Unit 7

Working with Files

Learning Outcomes

- Create a program that outputs information to a text file.
- Create a program that reads a text file and does some simple processing of its contents.
- Describe the role on the operating system.
- Explain in general terms how a disk stores information.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 11 in *How to Think Like a Computer Scientist*.

Topic 7.1 File Output

Up to this point, we have had many ways to manipulate information in the computer's memory, but haven't had any way to write information to a file on a disk. Doing so will allow us to save information permanently, and create files that can be loaded into other programs.

We will only discuss reading and writing *text files* in this course. These are files that consist only of ASCII characters. These files can be opened and edited with a *text editor* like the IDLE editor, Notepad, or Simpletext.

```
dataout = file("sample.txt", "w")

dataout.write("Some text\n")
dataout.write("...that is written to a file\n")

dataout.close()
```

Figure 7.1: Writing text to a file

```
Some text
...that is written to a file
```

Figure 7.2: File created by Figure 7.1

Writing data to a text file is quite easy in Python. Writing a text file is a lot like printing text to the screen with `print`.

Before we can send information to a file, it must be *opened*. The `file` function opens a file on the disk for our use, and returns a *file object* that represents the file. For example,

```
dataout = file("sample.txt", "w")
```

This opens file file `sample.txt` in the current directory for write (the `"w"` indicates that we will write to the file). A file object is returned and stored in `dataout`. Note that when a file is opened for write, any existing contents are discarded.

When you're done with a file object, is should be closed with the `close` method. This writes all of the data to the file and frees it up so other programs can use it. For example,

```
dataout.close()
```

When we have a file object opened for write, we can send text to it. This text will be stored in the file on disk. The standard way to write text to a file object is to use its `write` method. The `write` method takes a string as its argument, and the characters in the string are written to the file.

See Figure 7.1 for a complete program that creates a text file. This program produces a file `sample.txt`; its contents can be seen in Figure 7.2.

In the two calls to the `write` method, you can see that a *newline character* is produced to indicate a line break should be written to the file. In a Python string, `\n` is used to indicate a newline character.

```
import math
csvfile = file("sample.csv", "w")

for count in range(10):
    csvfile.write( str(count) + "," )
    csvfile.write( str(count*count) + "," )
    csvfile.write( str(math.sqrt(count)) + "\n" )

csvfile.close()
```

Figure 7.3: Writing text to a file

When using `print`, a newline character is automatically generated after each statement. When using the `write` method, this must be done manually. A line break is produced for every `\n` in the argument. For example, there would be a blank line between the two lines of text if this statement had been used:

```
dataout.write("Some text\n\n")
```

◈ There is actually a form of the `print` statement that can be used on files. The statement `print >>dataout, "Hello world"` will "print" to the file object `dataout`.

Text files in specific formats can be used by many programs. As long as you know how information must be arranged in the file, it's possible to write a Python program to create a file than can then be imported/loaded in another program. This can be very useful in many applications.

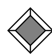
Spreadsheet programs can import *comma-separated value* (or *CSV*) files. The format for these files is simple. Each line in the file represents a row in the spreadsheet; the cells on each line are separated by commas.

It is quite easy to produce this format in a program. For example, the program in Figure 7.3 produces a CSV file with three columns: the first column is a number, the second is the square of the number, and the third is the square root. A row is produced for every number in `range(10)`.

The output of this program can be seen in Figure 7.4. This file can be loaded into any spreadsheet program, or any other program that can import CSV files. This way, a data produced in a Python program can be passed to a spreadsheet or other program for further manipulation or analysis.

```
0,0,0.0
1,1,1.0
2,4,1.41421356237
3,9,1.73205080757
4,16,2.0
5,25,2.2360679775
6,36,2.44948974278
7,49,2.64575131106
8,64,2.82842712475
9,81,3.0
```

Figure 7.4: File created by Figure 7.3



The `csv` module contains objects that allow even more convenient reading and writing of CSV files. It correctly handles cells that contain commas and other special characters, which is tricky to do by-hand.

Check-Up Questions

- Run the program in Figure 7.3; it will produce a file sample.csv. Load this into a spreadsheet program.

Topic 7.2 File Input

Reading data from a file is somewhat like getting input from the user with `raw_input`. The same problems arise: bad input and extracting the information we want from the string of characters. But, unlike user input, we can't just ask the question again if we get bad input. We either have to process the data in the file or fail outright.

