be), but there is reader software freely available for most platforms, and the PDF file should look the same no matter which reader you use or which platform you are on (as opposed to HTML, with which the correct fonts may not be available, you would normally have to ship pictures as separate files, and so on). If you don't already have a PDF reader, Adobe's own Acrobat Reader is freely available from the Adobe web site (`http://adobe.com/products/acrobat/readstep.html`).

What's the Problem?

Python is excellent for analyzing data. With its file-handling and string-processing facilities, it's probably easier to create some form of report from a data file than to create something similar in your average spreadsheet, especially if what you want to do requires some complicated programming logic.

You have seen (in Chapter 3) how you can use string formatting to get pretty output—for example, if you want to print numbers in columns. However, sometimes plain text just isn't enough. (As they say, a picture is worth a thousand words.) In this project, you learn the basics of the ReportLab package, which enables you to create graphics and documents in the PDF format (and a few other formats) almost as easily as you created plain text earlier.

As you play with the concepts in this project, I encourage you to find some application that is interesting to you. I have chosen to use data about sunspots (from the Space Weather Prediction Center, a part of the US National Oceanic and Atmospheric Administration) and to create a line diagram from this data.

The program should be able to do the following:

- Download a data file from the Internet.
- Parse the data file and extract the interesting parts.
- Create PDF graphics based on the data.

As in the previous project, these goals might not be fully met by the first prototype.

Useful Tools

The crucial tool in this project is the graphics-generating package. Quite a few such packages are available. If you visit the Vaults of Parnassus site (`http://www.vex.net/parnassus`), you will find a separate category for graphics. I have chosen ReportLab because it is easy to use and has extensive functionality for both graphics and document generation in PDF. If you want to go beyond the basics, you might also want to consider the PyX graphics package (`http://pyx.sf.net`), which is really powerful and has support for TeX-based typography.

To get the ReportLab package, go to the official web site at `http://www.reportlab.org`. There you will find the software, documentation, and samples. The software should be available at `http://www.reportlab.org/downloads.html`. Simply download the ReportLab toolkit, uncompress the archive (`ReportLab_x.zip`, where `x` is a version number), and put the `reportlab` directory inside the uncompressed directory in your Python path.

When you have done this, you should be able to import the `reportlab` module, as follows:

```
>>> import reportlab
>>>
```

■ **Note** Although I show you how some ReportLab features work in this project, much more functionality is available. To learn more, I suggest you obtain the user guide and the (separate) graphics guide, made available on the ReportLab web site (on the documentation page). They are quite readable and are much more comprehensive than this one chapter could possibly be.

Preparations

Before you start programming, you need some data with which to test your program. I have chosen (quite arbitrarily) to use data about sunspots, available from the web site of the Space Weather Prediction Center (`http://www.swpc.noaa.gov`). You can find the data I use in my examples at `http://www.swpc.noaa.gov/ftpdir/weekly/Predict.txt`.

This data file is updated weekly and contains information about sunspots and radio flux. (Don't ask me what that means.) Once you have this file, you're ready to start playing with the problem.

Here is a part of the file to give you an idea of how the data looks:

```
#         Predicted Sunspot Number And Radio Flux Values
#                     With Expected Ranges
#
#         -----Sunspot Number------  ----10.7 cm Radio Flux----
# YR MO   PREDICTED    HIGH    LOW   PREDICTED    HIGH    LOW
#-------------------------------------------------------------
2007 12         4.8     5.0     4.7       67.6    70.4    64.7
2008 01         4.3     4.4     4.2       66.7    69.5    63.8
2008 02         4.0     4.1     3.9       66.1    68.9    63.2
2008 03         4.2     4.3     4.0       65.7    68.6    62.8
2008 04         4.6     4.8     4.4       65.7    68.6    62.7
2008 05         5.2     5.6     4.9       65.6    68.7    62.5
2008 06         5.8     6.3     5.2       65.2    68.5    62.0
2008 07         6.3     7.1     5.5       64.9    68.4    61.4
2008 08         7.4     8.6     6.3       65.1    68.9    61.2
2008 09         8.6    10.2     7.0       65.4    69.6    61.2
```

First Implementation

In this first implementation, let's just put the data into our source code, as a list of tuples. That way, it's easily accessible. Here is an example of how you can do it:

```
data = [
#    Year  Month  Predicted  High   Low
(2007, 12,      4.8,       5.0,   4.7),
(2008,  1,      4.3,       4.4,   4.2),
    # Add more data here
    ]
```

With that out of the way, let's see how you can turn the data into graphics.

