6

Files and the Internet

Python makes it easy for your programs to use files and connect to the Internet. You can read data from files, write data to files, and fetch content from the Internet. You can even check for new mail and tweet—all from your program.

Files

When you run a Python program, any values you have in variables will be lost. Files provide a means of making data more permanent.

Reading Files

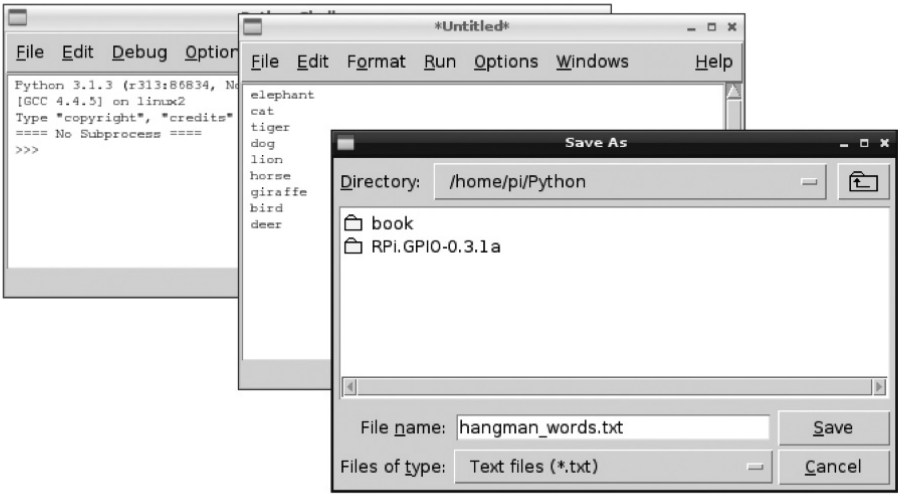
Python makes reading the contents of a file extremely easy. As an example, we can convert the Hangman program from Chapter 4 to read the list of words from a file rather than have them fixed in the program.

First of all, start a new file in IDLE and put some words in it, one per line. Then save the file with the name hangman_words.txt in the same directory as the Hangman program from Chapter 4 (04_08_hangman_full.py). Note that in the Save dialog you will have to change the file type to .txt (see Figure 6-1).

Before we modify the Hangman program itself, we can just experiment with reading the file in the Python console. Enter the following into the console:

```
>>> f = open('Python/hangman_words.txt')
```

Note that the Python console has a current directory of /home/pi, so the directory Python (or wherever you saved the file) must be included.



```
Python 3.1.3 (r313:86834, No
[GCC 4.4.5] on linux2
Type "copyright", "credits"
==== No Subprocess ====
>>>
```

```
elephant
cat
tiger
dog
lion
horse
giraffe
bird
deer
```

Directory: /home/pi/Python

book
RPi.GPIO-0.3.1a

File name: hangman_words.txt

Files of type: Text files (*.txt)

Figure 6-1 *Creating a text file in IDLE*

Next enter the following into the Python console:

```
>>> words = f.read()
>>> words
'elephant\ncat\ntiger\ndog\nlion\nhorse\ngiraffe\nbird\ndeer\n'
>>> words.splitlines()
['elephant', 'cat', 'tiger', 'dog', 'lion', 'horse', 'giraffe'

, 'bird', 'deer']
>>>
```

I told you it was easy! All we need to do to add this file to the Hangman program is replace the line

```
words = ['chicken', 'dog', 'cat', 'mouse', 'frog']
```

with the following lines:

```
f = open('hangman_words.txt')
words = f.read().splitlines()
f.close()
```

The line `f.close()` has been added. You should always call the `close` command when you are done with a file to free up operating system resources. Leaving a file open can lead to problems.

The full program is contained in the file 06_01_hangman_file.py, and a suitable list of animal names can be found in the file hangman_words.txt.

This program does nothing to check that the file exists before trying to read it. So, if there file isn't there, we get an error that looks something like this:

```
Traceback (most recent call last):
  File "06_01_hangman_file.py", line 4, in <module>
    f = open('hangman_words.txt')
IOError: [Errno 2] No such file or directory: 'hangman_words.txt'
```

To make this a bit more user friendly, the file-reading code needs to be inside a `try` command, like this:

```
try:
       f = open('hangman_words.txt')
       words = f.read().splitlines()
       f.close()
except IOError:
       print("Cannot find file 'hangman_words.txt'")
       exit()
```

Python will try to open the file, but because the file is missing it will not be able to. Therefore, the `except` part of the program will apply, and the more friendly message will be displayed. Because we cannot do anything without a list of words to guess, there is no point in continuing, so the `exit` command is used to quit.