A file can be opened for reading the same way as for writing, except we use `"r"` to indicate that we want to read the file.

```
datain = file("sample.txt", "r")
```

Again, a file object is returned and we store it in the variable `datain`. File input objects should also be closed when you're done using them.

```
filename = raw_input("Enter the file name: ")
datain = file(filename, "r")

for line in datain:
    print len(line)

datain.close()
```

Figure 7.5: Reading a text file and counting line lengths

The usual way to read a text file in Python is to process it one line at a time. Files can be very large, so we probably don't want to read the whole file into memory at once. Handling one line at a time doesn't use too much memory, and is often the most useful way to look at the file anyway.

In Topics 5.2 and 5.4, we saw that the `for` loop in Python can be used to iterate over every item in any list or string. It can also be used to iterate through file objects. The body of the `for` is executed once for every line in the file.

For example, the code in Figure 7.5 reads a text file and prints the number of characters on each line. If we give it the file from Figure 7.2, the output would be:

```
Enter the file name: sample.txt
10
29
```

Again, the `for` loop reads just like what it actually does: "`for` (every) `line in` (the) `datain` (file object)...".

Processing File Input

Have a more careful look at the text in Figure 7.2, and the sample output of Figure 7.5 above. The first line in Figure 7.2 is "`Some text`" (9 characters), but the program reports the length of the line as 10. Where did that extra character come from?

Look back at the code in Figure 7.1 that produced the file. The first call to `write` actually produces 10 characters: 9 "visible" characters and a newline. It we modified Figure 7.5 to just output the contents of `line`

instead of the length, we would see an extra line break caused by the newline character in the string.

There are a couple of ways to deal with the newline character if you don't want it in the string as you process it. The easiest is to just discard the last character of each line:

```
for line in datain:
    line = line[:-1]
    ...
```

Assuming there's a newline on the *last* line of the file, this will work. If you want to get rid of the newline and *any other* spaces or tabs at the end of each line, the `rstrip` string method can do that:

```
for line in datain:
    line = line.rstrip()
    ...
```

Remember that this removes *any* trailing whitespace, which may not be appropriate in all situations.

Once the newline character has been removed (if necessary), you can process the line as you would with any other string. Of course, there are many different ways you might need to handle the lines of a file.

As an example, we will read a file that contains a time on each line: each time will be in *hh*:*mm*:*ss* format. We will calculate and report the total number of seconds represented by each time. For example, `1:00:00` is $60 \times 60 = 3600$ seconds.

In order to do this, we will read each line of the file (as in Figure 7.5). Any trailing whitespace will be removed from the line with `rstrip`. Finally, the line can be split into hour, minute, second sections by the `split` string method.

The `split` method is used to divide a string into "words" that are separated by a given *delimiter*. In this case, we want to divide the string around any colon characters, so the method call will be `string.split(":")`. This will return substrings for the hour, minute, and second, in order.

Once we have strings for each of the hour, minute, and second, these can be converted to integers and the number of seconds easily calculated. This has been done in Figure 7.6. Figure 7.7 contains a sample input file, and Figure 7.8 contains the output when the program is run.

```
timefile = file("times.txt", "r")
total_secs = 0

for line in timefile:
    # break line up into hours, minutes, seconds
    h,m,s = line.rstrip().split(":")

    # calculate total seconds on this line
    secs = 3600*int(h) + 60*int(m) + int(s)
    total_secs += secs
    print secs

timefile.close()
print "Total:", total_secs, "seconds"
```

Figure 7.6: Program to read a series of times in a file

```
2:34:27
0:58:10
01:09:56
0:23:01
10:12:00
```

Figure 7.7: Input file (times.txt) for Figure 7.6

```
9267
3490
4196
1381
36720
Total: 55054 seconds
```

Figure 7.8: Output of Figure 7.6

Topic 7.3 The Operating System

In Topic 7.2, you didn't have to know what part of the disk contained the file, you only had to know its file name to get at its contents.

Similarly, in the Topic 7.1, when you wanted to write data to a file, you didn't have to actually know how information was arranged on the disk or what part of the disk your file was being written to. When your data is written to the disk, something has to make sure you're given a part of the disk that isn't being used by another file; if you're using an existing file name, the old version has to be overwritten; and so on.

