Chapter 14

Files

This chapter introduces the idea of "persistent" programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2 Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file. The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

14.3 Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value `42`.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

For more information on the format operator, see https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting. A more powerful alternative is the string format method, which you can read about at https://docs.python.org/3/library/stdtypes.html#str.format.

14.4 Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The os module provides functions for working with files and directories ("os" stands for "operating system"). os.getcwd returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

cwd stands for "current working directory". The result in this example is /home/dinsdale, which is the home directory of a user named dinsdale.

A string like '/home/dinsdale' that identifies a file or directory is called a **path**.

A simple filename, like memo.txt is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is /home/dinsdale, the filename memo.txt would refer to /home/dinsdale/memo.txt.

A path that begins with / does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use os.path.abspath:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

The `os` module provides a function called `walk` that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from http://thinkpython2.com/code/walk.py.

14.5 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does. The syntax is similar to an `if...else` statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

14.6 Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like `keys` and `items`, don't work with database objects. But iteration with a `for` loop works:

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7 Pickling

A limitation of `dbm` is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The `pickle` module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

`pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for `pickle` to interpret. `pickle.loads` ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

14.8 Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen`[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

[1] `popen` is deprecated now, which means we are supposed to stop using it and start using the `subprocess` module. But for simple cases, I find `subprocess` more complicated than necessary. So I am going to keep using `popen` until they take it away.

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a "checksum". You can read about MD5 at `http://en.wikipedia.org/wiki/Md5`. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

14.10 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at `http://en.wikipedia.org/wiki/Newline`. Or, of course, you could write one yourself.

14.11 Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

14.12 Exercises

Exercise 14.1. *Write a function called* `sed` *that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.*

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution: `http://thinkpython2.com/code/sed.py`.

Exercise 14.2. *If you download my solution to Exercise 12.2 from* `http://thinkpython2.com/code/anagram_sets.py`*, you'll see that it creates a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example,* `'opst'` *maps to the list* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Write a module that imports `anagram_sets` *and provides two new functions:* `store_anagrams` *should store the anagram dictionary in a "shelf";* `read_anagrams` *should look up a word and return a list of its anagrams. Solution:* `http://thinkpython2.com/code/anagram_db.py`.

Exercise 14.3. *In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.*

1. *Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like* `.mp3`*). Hint:* `os.path` *provides several useful functions for manipulating file and path names.*

2. *To recognize duplicates, you can use* `md5sum` *to compute a "checksum" for each files. If two files have the same checksum, they probably have the same contents.*

3. *To double-check, you can use the Unix command* `diff`*.*

Solution: `http://thinkpython2.com/code/find_duplicates.py`.