Drawing with ReportLab

ReportLab consists of many parts and enables you to create output in several ways. The most basic module for generating PDFs is `pdfgen`. It contains a `Canvas` class with several low-level methods for drawing. To draw a line on a `Canvas` called `c`, you call the `c.line` method, for example.

You'll use the more high-level graphics framework (in the package `reportlab.graphics` and its submodules), which will enable you to create various shape objects and to add them to a `Drawing` object that you can later output to a file in PDF format.

Listing 21-1 shows a sample program that draws the string "Hello, world!" in the middle of a 100 × 100-point PDF figure. (You can see the result in Figure 21-1.) The basic structure is as follows: you create a drawing of a given size, you create graphical elements (in this case, a

String object) with certain properties, and then you add the elements to the drawing. Finally, the drawing is rendered into PDF format and saved to a file.

Figure 21-1. *A simple ReportLab figure*

Listing 21-1. *A Simple ReportLab Program (hello_report.py)*

```
from reportlab.graphics.shapes import Drawing, String
from reportlab.graphics import renderPDF

d = Drawing(100, 100)
s = String(50, 50, 'Hello, world!', textAnchor='middle')

d.add(s)

renderPDF.drawToFile(d, 'hello.pdf', 'A simple PDF file')
```

The call to `renderPDF.drawToFile` saves your PDF file to a file called `hello.pdf` in the current directory.

The main arguments to the `String` constructor are its x and y coordinates and its text. In addition, you can supply various attributes (such as font size, color, and so on). In this case, I've supplied a `textAnchor`, which is the part of the string that should be placed at the point given by the coordinates.

■ **Note** When you run this program, you may get two warnings: one saying that the Python Imaging Library is not available, and the other that `zlib` is not available. (If you have installed either of these, that warning will, of course, not appear.) You won't need either of these libraries for the code in this project, so you can simply ignore the warnings. And if you don't get the warning, that's not a problem, of course.

Constructing Some PolyLines

To create a line diagram (a graph) of the sunspot data, you need to create some lines. In fact, you need to create several lines that are linked. ReportLab has a special class for this: `PolyLine`.

A `PolyLine` is created with a list of coordinates as its first argument. This list is of the form `[(x0, y0), (x1, y1), ...]`, with each pair of x and y coordinates making one point on the `PolyLine`. See Figure 21-2 for a simple `PolyLine`.