All of this is taken care of by the *operating system*. The operating system is a piece of software that takes care of all communication with the computer's hardware. The operating system handles all communications with the hard disk, printer, monitor, and other pieces of hardware. Thus, it can make sure that no two application programs are using the same resource at the same time.

When we write files, we are relying on the operating system to give us parts of the hard disk, and put our data there so we can retrieve it later. Since the OS takes care of all of this, we don't have to worry about it when we're programming. The operating system is also responsible for allocating parts of the computer's memory to particular programs; this is necessary when we use variables in a program.

Modern operating systems come bundled with many applications: file managers, Wordpad, Media Player, iPhoto, and so on. Microsoft has even claimed that some of these (notably, Internet Explorer) are inseparable from the operating system. Still, they are application programs, not really part of the operating system, according to its definition. Whether or not they can be separated from the OS is another problem that has more to do with marketing than computing science.

So, what really separates the operating system from other software is that the OS does all of the communication with hardware. It also mediates conflicts (if two applications want to access the hard disk at the same time) and allocates resources (giving out memory and processor time as needed). Figure 7.9 summarizes the communication between the various pieces of the system.

In Figure 7.9, there is one user (Langdon) who has four applications open.

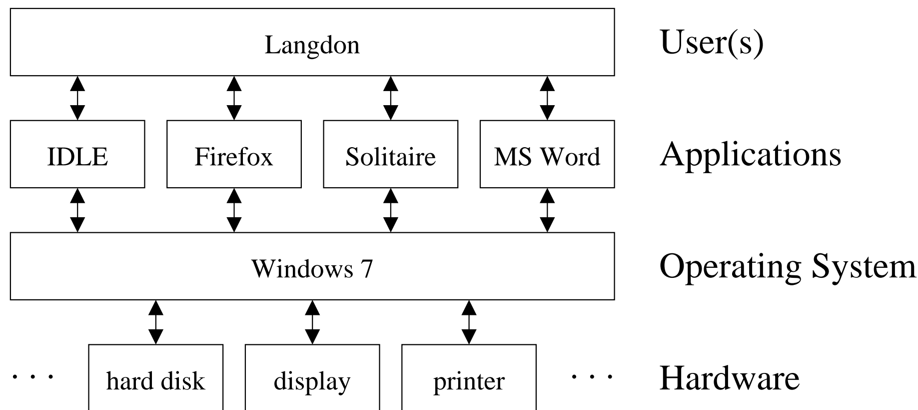


Figure 7.9: Communication between the user, applications, OS, and hardware

Whenever any of these applications needs to interact with the computer's hardware (open a file, draw something on the screen, get keyboard input, and so on), it makes a request through the operating system. The operating system knows how to talk to the hardware (through *device drivers*) and fills these requests.

Having the OS in the middle means that the applications don't have to worry about the details of the hardware. If an application wants to print, it just asks the OS to do the dirty work. If not for the OS, every application would have to have its own printer drivers and it would be very hard to avoid problems if several tried to print at once.

◈ The role of the operating system and how it does its job are explored further in CMPT 300 (Operating Systems). The interaction between hardware and the OS is discussed briefly in CMPT 250 (Computer Architecture).

Topic 7.4 Disks and Files

When we read and wrote files in Topics 7.1 and 7.2, we took for granted that the operating system could put the information on the disk and get it back later. This is no small job for the OS to do—we should look a little more at what happens.

Note that whenever we're talking about storing information, a *disk* can refer to any device you can use in a computer to store information. The storage must keep the information when the computer is turned off, so the computer's memory doesn't count. These are referred to as *nonvolatile storage*. They include:

- *hard drive*: fast high capacity storage that is used to store the operating system, applications, and most of your data on a computer.
- *floppy disk*: slow low capacity disks that can be easily transported.
- *flash media cards*: small storage devices with no moving parts. These are often used with digital cameras, MP3 players, and other portable devices
- *USB "disks"*: small keychain-sized devices that can be connected directly to a USB port on a computer and then easily transported to another.
- *MP3 player* or *digital camera*: These can often be connected directly to your computer and transfer information to and from built-in storage (or any inserted media cards).
- *compact disc*: used to distribute programs and other information since they are high capacity and cheap to produce. You can't write to compact discs (at least, not quite the same way you can to the other devices listed here).

