File Operations in Python

Leigh J. Little SUNY Brockport November 16, 2020

Contents

1 Introduction

This chapter discusses working with data files in Python. This is an important component to any programming language. The data that a program needs is frequently obtained from the output from some other program. Python programs need to be able to read this data. Similarly it is necessary to be able to save the important portions of a computer simulation so that they can be used at some future date. This data needs to be written to a data file.

1.1 Types of Data Files

Data files can generally be broken down into two types; *ascii* and *binary*.

- ascii - ascii files are also known as plain-text files. ascii is an acronym for American Standard Code for Information Interchange and has been the dominant method for assigning letters and other characters to computer numbers for more than 50 years. Every computer character needs to have a uniqie ascii code. For example, a capital A has ascii code 65. Letters, spaces, tabs and special symbols all have ascii codes.

  The most important characteristic of an ascii file is that if you open it in a text editor (like Notepad, Notepad++, the spyder editor, etc.), you will see characters that look like plain English (most of the time).

- Binary - The contents of a binary file are more complex. If you open a binary data file in a text editor, you will often see one long line of mostly undescipherable characters. When a binary file is created, its contents are written directly as a long sequence of 0's and 1's (though hexadecimal is often used to look at binary files). The main advantage to binary files is that they are often smaller in size than if the contents were written in ascii format. This can be critical for simulations that generate terabytes of computational results.

In this class we will be concentrating on ascii files.

2 Opening and Closing a File

Before you can read from or write to a file, it is necessary to first open the file. In Python, this is done using the `open` command. The format of this command is

```
f = open(name, option)
```

Here, `name` is the name of the file and `option` is how the file should be opened. The variable `f` is called a file handle. It gives you a way to refer to the file in your programs. You can have multiple files open at once, but you need to have a different variable for each file. The `option` argument can have one of four values:

- `'r'` - this means the file should be open for reading only. If you try to write to a file that was opened with this option, an error will be generated. The file must also exist. For now we will assume that the file is in the current directory where your script it located.

- `'w'` - this means the file should be open for writing only. If you try to read from a file that was opened with this option, an error will be generated. If the file does not exist, it will be created. Any new write operations will overwrite existing contents.
- `'r+'` - this means the file should be open as read/write capable. You can read or write to the file. Any new write operations will overwrite existing contents.
- `'a'` - this opens the file in append mode. Any write operations will be appended to the file.

Any file that is opened should always be closed prior to terminating the program. To close a file, use the syntax

```
f.close()
```

Here, we are applying the `.close()` method to the file handle `f`.

2.1 Example 1: Simple Read from File

As a simple example, we will write a script that will open the file `file1.txt`, read the contents of the file and display the contents on the screen. Download the `file1.txt` file from the website into your working directory. Enter the code below:

```
f = open('file1.txt','r')

t = next(f)
print(t)
t = next(f)
print(t)
t = next(f)
print(t)
f.close()
```

You should see the output

```
This is line 1

This is line 2

This is line 3
```

This script performs three read operations using the `next` function. The output was stored in the variable `t` and then printed to the screen. You will note that there are blank lines in the output, but not in the data file. We will discuss why this happens later on. Finally, we close the file before ending the script/

In this case, we could easily look at the data file to see how many lines were in the file. What if we performed 4 read operations? Add in one more read before closing the file and you will get the the error

```
This is line 1

This is line 2

This is line 3

Traceback (most recent call last):

File "....."', line 12, in <module>
t = next(f)

StopIteration
```

In this case an error is generated because we tried to read past the end of the file. This is known as an *end-of-file* or EOF error.

2.2 A Better Technique for Reading from Files

The approach in the previous section causes something of a problem because we don't want to have to manually open a file and count the lines in order to avoid EOF errors. A data file can have a million or more lines.

A better way to handle this is to use a special looping structure designed for reading files of unknown length. Enter the code below into a script and run it.

```
with open('file1.txt','r') as f:
     for t in f:
          print(t)
```

You should see the same output as from before. In this example, we are using the `with` construct to open the file handle `f`. This structure is similar to loops and if-then statements. All commands to be performed while the file is open should be indented. This structure will also automatically close the file once a statement indented to the same level as `with` is executed. Note that no EOF error was generated here. The contents of the file are read line by line until the end of the file is reached.

2.3 A Note Regarding Python 2 versus Python 3

Recently, the Python language underwent a significant change from Version 2 to Version 3. A large number of new features were added and some third party libraries were consolidated. As a result, some commands look different between the two versions. At this point, you should always use Python 3 syntax instead of Python 2 syntax unless you are modifying a Python 2 code. As an example, consider the

```
next(f)
```

command in the example in Section 2.1. This is the Python 3 version of this operation. In Python 2, the syntax for this operation would be

```
f.next()
```

If you try to use the above syntax in a Python 3 code, you will get a runtime error. This can make working with Python difficult, especially if you don't realize that you are looking at Python 2 code.

2.4 Example 2: Simple Write to a File

As an example of writing data to a file, we will generate a simple table of values and save these to a file. Enter the following into a script file and run it:

```
f = open('file2.txt','w')
for i in range(1,11):
     x = [i, i**2]
     f.write(x)
f.close()
```

You will see that an error is generated. This is because the `f.write()` method requires that the quantity being written be a string variable instead of a list. We can fix this (kind of) by doing

```
f = open('file2.txt','w')
for i in range(1,11):
     x = [i, i**2]
     f.write(str(x))
f.close()
```

The `str` function converts `x` from a list to a string, so now the program runs. However, if you look at `file2.txt`, you will notice that this is not quite what we are looking for. The contents of `file2.txt` indicate that the output appears all on one line as a series of 10 lists of length 2.

How can we write the data into two columns of data? To do this we will need to know about a special character, `n\`. This is called the *newline* character. This character is not printed to a file or to the screen; rather is is a special instruction to these routines that says 'go to the next line.' To see how it works, change the script to:

```
f = open('file2.txt','w')
for i in range(1,11):
     x = [i, i**]
     y = str(x) + '\n'   # Use the + operator to add \n to the end
     f.write(y)
f.close()
```

In this case, we have appended the newline character to `x`. This causes the next write statement to occur on the next line of the file.

Notice that this still is not quite what we want. We would prefer to not have the square brackets in the output file. How can this be done? It turns out the easiest way to handle this is to create 1 long string.

```
f = open('file2.txt','w')
for i in range(1,11):
     y = str(i) + ',' + str(i**2) + '\n'
     f.write(y)
f.close()
```

As can be seen, writing data to files is not as easy as reading from files. This is one of the reasons that people have written third party libraries to make this process easier.