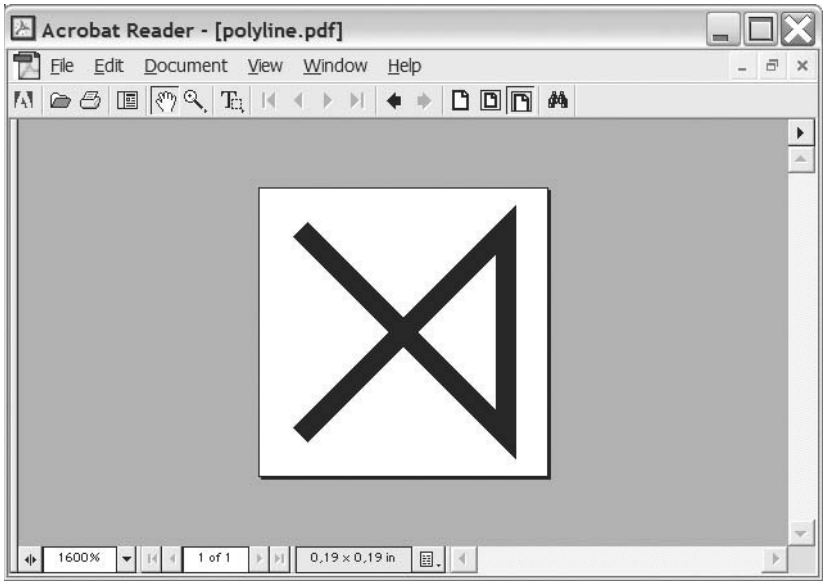

Figure 21-2. *PolyLine([(0, 0), (10, 0), (10, 10), (0, 10)])*

To make a line diagram, one polyline must be created for each column in the data set. Each point in these polylines will consist of a time (constructed from the year and month) and a value (which is the number of sunspots, taken from the relevant column). To get one of the columns (the values), list comprehensions can be useful:

```
pred = [row[2] for row in data]
```

Here, `pred` (for "predicted") will be a list of all the values in the third column of the data. You can use a similar strategy for the other columns. (The time for each row would need to be calculated from both the year and month; for example, *year* + *month*/12.)

Once you have the values and the timestamps, you can add your polylines to the drawing like this:

```
drawing.add(PolyLine(zip(times, pred), strokeColor=colors.blue))
```

It isn't necessary to set the stroke color, of course, but it makes it easier to tell the lines apart. (Note how `zip` is used to combine the times and values into a list of tuples.)

Writing the Prototype

You now have what you need to write your first version of the program. The source code is shown in Listing 21-2.

Listing 21-2. *The First Prototype for the Sunspot Graph Program (sunspots_proto.py)*

```
from reportlab.lib import colors
from reportlab.graphics.shapes import *
from reportlab.graphics import renderPDF

data = [
#    Year  Month  Predicted  High  Low
    (2007,  8,    113.2,     114.2, 112.2),
    (2007,  9,    112.8,     115.8, 109.8),
    (2007, 10,    111.0,     116.0, 106.0),
    (2007, 11,    109.8,     116.8, 102.8),
    (2007, 12,    107.3,     115.3,  99.3),
    (2008,  1,    105.2,     114.2,  96.2),
    (2008,  2,    104.1,     114.1,  94.1),
    (2008,  3,     99.9,     110.9,  88.9),
    (2008,  4,     94.8,     106.8,  82.8),
    (2008,  5,     91.2,     104.2,  78.2),
    ]

drawing = Drawing(200, 150)

pred = [row[2]-40 for row in data]
high = [row[3]-40 for row in data]
low = [row[4]-40 for row in data]
times = [200*((row[0] + row[1]/12.0) - 2007)-110 for row in data]

drawing.add(PolyLine(zip(times, pred), strokeColor=colors.blue))
drawing.add(PolyLine(zip(times, high), strokeColor=colors.red))
drawing.add(PolyLine(zip(times, low),  strokeColor=colors.green))
```

```
drawing.add(String(65, 115, 'Sunspots', fontSize=18, fillColor=colors.red))

renderPDF.drawToFile(drawing, 'report1.pdf', 'Sunspots')
```

As you can see, I have adjusted the values and timestamps to get the positioning right. The resulting drawing is shown in Figure 21-3.