All of these are treated as "disks" by the operating system. As far as the user and programmer are concerned, they all work the same way. The operating system makes sure they all behave the same way as far as the user or programmer is concerned, regardless of how they work.

Once the operating system knows how to physically store information on all of these "disks", it still has to arrange the information so that it can find it later. When your program asks for a particular file, the computer has to know what information on the disk corresponds to that file; if you change the file, it has to change the information on the disk, adding or removing the space reserved for the file as necessary.

The operating system arranges information on the disk into a *file system*. A file system is a specification of how information is stored on a disk, how to store directories or folders, information about the files (last modified date

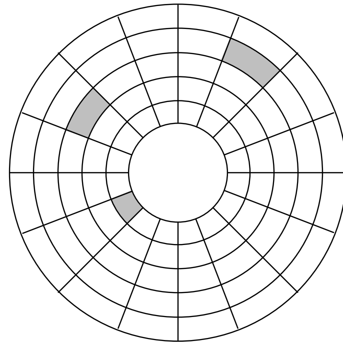


Figure 7.10: A disk divided into blocks

and access permissions, for example), and any other information that the computer has to store to keep everything working.

The space on the disk is divided up into *disk blocks*. The block size can vary, but is most commonly 4 kB. The blocks are then allocated to the various files that are stored on the disk. Figure 7.10 shows that the blocks on a disk might look like.

Figure 7.10 could be either a hard disk or floppy disk. The insides look the same, but information can be stored much more densely on a hard disk because it's sealed in its enclosure. A *read head* sits above the disk surface and can read information from the disk as it spins below it. The time it takes to get information from the disk depends on two factors: the amount of time it takes to position the head so it's the right distance from the centre of the disk, and the speed the disk is rotating.

If the filesystem uses 4 kB blocks, a 10 kB file would need three blocks. That means that the file is actually using 12 kB of disk space—the last 2 kB in the last block are wasted. This is refereed to as *internal fragmentation*. Internal fragmentation occurs whenever some of the disk block is left over after the file is written—every file on the disk (unless it *exactly* fills the block) will cause a little internal fragmentation.

When the computer tries to read this file, it will have to read the information from three widely separated blocks on the disk. This will be slow since you have to wait for the read head to move and disk to spin to the right position three times.

If you *defragment* your hard drive, it repairs *external fragmentation* and moves scattered blocks like this together. In the example, we might end up

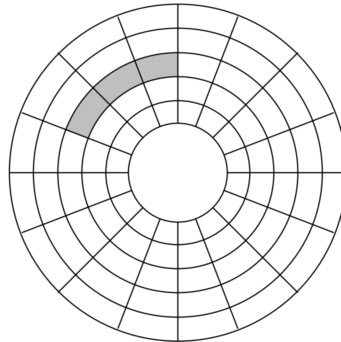


Figure 7.11: The blocks of a file defragmented

with the three-block file stored as in Figure 7.11. Now, reading them will be much faster: you only have to move the read head once and the three block will spin under the read head quickly since they are adjacent.

Topic 7.5 Example Problem Solving: File Statistics

As an example of using file input, we will create a program that opens up a text file specified by the user and outputs some statistics about it. We want the program to count the number of lines, words, and characters in the file.

The first step will be to get the file name and open the file. This has been done in Figure 7.12. This program also loops through the lines in the file, counting as it goes.

We can now test this program with a sample file, as seen in Figure 7.13. When we run the program in Figure 7.12 and give it this file, the output is:

```
Filename: wc-test.txt
Total lines: 6
```

There are six lines in the input file, so we're on the right track.