In writing the error message, we have repeated the name of the file. Sticking strictly to the Don't Repeat Yourself (DRY) principle, the filename should be put in a variable, as shown next. That way, if we decide to use a different file, we only have to change the code in one place.

```
words_file = 'hangman_words.txt'
try:
       f = open(words_file)
       words = f.read().splitlines()
       f.close()
except IOError:
       print("Cannot find file: " + words_file)
       exit()
```

A modified version of Hangman with this code in it can be found in the file 06_02_hangman_file_try.py.

Reading Big Files

The way we did things in the previous section is fine for a small file containing some words. However, if we were reading a really huge file (say, several megabytes), then two things would happen. First, it would take a significant amount of time for Python to read all the data. Second, because all the data is read at once, at least as much memory as the file size would be used, and for truly enormous files, that might result in Python running out of memory.

If you find yourself in the situation where you are reading a big file, you need to think about how you are going to handle it. For example, if you were searching a file for a particular string, you could just read one line of the file at a time, like this:

```
#06_03_file_readline
words_file = 'hangman_words.txt'
try:
       f = open(words_file)
       line = f.readline()
       while line != '':
              if line == 'elephant\n':
                     print('There is an elephant in the file')
                     break
              line = f.readline()
       f.close()
except IOError:
       print("Cannot find file: " + words_file)
```

When the function `readline` gets to the last line of the file, it returns an empty string (`''`). Otherwise, it returns the contents of the line, including the end-of-line character (\n). If it reads a blank line that is actually just a gap between lines and not the end of the file, it will return just the end-of-line character (\n). By the program only reading one line at a time, the memory being used is only ever equivalent to one full line.

If the file is not broken into convenient lines, you can specify an argument in `read` that limits the number of characters read. For example, the following will just read the first 20 characters of a file:

```
>>> f = open('hangman_words.txt')
>>> f.read(20)
'elephant\ncat\ntiger\nd'
>>> f.close()
```

Writing Files

Writing files is almost as simple. When a file is opened, as well as specifying the name of the file to open, you can also specify the mode in which to open the file. The mode is represented by a character, and if no mode is specified it is assumed to be `r` for `read`. The modes are as follows:

- **`r` (`read`)**.
- **`w` (`write`)** Replaces the contents of any existing file with that name.
- **`a` (`append`)** Appends anything to be written onto the end of an existing file.
- **`r+`** Opens the file for both reading and writing (not often used).

To write a file, you open it with a second parameter of `'w'`, `'a'`, or `'r+'`. Here's an example:

```
>>> f = open('test.txt', 'w')
>>> f.write('This file is not empty')
>>> f.close()
```

The File System

Occasionally, you will need to do some file-system-type operations on files (moving them, copying them, and so on). Python uses Linux to perform these actions, but provides a nice Python-style way of doing them. Many of these functions are in the `shutil` (shell utility) package. There's a number of subtle variations on the basic copy and move features that deal with file permissions and metadata. In this section, we just deal with the basic operations. You can refer to the official Python documentation for any other functions (http://docs.python.org/release/3.1.5/library).

Here's how to copy a file:

```
>>> import shutil
>>> shutil.copy('test.txt', 'test_copy.txt')
```

To move a file, either to change its name or move it to a different directory:

```
shutil.move('test_copy.txt', 'test_dup.txt')
```

This works on directories as well as files. If you want to copy an entire folder—including all its contents and its content's contents—you can use the function `copytree`. The rather dangerous function `rmtree`, on the other hand, will recursively remove a directory and all its contents—exercise extreme caution with this one!

