Chapter 10

Files

10.1 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 non-volatile 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 chapters 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.

10.2 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. To read a file, you can use `open` to create a file object:

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7eb2380>
```

Mode `'r'` means that this file is open for reading. The file object provides several methods for reading data, including `readline`:

```
>>> line = fin.readline()
>>> print line
aa
```

The file object keeps track of where it is in the file, so if you invoke `readline` again, it picks up from where it left off. You can also use a file object in a for loop, as we saw in Section 19.1.

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

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

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.

The `write` method puts data into the file.

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

Again, the file object keeps track of where it is, so if you call `write` again, it add the new data to the end.

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

When you are done writing, you have to close the file.

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

10.3 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
>>> f.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**, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string.

As an example, the **format sequence** `'%d'` means that the first expression in the tuple should be formatted as an integer (`d` stands for "decimal"):

```
>>> 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 format string, so you can embed a value in a sentence:

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

The format sequence `'%g'` formats the next element in the tuple as a floating-point number (don't ask why), and `'%s'` formats the next item as a string:

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

By default, the floating-point format prints six decimal places.

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: illegal argument type for built-in operation
```

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

You can specify the number of digits as part of the format sequence. For example, the sequence `'%8.2f'` formats a floating-point number to be 8 characters long, with 2 digits after the decimal point:

```
>>> '%8.2f' % 3.14159
'    3.14'
```

The result takes up eight spaces with two digits after the decimal point.

10.4 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 create a new file with open, the new file goes in the current directory. And when you open a file for reading, Python looks for it in the current directory.

The module os 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()
>>> print 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 cwd that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use abspath, which is in the module os.path.

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

os.path.exists checks whether the file (or directory) specified by a path 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('music')
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(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, 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.

Exercise 10.1. *Modify* `walk` *so that instead of printing the names of the files, it returns a list of names.*

10.5 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')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

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

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

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 (based on the last error message, 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` statement:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
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 executes 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.

10.6 Databases

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

The module `anydbm` 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 anydbm
>>> db = anydbm.open('captions.db', '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. If you create a new item, `anydbm` updates the database file.

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

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

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

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

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

Many dictionary methods, like `keys` and `items`, also work with database objects. So does iteration with a `for` statement.

```
for key in db:
     print key
```

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

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

10.7 Pickling

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

But 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)
'(lp0\nI1\naI2\naI3\na.'
```

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)
>>> print t2
[1, 2, 3]
```

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

```
>>> t == t2
True
>>> t 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.

Exercise 10.2. *If you did Exercise 17.3, modify your solution so that it creates a database that maps from each word in the list to a list of words that use the same set of letters.*

Write a different program that opens the database and prints the contents in a human-readable format.

10.8 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 non-volatile 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.