Next, we can try to count the number of characters. Since each line in the file is a string, we can just use `len` to get the number of characters in the string and add it to another counter. This has been done in Figure 7.14.

```
# get the filename and open it
filename = raw_input("Filename: ")
file = open(filename, "r")

# initialize the counter
total_lines = 0

for line in file:
    # do the counting
    total_lines += 1

# summary output
print "Total lines:", total_lines
```

Figure 7.12: Word count: open file and count lines

```
"One trick is to tell them stories that don't go anywhere, like
the time I caught the ferry over to Shelbyville.  I needed a new
heel for my shoe, so I decided to go to Morganville, which is
what they called Shelbyville in those days.  So I tied an onion
to my belt, which was the style at the time..."
- Abe
```

Figure 7.13: Word count test file

```
# get the filename and open it
filename = raw_input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0

for line in file:
    # do the counting
    total_lines += 1
    total_chars += len(line)
    print total_chars

# summary output
print "Total lines:", total_lines
print "Total characters:", total_chars
```

Figure 7.14: Word count: counting characters 1

Notice that a `print` statement has been added to the loop to help with debugging. When this program is run on our sample data file, we get this output:

```
Filename: wc-test.txt
64
129
191
255
305
311
Total lines: 6
Total characters: 311
```

It's not easy to check if this is correct or not. The program probably should be tested on a smaller file, where we can actually count the number of characters by hand to verify. But, the debugging output has given us something to work with.

Look at the last line of Figure 7.13. It contains five characters: dash, space, A, b, e. Why did our program count $311 - 305 = 6$ characters on that

```
# get the filename and open it
filename = raw_input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0

for line in file:
    # clean any trailing whitespace off the string
    line = line.rstrip()

    # do the counting
    total_lines += 1
    total_chars += len(line)
    print total_chars

# summary output
print "Total lines:", total_lines
print "Total characters:", total_chars
```

Figure 7.15: Word count: counting characters 2

line?

Like we saw in Topic 7.2, reading the lines from the file includes the newline characters. But, we don't want to count the "invisible" characters in the file. We can use `rstrip` to get rid of these, along with any trailing spaces, before we do the counting. The program in Figure 7.15 does this.

Now when we run the program on the sample file, we get this output:

```
Filename: wc-test.txt
63
127
188
251
298
303
Total lines: 6
Total characters: 303
```

Now we get the right number of characters on the last line. Testing with some other sample files confirms that we are now counting the characters on each line properly.

We can now turn to the problem of counting the number of words on each line. Your first thought might be to just count the number of spaces on the line, but that won't work. If there are several spaces together (the test file has two spaces after each period), then that will count as two "words".

In order to count the number of words in the line, we will to check for *the beginning of each word*. This takes some more careful examination of the string. Since it's more complicated than the other counting we've done, it has been split into a separate function, `words`.

The function `words` characterizes the "start of a word" as a non-space character after by either the start of the string or a space.

See Figure 7.16 for the implementation. Since it's in a function, we can test it separately:

```
>>> words("abc-def ghi")
2
>>> words("abcde f      ghijkl")
3
>>> words("... ,,,    :::")
3
>>> words("")
0
```

The final program is assembled in Figure 7.16. If we run it on our test program, we get this output:

```
Filename: wc-test.txt
Total lines: 6
Total words: 62
Total characters: 303
```

Check-Up Questions

- ▶ Try Figure 7.15 on a text file with a few shorter lines in it. Is it counting the number of characters correctly?
- ▶ Make a copy of Figure 7.16. Print out the number of words after each iteration of the `for line in file` loop. Does it match the number of words you count on each line?

```
def words(line):
    """
    Count the number of words in the string line.
    """
    words = 0
    for pos in range(len(line)):
        # line[pos] is the start of a word if it is a non-space
        # and it is either the start of string or comes after
        # a space
        if line[pos]!=" " and (pos==0 or line[pos-1]==" "):
            words += 1

    return words

# get the filename and open it
filename = raw_input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0
total_words = 0

for line in file:
    # clean any trailing whitespace off the string
    line = line.rstrip()

    # do the counting
    total_lines += 1
    total_chars += len(line)
    total_words += words(line)

# summary output
print "Total lines:", total_lines
print "Total words:", total_words
print "Total characters:", total_chars
```

Figure 7.16: Word count: final program

▶ In Figure 7.16, the `if` condition in the `words` function checks `pos+1` against `len(line)-1`. What is the purpose of the plus one and minus one?

Summary

Working with files gives you a way to "save" things in your program. Information you put in a file can be read back in the next time the program runs.

Hopefully you have an idea of how to read and write simple text files with a Python program. You should also have some idea of what actually happens when a program, one you write or any other, stores information on a disk.

Key Terms

- text file
- file object
- newline character
- operating system
- disk
- file system
- disk block
- internal fragmentation
- external fragmentation