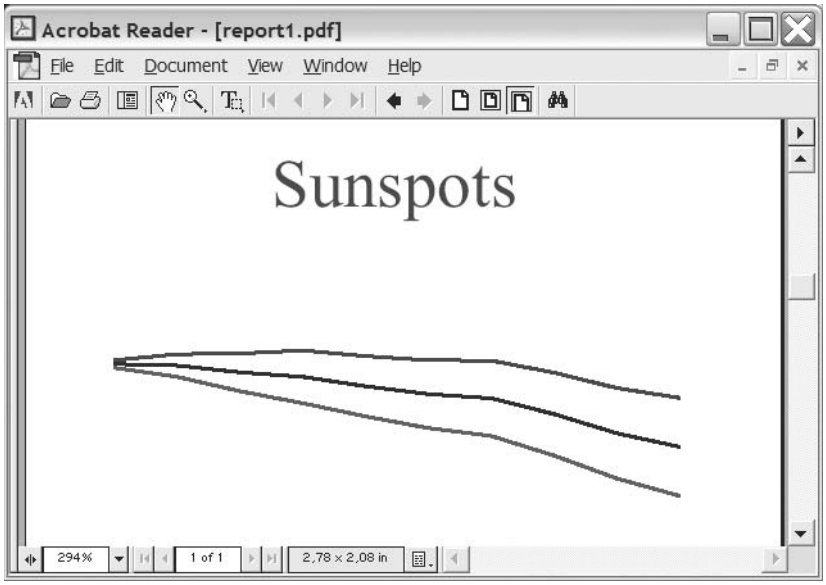

Figure 21-3. *A simple sunspot graph*

Although it is pleasing to have made a program that works, there is clearly still room for improvement.

Second Implementation

So, what did you learn from your prototype? You have figured out the basics of how to draw stuff with ReportLab. You have also seen how you can extract the data in a way that works well for drawing your graph. However, there are some weaknesses in the program. To position things properly, I had to add some ad hoc modifications to the values and timestamps. And the program doesn't actually get the data from anywhere (or, rather, it "gets" the data from a list inside the program itself, rather than reading it from an outside source).

Unlike Project 1 (in Chapter 20), the second implementation won't be much larger or more complicated than the first. It will be an incremental improvement that uses some more appropriate features from ReportLab and actually fetches its data from the Internet.

Getting the Data

As you saw in Chapter 14, you can fetch files across the Internet with the standard module `urllib`. Its function `urlopen` works in a manner quite similar to `open`, but takes a URL instead of a file name as its argument. When you have opened the file and read its contents, you need to filter out what you don't need. The file contains empty lines (consisting of only whitespace) and lines beginning with some special characters (`#` and `:`). The program should ignore these. (See the sample file fragment in the section "Preparations" earlier in this chapter.)

Assuming that the URL is kept in a variable called `URL`, and that the variable `COMMENT_CHARS` has been set to the string `'#:'`, you can get a list of rows (as in our original program) like this:

```
data = []
for line in urlopen(URL).readlines():
    if not line.isspace() and not line[0] in COMMENT_CHARS:
        data.append([float(n) for n in line.split()])
```

The preceding code will include all the columns in the data list, although you aren't particularly interested in the ones pertaining to radio flux. However, those columns will be filtered out when you extract the columns you really need (as you did in the original program).

■ **Note** If you are using a data source of your own (or if, by the time you read this, the data format of the sunspot file has changed), you will, of course, need to modify this code accordingly.

Using the LinePlot Class

If you thought getting the data was surprisingly simple, drawing a prettier line plot isn't much of a challenge either. In a situation like this, it's best to thumb through the documentation (in this case, the ReportLab docs) to see if a feature that can do what you need already exists, so you don't need to implement it all yourself. Luckily, there is just such a thing: the `LinePlot` class from the module `reportlab.graphics.charts.lineplots`. Of course, you could have looked for this to begin with, but in the spirit of rapid prototyping, you just used what was at hand to see what you could do. Now it's time to go one step further.

The `LinePlot` is instantiated without any arguments, and then you set its attributes before adding it to the `Drawing`. The main attributes you need to set are `x`, `y`, `height`, `width`, and `data`. The first four should be self-explanatory; the latter is simply a list of point-lists, where a point-list is a list of tuples, like the one you used in your `PolyLines`.

To top it off, let's set the stroke color of each line. The final code is shown in Listing 21-3. The resulting figure (which will, of course, look quite a bit different with different input data) is shown in Figure 21-4.