The nicest way of finding out what is in a directory is via *globbing*. The package glob allows you to create a list of files in a directory by specifying a wildcard (*). Here's an example:

```
>>> import glob
glob.glob('*.txt')
['hangman_words.txt', 'test.txt', 'test_dup.txt']
```

If you just want all the files in the folder, you could use this:

```
glob.glob('*')
```

Pickling

Pickling involves saving the contents of a variable to a file in such a way that the file can be later loaded to get the original value back. The most common reason for wanting to do this is to save data between runs of a program. As an example, we can create a complex list containing another list and various other data objects and then pickle it into a file called mylist.pickle, like so:

```
>>> mylist = ['a', 123, [4, 5, True]]
>>> mylist
['a', 123, [4, 5, True]]
>>> import pickle
>>> f = open('mylist.pickle', 'w')
>>> pickle.dump(mylist, f)
>>> f.close()
```

If you find the file and open it in an editor to have a look, you will see something cryptic that looks like this:

```
(lp0
S'a'
p1
aI123
a(lp2
I4
aI5
aI01
aa.
```

That is to be expected; it is text, but it is not meant to be in human-readable form. To reconstruct a pickle file into an object, here is what you do:

```
>>> f = open('mylist.pickle')
>>> other_array = pickle.load(f)
>>> f.close()
>>> other_array
['a', 123, [4, 5, True]]
```

Internet

Most applications use the Internet in one way or another, even if it is just to check whether a new version of the application is available to remind the user about. You interact with a web server by sending HTTP (Hypertext Transfer Protocol) requests to it. The web server then sends a stream of text back as a response. This text will be HTML (Hypertext Markup Language), the language used to create web pages.

Try entering the following code into the Python console.

```
>>> import urllib.request
>>> u = 'http://www.amazon.com/s/ref=nb_sb_noss?field-keywords=raspberry+pi'
>>> f = urllib.request.urlopen(u)
>>> contents = f.read()
… lots of HTML
>>> f.close()
```

Note that you will need to execute the `read` line as soon as possible after opening the URL. What you have done here is to send a web request to www.amazon.com, asking it to search on "raspberry pi." This has sent back the HTML for Amazon's web page that would display (if you were using a browser) the list of search results.

If you look carefully at the structure of this web page, you can see that you can use it to provide a list of Raspberry Pi–related items found by Amazon. If you scroll around the text, you will find some lines like these:

```
<div class="productTitle"><a href="http://www.amazon
.com/Raspberry-User-Guide

-Gareth-Halfacree/dp/111846446X"> Raspberry Pi User Guide</a> <span

class="ptBrand">by <a href="/Gareth-Halfacree/e
/B0088CA5ZM">Gareth
```

```
Halfacree</a> and Eben Upton</span><span
class="binding"> (<span class

="format">Paperback</span> - Nov. 13, 2012)</span></div>
```

They key thing here is `<div class="productTitle">`. There is one instance of this before each of the search results. (It helps to have the same web page open in a browser for comparison.) What you want to do is copy out the actual title text. You could do this by finding the position of the text `productTitle`, counting two > characters, and then taking the text from that position until the next < character, like so:

```
#06_04_amazon_scraping
import urllib.request

u = 'http://www.amazon.com/s/ref=nb_sb_noss?field-
keywords=raspberry+pi'
f = urllib.request.urlopen(u)
contents = str(f.read())
f.close()
i = 0
while True:
        i = contents.find('productTitle', i)
        if i == -1:
                break
        # Find the next two '>' after 'productTitle'
        i = contents.find('>', i+1)
        i = contents.find('>', i+1)
        # Find the first '<' after the two '>'
        j = contents.find('<', i+1)
        title = contents[i+2:j]
        print(title)
```

When you run this, you will mostly get a list of products. If you really get into this kind of thing, then search for "Regular Expressions in Python" on the Internet. Regular expressions are almost a language in their own right; they are used for doing complex searches and validations of text. They are not easy to learn or use, but they can simplify tasks like this one.

What we have done here is called *web scraping*, and it is not ideal for a number of reasons. First of all, organizations often do not like people "scraping" their web pages with automated programs. Therefore, you may get a warning or even banned from some sites.

Second, this action is very dependent on the structure of the web page. One tiny change on the website and everything could stop working. A much better approach is to look for an official web service interface to the site. Rather than returning the data as HTML, these services return much more easily processed data, often in XML or JSON format.

If you want to learn more about how to do this kind of thing, search the Internet for "web services in Python."

Summary

This chapter has given you the basics of how to use files and access web pages from Python. There is actually a lot more to Python and the Internet, including accessing e-mail and other Internet protocols. For more information on this, have a look at the Python documentation at http://docs.python.org/release/3.1.5/library/internet.html.