CSC110 Supplemental Reading: Week08

Contents

1 File I/O

Files allow a program to work with stored information or to store information for later use. In Python, files are another kind of object.

1.1 Opening a File

To work with a file, the first thing we must do is open the file. We do this using the open built-in function. The first parameter is the name of the file we want to open. The second parameter is the mode for the file. This parameter is optional. The mode we choose is based on what we want to do with the file. There are three modes of interest to us for the purposes of this course: 'r' - open for reading into the program, 'w' - open for writing, and 'a' - open for appending. The default mode is 'r' - open for reading.

```
f = open('mydata.txt')
g = open('otherdata.txt', 'r')
h = open('newstuff.txt', 'w')
i = open('oldstuff.txt', 'a')
```

Notice that the mode is specified as a string. The variable f is now a identifier for input. It supports reading from the file mydata.txt. Similarly, the variable g is a identifier for input. It supports reading from the file otherdata.txt. Both variables h and i are identifiers for output. They support writing to the files newstuff.txt and oldstuff.txt, respectively.

The difference between 'w' and 'a' is significant. The 'w' mode ***creates*** a file for writing output. If there is already a file with that name, the 'w' option will write over the existing file, no questions asked. The 'a' mode opens the file to *append* (at to the end). If the file exists, the data in the file is preserved. Note that if the file does not exist, both the 'a' mode and 'w' mode will create the file.

All of the files that we'll be working with in this class will be text files. So, the file will be opened to read and write string values, bunches of characters. There are other modes available for working explicitly with other types of files.

Now that the file is open, you will use the "file identifier" stored in the variable to manipulate the file.

A note about file names. You can give folder/directory information in the string for the file name. To do this, you can use the standard access strings that your operating system uses. On Unix/Linux and Mac OSX, the typical folder delimiter is the forward slash, (/). On Windows, the classic character is the backslash, (\). This poses a special problem, since the backslash character is the string escape character. You can use the prefix `r` before the string to tell Python to treat the string as raw input (not to think of the backslash as an escape character). For example:

```
inputFile = open(r'C:\someFolder\someFile.txt', 'a')
```

Another option, to get a literal backslash character, you can use a double backslash, (\\).

As the programmer, you need to use a double backslash or the r prefix if you want to type the backslash in your Python code. However, when the user is typing in a file name, they **don't** double the backslash. The simplest solution is to use a *forward slash* in your filenames, instead of the backslash (yes, even on your Windows machines). They work just fine for opening files, etc. and you don't have to hassle with deciding when you need to use double backslashes.

```
f = open('c:/teaching/csc110/data.txt')
```

1.2 Closing a File

It is a good practice to close the file when you're done using it. To close a file, use the close method. In fact, typical practice is to open the file, read or write what you need, and then close the file.

```
f = open(filename)
# reading stuff from the file
f.close()
```

Another reason to remember to close your files comes from the fact that file I/O is slow. It takes the computer "a lot" of time to read or write to the disk. So, when you are working with a file, the computer typically works with a largish chunk of data, called a block, reading or writing whole blocks of data. This is pretty transparent to us when we're reading a file. But, when we're writing a file, the stuff that we've "written" to the file may just be in temporary storage within RAM (called a buffer) waiting for us to fill up the block before actually writing it out to disk. The close method forces anything that's in the buffer to be written to disk before closing the file. Now, an orderly shut-down of the program will also close the file, flushing (yes, that's the word they use) the buffer. But, if your program crashes because of some error … your data may still be in the buffer, never to be seen again.

1.3 Writing to a File

The write method adds a string to the file. The biggest 'gotcha' with the write method is that it does ***not*** automatically add a newline character, the way that the print statement does. So, you could easily end up with one *very long* line if you don't explicitly put in newline characters.

```
f = open(filename, 'w')
f.write('This is line 1\n')
s = 'This is line 2'
f.write(s + '\n')
f.write('this is line 3')
f.write(', written using ')
f.write('multiple calls\n')
f.close()
```

1.4 Reading from a File

There are several ways to read from a file. The read method will read the whole file in one fell swoop.

```
f = open(filename)
data = f.read()
f.close()
```

Alternately, you can read in a given number of characters by passing an argument to read. It will read that many characters, if it can. When it comes to the end of the file, it just stops reading. If you call read again, it returns an empty string.

Probably the most useful way to read a file is one line at a time. (It's a text file after all.) The readline method reads a line from the file. All of these methods include the newline character in the stuff read from the file. In the case of the readline method, the newline is always the last character in the string returned. If it's a blank line, then the only character in the string is the newline. As with the other reading methods, when the end of file (EOF) is encountered, the return value is an empty string.

1.5 Processing through a File

The book does a good job of illustrating the for loop to read in lines from a file. Another typical pattern for working with a file uses the *while* loop. The loop takes care of reading the file. So, the file should be opened before the loop, and closed after the loop. Notice that each of the reading methods returns an empty string to signal EOF.

```
# open the file for reading
f = open(filename)
# a priming read
line = f.readline()
while line ! = '': # testing that we don't have the empty string
    # remove the white space at the end, then print
    line = line.rstrip('\n')
    print (line)

    # get the next line
    line = f.readline()
# close the file
f.close()
```

An important thing to remember when using a while loop is that immediately after a readline() is executed, you want the code to test whether it was the EOF or not. Notice the 2 places where this code calls readline() and understand that the line executed immediately afterwards is the test of the while loop.