Listing 21-3. *The Final Sunspot Program (sunspots.py)*

```
from urllib import urlopen
from reportlab.graphics.shapes import *
from reportlab.graphics.charts.lineplots import LinePlot
from reportlab.graphics.charts.textlabels import Label
from reportlab.graphics import renderPDF

URL = 'http://www.swpc.noaa.gov/ftpdir/weekly/Predict.txt'
COMMENT_CHARS = '#:'


drawing = Drawing(400, 200)
data = []
for line in urlopen(URL).readlines():
    if not line.isspace() and not line[0] in COMMENT_CHARS:
        data.append([float(n) for n in line.split()])

pred = [row[2] for row in data]
high = [row[3] for row in data]
low = [row[4] for row in data]
times = [row[0] + row[1]/12.0 for row in data]

lp = LinePlot()
lp.x = 50
lp.y = 50
lp.height = 125
lp.width = 300
lp.data = [zip(times, pred), zip(times, high), zip(times, low)]
lp.lines[0].strokeColor = colors.blue
lp.lines[1].strokeColor = colors.red
lp.lines[2].strokeColor = colors.green

drawing.add(lp)

drawing.add(String(250, 150, 'Sunspots',
            fontSize=14, fillColor=colors.red))


renderPDF.drawToFile(drawing, 'report2.pdf', 'Sunspots')
```

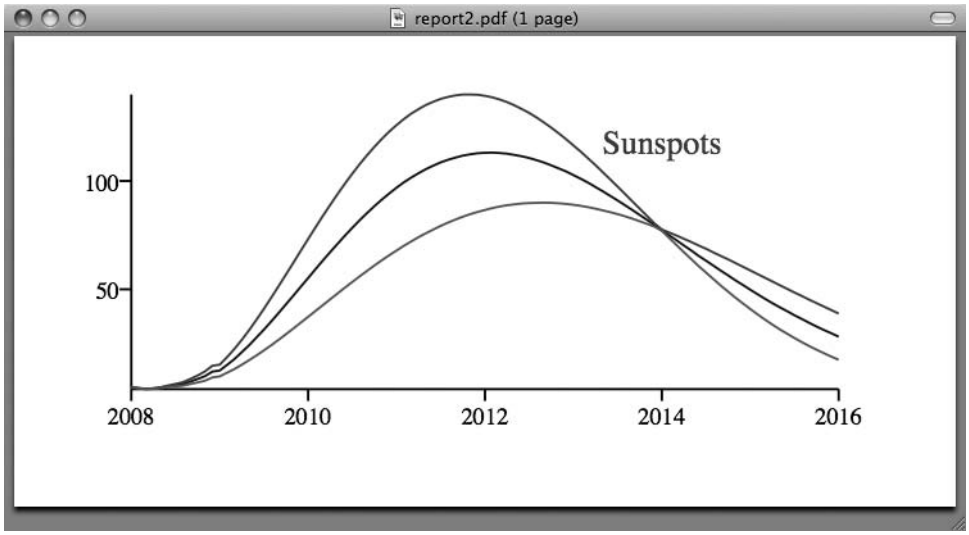

Figure 21-4. *The final sunspot graph*

Further Exploration

Many graphics and plotting packages are available for Python. One good alternative to ReportLab is PyX, which I mentioned earlier in this chapter. It is also possible to use wxPython (discussed in Chapter 12) to create vector graphics files of different kinds.

Using either ReportLab or PyX (or some other package), you could try to incorporate automatically generated graphics into a document (perhaps generating parts of that as well). You could use some of the techniques from Chapter 20 to add markup to the text. If you want to create a PDF document, Platypus, a part of ReportLab, is useful for that. (You could also integrate the PDF graphics with some typesetting system such as LaTeX.) If you want to create web pages, there are ways of creating pixmap graphics (such as GIF or PNG) using Python as well—just do a web search on the topic.

If your primary goal is to plot data (which is what we did in this project), you have many alternatives to ReportLab and PyX. One good option is Matplotlib/pylab (`http://matplotlib.sf.net`), but a lot of other (similar) packages are available.

What Now?

In the first project, you learned how to add markup to a plain-text file by creating an extensible parser. In the next project, you learn about analyzing marked-up text (in XML) by using parser mechanisms that already exist in the Python standard library. The goal of the project is to use a single XML file to specify an entire web site, which will then be generated automatically (with files, directories, added headers, and footers) by your program. The techniques you learn in the next project will be applicable to XML parsing in general, and with XML being used in an increasing number of different settings, that can't hurt.